```
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE TypeSynonymInstances #-}
module DSLsofMath.W06 where
import DSLsofMath.FunExp hiding (eval, f)
import DSLsofMath.W05
import DSLsofMath.Simplify
```

6 Higher-order Derivatives and their Applications

6.1 Review

- key notion *homomorphism*: $S1 \to S2$
- questions ("equations"):
  - $S1 \overset{?}{\leftarrow} S2$ what is the homomorphism between two given structures
    - e.g., $apply\ c : Num\ (x \to a) \to Num\ a$
  - $S1? \to S2$ what is $S1$ compatible with a given homomorphism
    - e.g., $eval : Poly\ a \to (a \to a)$
  - $S1 \to S2?$ what is $S2$ compatible with a given homomorphism
    - e.g., $applyFD\ c : FD\ a \to (a, a)$
  - $S1 \overset{?}{\leftarrow} S2?$ can we find a good structure on $S2$ so that it becomes homomorphic w. $S1$?
    - e.g., $evalD : FunExp \to FD\ a$
- The importance of the last two is that they offer "automatic differentiation", i.e., any function constructed according to the grammar of *FunExp*, can be "lifted" to a function that computes the derivative (e.g., a function on pairs).

Example $f\ x = sin\ x + 2 * x$

We have: $f\ 0 = 0$, $f\ 2 = 4.909297426825682$, etc.

The type of f is $f :: Floating\ a \Rightarrow a \to a$.

How do we compute, say, $f'\ 2$?

We have several choices.

a. Using *FunExp*

   Recall (section 3.8):

```
data FunExp = Const Rational
            | Id
            | FunExp :+: FunExp
            | FunExp :*: FunExp
            | FunExp :/: FunExp
            | Exp FunExp
            | Sin FunExp
            | Cos FunExp
                -- and so on
  deriving (Eq, Show)
```

What is the expression e for which $f = eval\ e$?

We have

$$
\begin{aligned}
&\quad eval\ e\ x = f\ x \\
&\Leftrightarrow \\
&\quad eval\ e\ x = sin\ x + 2 * x \\
&\Leftrightarrow \\
&\quad eval\ e\ x = eval\ (Sin\ Id)\ x + eval\ (Const\ 2 \mathbin{:\!*\!:} Id)\ x \\
&\Leftrightarrow \\
&\quad eval\ e\ x = eval\ ((Sin\ Id) \mathbin{:\!+\!:} (Const\ 2 \mathbin{:\!*\!:} Id))\ x \\
&\Leftarrow \\
&\quad e = Sin\ Id \mathbin{:\!+\!:} (Const\ 2 \mathbin{:\!*\!:} Id)
\end{aligned}
$$

Finally, we can apply *derive* and obtain

$$
\begin{aligned}
&e = Sin\ Id \mathbin{:\!+\!:} (Const\ 2 \mathbin{:\!*\!:} Id) \\
&f'\ 2 = evalFunExp\ (derive\ e)\ 2
\end{aligned}
$$

This can hardly be called "automatic", look at all the work we did in deducing e!

However, consider this definition:

$$
\begin{aligned}
&e_2 :: FunExp \\
&e_2 = f\ Id
\end{aligned}
$$

As $Id :: FunExp$, Haskell will look for *FunExp* instances of *Num* and friends and build the syntax tree for f instead of computing its semantic value. (Perhaps it would have been better to use, in the definition of *FunExp*, X instead of *Id*.)

In general, to find the value of the derivative of a function f at a given x, we can use

$$drv\ f\ x = evalFunExp\ (derive\ (f\ Id))\ x$$

b. Using *FD*

Recall

$$
\begin{aligned}
&\textbf{type}\ FD\ a = (a \to a, a \to a) \\
&applyFD\ x\ (f, g) = (f\ x, g\ x)
\end{aligned}
$$

The operations on $FD\ a$ are such that, if $eval\ e = f$, then

$$(eval\ e, eval'\ e) = (f, f')$$

We are looking for (g, g') such that

$$f\ (g, g') = (f, f') \quad \text{-- (*)}$$

so we can then do

$$f'\ 2 = snd\ (applyFD\ 2\ (f\ (g, g')))$$

We can fullfill (*) if we can find a (g, g') that is a sort of "unit" for $FD\ a$:

$$sin\ (g, g') = (sin, cos)$$
$$exp\ (g, g') = (exp, exp)$$

and so on.

In general, the chain rule gives us

$$f\ (g, g') = (f \circ g, (f' \circ g) * g')$$

Therefore, we need: $g = id$ and $g' = const\ 1$.

Finally

$$f'\ 2 = snd\ (applyFD\ 2\ (f\ (id, const\ 1)))$$

In general

$$drvFD\ f\ x = snd\ (applyFD\ x\ (f\ (id, const\ 1)))$$

computes the derivative of f at x.

$$f_1 :: FD\ Double \rightarrow FD\ Double$$
$$f_1 = f$$

c. Using pairs

We have **instance** *Floating* $a \Rightarrow$ *Floating* (a, a), moreover, the instance declaration looks exactly the same as that for $FD\ a$:

$$\begin{array}{ll}
\textbf{instance}\ Floating\ a \Rightarrow Floating\ (FD\ a)\ \textbf{where} & \text{-- pairs of functions} \\
\quad exp\ (f, f') = (exp\ f, (exp\ f) * f') & \\
\quad sin\ (f, f') = (sin\ f, (cos\ f) * f') & \\
\quad cos\ (f, f') = (cos\ f, -(sin\ f) * f') & \\
\textbf{instance}\ Floating\ a \Rightarrow Floating\ (a, a)\ \textbf{where} & \text{-- just pairs} \\
\quad exp\ (f, f') = (exp\ f, (exp\ f) * f') & \\
\quad sin\ (f, f') = (sin\ f, cos\ f * f') & \\
\quad cos\ (f, f') = (cos\ f, -(sin\ f) * f') &
\end{array}$$

In fact, the latter represents a generalisation of the former. To see this, note that if we have a *Floating* instance for some A, we get a floating instance for $x \rightarrow A$ for all x from "FunNumInst". Then from the instance for pairs we get an instance for any type of the form $(x \rightarrow A, x \rightarrow A)$. As a special case when $x = A$ this includes all $(A \rightarrow A, A \rightarrow A)$ which is $FD\ A$. Thus it is enough to have "FunNumInst" and the pair instance to get the "pairs of functions" instance (and more).

The pair instance is also the "maximally general" such generalisation (discounting the "noise" generated by the less-than-clean design of *Num*, *Fractional*, *Floating*).

Still, we need to use this machinery. We are now looking for a pair of values (g, g') such that

$$f\ (g, g') = (f\ 2, f'\ 2)$$

In general

$$f\ (g, g') = (f\ g, (f'\ g) * g')$$

Therefore

$$
\begin{aligned}
& f\ (g, g') = (f\ 2, f'\ 2) \\
\Leftrightarrow & \\
& (f\ g, (f'\ g) * g') = (f\ 2, f'\ 2) \\
\Leftarrow & \\
& g = 2, g' = 1
\end{aligned}
$$

Introducing

$$var\ x = (x, 1)$$

we can, as in the case of FD, simplify matters a little:

$$f'\ x = snd\ (f\ (var\ x))$$

In general

$$drvP\ f\ x = snd\ (f\ (x, 1))$$

computes the derivative of f at x.

$$
\begin{aligned}
& f_2 :: (Double, Double) \to (Double, Double) \\
& f_2 = f
\end{aligned}
$$

TODO: some ending sentence for this part

6.2 Higher-order derivatives

Consider

$$[f, f', f'', ...]$$

representing the evaluation of an expression and its derivatives:

$$evalAll\ e = (evalFunExp\ e) : evalAll\ (derive\ e)$$

Notice that, if

$$[f, f', f'', ...] = evalAll\ e$$

then

$$[f', f'', ...] = evalAll\ (derive\ e)$$

We want to define the operations on lists of functions in such a way that *evalAll* is a homomorphism. For example:

$$evalAll\ (e_1 :*: e_2) = evalAll\ e_1 * evalAll\ e_2$$

where the $(*)$ sign stands for the multiplication of infinite lists of functions, the operation we are trying to determine.

We have, writing *eval* for *evalFunExp* in order to save ink

$$
\begin{array}{l}
\quad \mathit{evalAll}\ (e_1 :\!*\!: e_2) = \mathit{evalAll}\ e_1 * \mathit{evalAll}\ e_2 \\
\Leftrightarrow \\
\quad \mathit{eval}\ (e_1 :\!*\!: e_2) : \mathit{evalAll}\ (\mathit{derive}\ (e_1 :\!*\!: e_2)) = \\
\quad \mathit{eval}\ e_1 : \mathit{evalAll}\ (\mathit{derive}\ e) * \mathit{eval}\ e_1 : \mathit{evalAll}\ (\mathit{derive}\ e_2) \\
\Leftrightarrow \\
\quad (\mathit{eval}\ e_1 * \mathit{eval}\ e_2) : \mathit{evalAll}\ (\mathit{derive}\ (e_1 :\!*\!: e_2)) = \\
\quad \mathit{eval}\ e_1 : \mathit{evalAll}\ (\mathit{derive}\ e) * \mathit{eval}\ e_1 : \mathit{evalAll}\ (\mathit{derive}\ e_2) \\
\Leftrightarrow \\
\quad (\mathit{eval}\ e_1 * \mathit{eval}\ e_2) : \mathit{evalAll}\ (\mathit{derive}\ e_1 :\!*\!: e_2 :\!+\!: e_1 * \mathit{derive}\ e_2) = \\
\quad \mathit{eval}\ e_1 : \mathit{evalAll}\ (\mathit{derive}\ e) * \mathit{eval}\ e_1 : \mathit{evalAll}\ (\mathit{derive}\ e_2) \\
\Leftarrow \\
\quad (a : \mathit{as}) * (b : \mathit{bs}) = (a * b) : (\mathit{as} * (b : \mathit{bs}) + (a : \mathit{as}) * \mathit{bs})
\end{array}
$$

The final line represents the definition of $(*)$ needed for ensuring the conditions are met.

As in the case of pairs, we find that we do not need any properties of functions, other than their *Num* structure, so the definitions apply to any infinite list of *Num a*:

$$
\begin{array}{l}
\textbf{type}\ \mathit{Stream}\ a = [a] \\
\textbf{instance}\ \mathit{Num}\ a \Rightarrow \mathit{Num}\ (\mathit{Stream}\ a)\ \textbf{where} \\
\quad (+) = \mathit{addStream} \\
\quad (*) = \mathit{mulStream} \\
\mathit{addStream} :: \mathit{Num}\ a \Rightarrow \mathit{Stream}\ a \to \mathit{Stream}\ a \to \mathit{Stream}\ a \\
\mathit{addStream}\ (a : \mathit{as})\ (b : \mathit{bs}) = (a + b) : (\mathit{as} + \mathit{bs}) \\
\mathit{mulStream} :: \mathit{Num}\ a \Rightarrow \mathit{Stream}\ a \to \mathit{Stream}\ a \to \mathit{Stream}\ a \\
\mathit{mulStream}\ (a : \mathit{as})\ (b : \mathit{bs}) = (a * b)\ : (\mathit{as} * (b : \mathit{bs}) + (a : \mathit{as}) * \mathit{bs})
\end{array}
$$

Exercise: complete the instance declarations for *Fractional* and *Floating*. Note that it may make more sense to declare a **newtype** for *Stream a* first, for at least two reasons. First, because the type $[a]$ also contains finite lists, but we use it here to represent only the infinite lists (also known as streams). Second, because there are competing possibilities for *Num* instances for infinite lists, for example applying all the operations "pointwise" as with "FunNumInst". We used just a type synonym here to avoid cluttering the definitions with the newtype constructors.

Write a general derivative computation, similar to *drv* functions above:

$\mathit{drvList}\ k\ f\ x = \mathit{undefined}$ -- kth derivative of f at x

Exercise: Compare the efficiency of different ways of computing derivatives.

6.3 Polynomials

$$
\begin{array}{l}
\textbf{data}\ \mathit{Poly}\ a = \mathit{Single}\ a \mid \mathit{Cons}\ a\ (\mathit{Poly}\ a) \\
\qquad\qquad\qquad \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord}) \\
\mathit{evalPoly} ::\ \ \mathit{Num}\ a \Rightarrow \mathit{Poly}\ a \to a \to a \\
\mathit{evalPoly}\ (\mathit{Single}\ a)\quad x = a \\
\mathit{evalPoly}\ (\mathit{Cons}\ a\ \mathit{as})\ x = a + x * \mathit{evalPoly}\ \mathit{as}\ x
\end{array}
$$

6.4 Formal power series

As we mentioned above, the Haskell list type contains both finite and infinite lists. The same holds for the type *Poly* that we designed as "syntax" for polynomials. Thus we can reuse that type also as "syntax for power series": potentially infinite "polynomials".

$$\textbf{type}\ \mathit{PowerSeries}\ a = \mathit{Poly}\ a \quad \text{-- finite and infinite non-empty lists}$$

Now we can divide, as well as add and multiply.

We can also compute derivatives:

$$\begin{aligned}
&\mathit{deriv}\ (\mathit{Single}\ a) \quad = \mathit{Single}\ 0\\
&\mathit{deriv}\ (\mathit{Cons}\ a\ as) = \mathit{deriv}'\ as\ 1\\
&\quad \textbf{where}\ \mathit{deriv}'\ (\mathit{Single}\ a) \quad n = \mathit{Single}\ (n * a)\\
&\qquad\qquad \mathit{deriv}'\ (\mathit{Cons}\ a\ as)\ n = \mathit{Cons}\ \ (n * a)\ (\mathit{deriv}'\ as\ (n + 1))
\end{aligned}$$

and integrate:

$$\begin{aligned}
&\mathit{integ} :: \mathit{Fractional}\ a \Rightarrow \mathit{PowerSeries}\ a \to a \to \mathit{PowerSeries}\ a\\
&\mathit{integ}\ as\ a_0 = \mathit{Cons}\ a_0\ (\mathit{integ}'\ as\ 1)\\
&\quad \textbf{where}\ \mathit{integ}'\ (\mathit{Single}\ a) \quad n = \mathit{Single}\ (a\ /\ n)\\
&\qquad\qquad \mathit{integ}'\ (\mathit{Cons}\ a\ as)\ n = \mathit{Cons}\ \ (a\ /\ n)\ (\mathit{integ}'\ as\ (n + 1))
\end{aligned}$$

Note that a_0 is the constant that we need due to indefinite integration.

These operations work on the type *PowerSeries* a which we can see as the syntax of power series, often called "formal power series". The intended semantics of a formal power series a is, as we saw in Chapter 5, an infinite sum

$$\begin{aligned}
&\mathit{eval}\ a : \mathbb{R} \to \mathbb{R}\\
&\mathit{eval}\ a = \lambda x \to \mathit{lim}\ s\ \textbf{where}\ s\ n = \textstyle\sum_{i=0}^{n} a_i * x^i
\end{aligned}$$

For any n, the prefix sum, $s\ n$, is finite and it is easy to see that the derivative and integration operations are well defined. We we take the limit, however, the sum may fail to converge for certain values of x. Fortunately, we can often ignore that, because seen as operations from syntax to syntax, all the operations are well defined, irrespective of convergence.

If the power series involved do converge, then *eval* is a morphism between the formal structure and that of the functions represented:

$$\begin{aligned}
&\mathit{eval}\ as + \mathit{eval}\ bs \quad = \mathit{eval}\ (as + bs) \quad \text{-- } H_2\ (\mathit{eval}, (+), (+))\\
&\mathit{eval}\ as * \mathit{eval}\ bs \quad = \mathit{eval}\ (as * bs) \quad \text{-- } H_2\ (\mathit{eval}, (*), (*))\\
&\mathit{eval}\ (\mathit{derive}\ as) \quad = D\ (\mathit{eval}\ as) \quad \text{-- } H_1\ (\mathit{eval}, \mathit{derive}, D)\\
&\mathit{eval}\ (\mathit{integ}\ as\ c)\ x = \textstyle\int_0^x (\mathit{eval}\ as\ t)\ dt + c
\end{aligned}$$

6.5 Simple differential equations

Many first-order differential equations have the structure

$$f'\ x = g\ f\ x, \qquad f\ 0 = f_0$$

i.e., they are defined in terms of the higher-order function g.

The fundamental theorem of calculus gives us

$$f\ x = \textstyle\int_0^x (g\ f\ t)\ dt + f_0$$

If $f = \mathit{eval}\ as$

$$eval\ as\ x = \int_0^x (g\ (eval\ as)\ t)\ dt + f_0$$

Assuming that g is a polymorphic function defined both for the syntax (*PowerSeries*) and the semantics ($\mathbb{R} \rightarrow \mathbb{R}$), and that

$$\forall\ as.\ \ eval\ (g_{syn}\ as) == g_{sem}\ (eval\ as)$$

or simply $H_1\ (eval, g, g)$. (This particular use of H_1 is read "g commutes with *eval*".) Then we can move *eval* outwards step by step:

$$\begin{aligned}
&\quad eval\ as\ x = \int_0^x (eval\ (g\ as)\ t)\ dt + f_0 \\
&\Leftrightarrow \\
&\quad eval\ as\ x = eval\ (integ\ (g\ as)\ f_0)\ x \\
&\Leftarrow \\
&\quad as = integ\ (g\ as)\ f_0
\end{aligned}$$

Finally, we have arrived at an equation expressed in only syntactic operations, which is implementable in Haskell (for reasonable g).

Which functions g commute with *eval*? All the ones in *Num*, *Fractional*, *Floating*, by construction; additionally, as above, *deriv* and *integ*.

Therefore, we can implement a general solver for these simple equations:

```
solve :: Fractional a => (PowerSeries a -> PowerSeries a) -> a -> PowerSeries a
solve g f0 = f    -- solves f' = g f, f 0 = f0
  where f = integ (g f) f0
```

To see this in action we can use *solve* on simple functions g, starting with *const* 1 and *id*:

```
idx :: Fractional a => PowerSeries a
idx = solve (\f -> 1) 0
idf :: Fractional a => a -> a
idf = eval 100 idx

expx :: Fractional a => PowerSeries a
expx = solve (\f -> f) 1
expf :: Fractional a => a -> a
expf = eval 100 expx
```

The first solution, *idx* is just the polynomial $[0, 1]$. We can easily check that its derivative is constantly 1 and its value at 0 is 0. The function *idf* is just there to check that the semantics behaves as expected.

The second solution *expx* is a formal power series representing the exponential function. It is equal to its derivative and it starts at 1. The function *expf* is a very good approximation of the semantics.

```
testExp = maximum $ map diff [0, 0.001 .. 1 :: Double]
  where diff = abs (expf - exp)   -- using the function instances for abs and exp
testExpUnits = testExp / ε
ε :: Double   -- one bit of Double precision
ε = last $ takeWhile (\x -> 1 + x ≠ 1) (iterate (/2) 1)
```

We can also use mutual recursion to define sine and cosine in terms of each other:

```
sinx = integ cosx     0
cosx = integ (-sinx) 1
```

$$\begin{aligned}
sinf &= eval\ 100\ sinx \\
cosf &= eval\ 100\ cosx
\end{aligned}$$

$$\begin{aligned}
sinx, cosx &:: Fractional\ a \Rightarrow PowerSeries\ a \\
sinf, cosf &:: Fractional\ a \Rightarrow a \to a
\end{aligned}$$

The reason why these definitions "work" (in the sense of not looping) is because *integ* immediately returns the first element of the stream before requesting any information about its first input. It is instructive to mimic part of what the lazy evaluation machinery is doing "by hand" as follows. We know that both *sinx* and *cosx* are streams, thus we can start by filling in just the very top level structure:

$$\begin{aligned}
sx &= sh : st \\
cx &= ch : ct
\end{aligned}$$

where *sh* & *ch* are the heads and *st* & *ct* are the tails of the two streams. Then we notice that *integ* fills in the constant as the head, and we can progress to:

$$\begin{aligned}
sx &= 0 : st \\
cx &= 1 : ct
\end{aligned}$$

At this stage we only know the constant term of each power series, but that is enough for the next step: the head of *st* is $\frac{1}{1}$ and the head of *ct* is $\frac{\text{-}0}{1}$:

$$\begin{aligned}
sx &= 0 : 1\ \ : _ \\
cx &= 1 : \text{-}0 : _
\end{aligned}$$

As we move on, we can always compute the next element of one series by the previous element of the other series (divided by n, for cx negated).

$$\begin{aligned}
sx &= 0 : 1\ \ : \text{-}0 : \tfrac{\text{-}1}{6} : error\ \texttt{"TODO"} \\
cx &= 1 : \text{-}0 : \tfrac{\text{-}1}{2} : 0\ \ : error\ \texttt{"TODO"}
\end{aligned}$$

6.6 The *Floating* structure of *PowerSeries*

Can we compute *exp as*?

Specification:

$$eval\ (exp\ as) = exp\ (eval\ as)$$

Differentiating both sides, we obtain

$$\begin{aligned}
& D\ (eval\ (exp\ as)) = exp\ (eval\ as) * D\ (eval\ as) \\
\Leftrightarrow & \{\text{- } eval \text{ morphism -}\} \\
& eval\ (deriv\ (exp\ as)) = eval\ (exp\ as * deriv\ as) \\
\Leftarrow & \\
& deriv\ (exp\ as) = exp\ as * deriv\ as
\end{aligned}$$

Adding the "initial condition" $eval\ (exp\ as)\ 0 = exp\ (head\ as)$, we obtain

$$exp\ as = integ\ (exp\ as * deriv\ as)\ (exp\ (head\ as))$$

Note: we cannot use *solve* here, because the g function uses both *exp as* and *as* (it "looks inside" its argument).

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Eq}\ a, \mathit{Floating}\ a) \Rightarrow \mathit{Floating}\ (\mathit{PowerSeries}\ a)\ \textbf{where} \\
\quad \pi \quad\ \ = \mathit{Single}\ \pi \\
\quad \mathit{exp} \ = \mathit{expPS} \\
\quad \mathit{sin} \ \ = \mathit{sinPS} \\
\quad \mathit{cos} \ \ = \mathit{cosPS}
\end{array}
$$

$$
\begin{array}{l}
\mathit{expPS}, \mathit{sinPS}, \mathit{cosPS} :: (\mathit{Eq}\ a, \mathit{Floating}\ a) \Rightarrow \mathit{PowerSeries}\ a \to \mathit{PowerSeries}\ a \\
\mathit{expPS}\ \ \mathit{fs}\ = \mathit{integ}\ (\mathit{exp}\ \mathit{fs}\ \ * \mathit{deriv}\ \mathit{fs})\ (\mathit{exp}\ (\mathit{val}\ \mathit{fs})) \\
\mathit{sinPS}\ \ \mathit{fs}\ = \mathit{integ}\ (\mathit{cos}\ \mathit{fs}\ \ * \mathit{deriv}\ \mathit{fs})\ (\mathit{sin}\ (\mathit{val}\ \mathit{fs})) \\
\mathit{cosPS}\ \ \mathit{fs}\ = \mathit{integ}\ (-\mathit{sin}\ \mathit{fs} * \mathit{deriv}\ \mathit{fs})\ (\mathit{cos}\ (\mathit{val}\ \mathit{fs}))
\end{array}
$$

$$
\begin{array}{l}
\mathit{val} :: \mathit{PowerSeries}\ a \to a \\
\mathit{val}\ \ (\mathit{Single}\ a) \qquad = a \\
\mathit{val}\ \ (\mathit{Cons}\ a\ \mathit{as}) \quad\ = a
\end{array}
$$

In fact, we can implement *all* the operations needed for evaluating *FunExp* functions as power series!

$$
\begin{array}{l}
\mathit{evalP} :: (\mathit{Eq}\ r, \mathit{Floating}\ r) \Rightarrow \mathit{FunExp} \to \mathit{PowerSeries}\ r \\
\mathit{evalP}\ (\mathit{Const}\ x)\ = \mathit{Single}\ (\mathit{fromRational}\ (\mathit{toRational}\ x)) \\
\mathit{evalP}\ (e_1 :+: e_2) = \mathit{evalP}\ e_1 + \mathit{evalP}\ e_2 \\
\mathit{evalP}\ (e_1 :*: e_2)\ = \mathit{evalP}\ e_1 * \mathit{evalP}\ e_2 \\
\mathit{evalP}\ (e_1 :/: e_2)\ = \mathit{evalP}\ e_1\ /\ \mathit{evalP}\ e_2 \\
\mathit{evalP}\ \mathit{Id} \qquad\quad\ = \mathit{idx} \\
\mathit{evalP}\ (\mathit{Exp}\ e) \quad = \mathit{exp}\ (\mathit{evalP}\ e) \\
\mathit{evalP}\ (\mathit{Sin}\ e) \quad\ = \mathit{sin}\ (\mathit{evalP}\ e) \\
\mathit{evalP}\ (\mathit{Cos}\ e) \quad = \mathit{cos}\ (\mathit{evalP}\ e)
\end{array}
$$

6.7 Taylor series

If $f = \mathit{eval}\ [a_0, a_1, ..., a_n, ...]$, then

$$
\begin{array}{l}
\quad f\ 0\ \ = a_0 \\
\quad f' \quad = \mathit{eval}\ (\mathit{deriv}\ [a_0, a_1, ..., a_n, ...]) \\
\qquad\quad\ = \mathit{eval}\ ([1 * a_1, 2 * a_2, 3 * a_3, ..., n * a_n, ...]) \\
\Rightarrow \\
\quad f'\ 0 = a_1 \\
\quad f'' \quad = \mathit{eval}\ (\mathit{deriv}\ [a_1, 2 * a_2, ..., n * a_n, ...]) \\
\qquad\quad\ = \mathit{eval}\ ([2 * a_2, 3 * 2 * a_3, ..., n * (n-1) * a_n, ...]) \\
\Rightarrow \\
\quad f''\ 0 = 2 * a_2
\end{array}
$$

In general:

$$f^{(k)} 0 = \mathit{fact}\ k * a_k$$

Therefore

$$f = \mathit{eval}\ [f\ 0, f'\ 0, f''\ 0\ /\ 2, ..., f^{(n)} 0\ /\ (\mathit{fact}\ n), ...]$$

The series $[f\ 0, f'\ 0, f''\ 0\ /\ 2, ..., f^{(n)} 0\ /\ (\mathit{fact}\ n), ...]$ is called the Taylor series centred in 0, or the Maclaurin series.

Therefore, if we can represent f as a power series, we can find the value of all derivatives of f at 0!

```
derivs :: Num a => PowerSeries a -> PowerSeries a
derivs as = derivs1 as 0 1
  where
  derivs1 (Cons a as) n factn = Cons  (a * factn)
                                      (derivs1 as (n + 1) (factn * (n + 1)))
  derivs1 (Single a)  n factn = Single (a * factn)
  -- remember that x = Cons 0 (Single 1)
ex3 = takePoly 10 (derivs (x^3 + 2 * x))
ex4 = takePoly 10 (derivs sinx)
```

In this way, we can compute all the derivatives at 0 for all functions f constructed with the grammar of *FunExp*. That is because, as we have seen, we can represent all of them by power series!

What if we want the value of the derivatives at $a \neq 0$?

We then need the power series of the "shifted" function g:

$$g\ x = f\ (x + a) \Leftrightarrow g = f \circ (+a)$$

If we can represent g as a power series, say $[b_0, b_1, ...]$, then we have

$$g^{(k)} 0 = \mathit{fact}\ k * b_k = f^{(k)}\, a$$

In particular, we would have

$$f\ x = g\ (x - a) = \sum b_n * (x - a)^n$$

which is called the Taylor expansion of f at a.

Example:

We have that $\mathit{idx} = [0, 1]$, thus giving us indeed the values

$$[\mathit{id}\ 0, \mathit{id}'\ 0, \mathit{id}''\ 0, ...]$$

In order to compute the values of

$$[\mathit{id}\ a, \mathit{id}'\ a, \mathit{id}''\ a, ...]$$

for $a \neq 0$, we compute

```
ida a = takePoly 10 (derivs (evalP (Id :+: Const a)))
```

More generally, if we want to compute the derivative of a function f constructed with *FunExp* grammar, at a point a, we need the power series of $g\ x = f\ (x + a)$:

```
d f a = takePoly 10 (derivs (evalP (f (Id :+: Const a))))
```

Use, for example, our $f\ x = \mathit{sin}\ x + 2 * x$ above.

As before, we can use directly power series:

```
dP f a = takePoly 10 (derivs (f (idx + Single a)))
```

6.8 Associated code

```
evalFunExp :: Floating a ⇒ FunExp → a → a
evalFunExp (Const α) = const (fromRational (toRational α))
evalFunExp Id        = id
evalFunExp (e₁ :+: e₂) = evalFunExp e₁ + evalFunExp e₂   -- note the use of “lifted +”
evalFunExp (e₁ :*: e₂) = evalFunExp e₁ * evalFunExp e₂   -- “lifted *”
evalFunExp (Exp e₁)    = exp (evalFunExp e₁)             -- and “lifted exp”
evalFunExp (Sin e₁)    = sin (evalFunExp e₁)
evalFunExp (Cos e₁)    = cos (evalFunExp e₁)
  -- and so on
derive (Const α) = Const 0
derive Id        = Const 1
derive (e₁ :+: e₂) = derive e₁ :+: derive e₂
derive (e₁ :*: e₂) = (derive e₁ :*:        e₂) :+: (e₁ :*:      derive e₂)
derive (Exp e)   = Exp e :*: derive e
derive (Sin e)   = Cos e :*: derive e
derive (Cos e)   = Const (−1) :*: Sin e :*: derive e
instance Num FunExp where
  (+) = (:+:)
  (*) = (:*:)
  fromInteger n = Const (fromInteger n)
instance Fractional FunExp where
  (/) = (:/:)
  fromRational = Const ∘ fromRational
instance Floating FunExp where
  exp =       Exp
  sin =       Sin
  cos =       Cos
```

6.8.1 Not included to avoid overlapping instances

```
instance Num a ⇒ Num (FD a) where
  (f, f′) + (g, g′) = (f + g, f′ + g′)
  (f, f′) * (g, g′) = (f * g, f′ * g + f * g′)
  fromInteger n = (fromInteger n, const 0)
instance Fractional a ⇒ Fractional (FD a) where
  (f, f′) / (g, g′) = (f / g, (f′ * g − g′ * f) / (g * g))
instance Floating a ⇒ Floating (FD a) where
  exp (f, f′)   = (exp f, (exp f) * f′)
  sin (f, f′)   = (sin f, (cos f) * f′)
  cos (f, f′)   = (cos f, −(sin f) * f′)
```

6.8.2 This is included instead

```
instance Num a ⇒ Num (a, a) where
  (f, f′) + (g, g′) = (f + g, f′ + g′)
  (f, f′) * (g, g′) = (f * g, f′ * g + f * g′)
  fromInteger n = (fromInteger n, fromInteger 0)
```

$$
\begin{array}{l}
\textbf{instance } \mathit{Fractional}\ a \Rightarrow \mathit{Fractional}\ (a, a) \textbf{ where} \\
\quad (f, f') \mathbin{/} (g, g') = (f \mathbin{/} g, (f' * g - g' * f) \mathbin{/} (g * g)) \\
\textbf{instance } \mathit{Floating}\ a \Rightarrow \mathit{Floating}\ (a, a) \textbf{ where} \\
\quad \mathit{exp}\ (f, f') \quad = (\mathit{exp}\ f, (\mathit{exp}\ f) * f') \\
\quad \mathit{sin}\ (f, f') \quad = (\mathit{sin}\ f, \mathit{cos}\ f * f') \\
\quad \mathit{cos}\ (f, f') \quad = (\mathit{cos}\ f, -(\mathit{sin}\ f) * f')
\end{array}
$$

6.9 Exercises

Exercise 6.1. As shown at the start of the chapter, we can find expressions $e :: \mathit{FunExp}$ such that $\mathit{eval}\ e = f$ automatically using the assignment $e = f\ \mathit{Id}$. This is possible thanks to the *Num*, *Fractional*, and *Floating* instances of *FunExp*. Use this method to find *FunExp* representations of the functions below, and show step by step how the application of the function to *Id* is evaluated in each case.

a. $f_1\ x = x\hat{\ }2 + 4$

b. $f_2\ x = 7 * \mathit{exp}\ (2 + 3 * x)$

c. $f_3\ x = 1 \mathbin{/} (\mathit{sin}\ x + \mathit{cos}\ x)$

Exercise 6.2. For each of the expressions $e :: \mathit{FunExp}$ you found in exercise 6.1, use *derive* to find an expression $e' :: \mathit{FunExp}$ representing the derivative of the expression, and verify that e' is indeed the derivative of e.

Exercise 6.3. At the start of this chapter, we saw three different ways of computing the value of the derivative of a function at a given point:

a. Using *FunExp*

b. Using *FD*

c. Using pairs

Try using each of these methods to find the values of $f_1'\ 2$, $f_2'\ 2$, and $f_3'\ 2$, i.e. the derivatives of each of the functions in exercise 6.1, evaluated at the point 2. You can verify that the result is correct by comparing it with the expressions e_1', e_2' and e_3' that you found in 6.2.

Exercise 6.4. The exponential function $\mathit{exp}\ t = e\hat{\ }t$ has the property that $\int \mathit{exp}\ t\ dt = \mathit{exp}\ t + C$. Use this fact to express the functions below as *PowerSeries* using *integ*. *Hint: the definitions will be recursive.*

a. $\lambda t \rightarrow \mathit{exp}\ t$

b. $\lambda t \rightarrow \mathit{exp}\ (3 * t)$

c. $\lambda t \rightarrow 3 * \mathit{exp}\ (2 * t)$

Exercise 6.5. In the chapter, we saw that a representation $\mathit{expx} :: \mathit{PowerSeries}$ of the exponential function can be implemented using *solve* as $\mathit{expx} = \mathit{solve}\ (\lambda f \rightarrow f)\ 1$. Use the same method to implement power series representations of the following functions:

a. $\lambda t \rightarrow \mathit{exp}\ (3 * t)$

b. $\lambda t \to 3 * exp\ (2 * t)$

Exercise 6.6.

a. Implement idx', $sinx'$ and $cosx'$ using $solve$

b. Complete the instance $Floating\ (PowerSeries\ a)$

Exercise 6.7. Consider the following differential equation:

$$f''\,t + f'\,t - 2 * f\,t = e^{3*t}, \quad f\,0 = 1, \quad f'\,0 = 2$$

We will solve this equation assuming that f can be expressed by a power series fs, and finding the three first coefficients of fs.

a. Implement $expx3 :: PowerSeries$, a power series representation of e^{3*t}

b. Find an expression for fs'', the second derivative of fs, in terms of $expx3$, fs', and fs.

c. Find an expression for fs' in terms of fs'', using $integ$.

d. Find an expression for fs in terms of fs', using $integ$.

e. Use $takePoly$ to find the first three coefficients of fs. You can check that your solution is correct using a tool such as MATLAB or WolframAlpha, by first finding an expression for $f\ t$, and then getting the Taylor series expansion for that expression.

Exercise 6.8. *From exam 2016-03-15*

Consider the following differential equation:

$$f''\,t - 2 * f'\,t + f\,t = e^{2*t}, \quad f\,0 = 2, \quad f'\,0 = 3$$

Solve the equation assuming that f can be expressed by a power series fs, that is, use $deriv$ and $integ$ to compute fs. What are the first three coefficients of fs?

Exercise 6.9. *From exam 2016-08-23*

Consider the following differential equation:

$$f''\,t - 5 * f'\,t + 6 * f\,t = e^{t}, \quad f\,0 = 1, \quad f'\,0 = 4$$

Solve the equation assuming that f can be expressed by a power series fs, that is, use $deriv$ and $integ$ to compute fs. What are the first three coefficients of fs?

Exercise 6.10. *From exam 2016-Practice*

Consider the following differential equation:

$$f''\,t - 2 * f'\,t + f\,t - 2 = 3 * e^{2*t}, \quad f\,0 = 5, \quad f'\,0 = 6$$

Solve the equation assuming that f can be expressed by a power series fs, that is, use $deriv$ and $integ$ to compute fs. What are the first three coefficients of fs?

Exercise 6.11. *From exam 2017-03-14*

Consider the following differential equation:

$$f''\,t + 4 * f\,t = 6 * \cos\,t, \quad f\,0 = 0, \quad f'\,0 = 0$$

Solve the equation assuming that f can be expressed by a power series *fs*, that is, use *integ* and the differential equation to express the relation between *fs*, *fs'*, *fs''*, and *rhs* where *rhs* is the power series representation of $(6*) \circ \mathit{cos}$. What are the first four coefficients of *fs*?

Exercise 6.12. *From exam 2017-08-22*

Consider the following differential equation:

$$f''\,t - 3\sqrt{2} * f'\,t + 4 * f\,t = 0, \quad f\,0 = 2, \quad f'\,0 = 3\sqrt{2}$$

Solve the equation assuming that f can be expressed by a power series *fs*, that is, use *integ* and the differential equation to express the relation between *fs*, *fs'*, and *fs''*. What are the first three coefficients of *fs*?