```haskell
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE TypeSynonymInstances #-}
```

Review

- key notion *homomorphism*: `S1 -> S2`
- questions ("equations"):
  - `S1 ->? S2` what is the homomorphism between two given structures
    - e.g., `apply: Num a -> Num (x -> a)`
  - `S1? -> S2` what is `S1` compatible with a given homomorphism
    - e.g., `eval : Poly a -> (a -> a)`
  - `S1 -> S2?` what is `S2` compatible with a given homomorphism
    - e.g., `applyFD : FD a -> (a, a)`
  - `S1 ->? S2?` can we find a good structure on `S2` so that it becomes homomorphic w. `S1`?
    - e.g., `evalD : FunExp -> 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

```haskell
f x = sin x + 2 * x
```

We have: `f 0 = 1`, `f 2 = 4.909297426825682`s, etc.

The type of `f` is `f :: Floating a => a -> a`.

How do we compute, say, `f' 2`?

We have several choices.

1. Using `FunExp`

Recall (week 3):

```haskell
data FunExp  =  Const Double
             |  Id
             |  FunExp :+: FunExp
             |  FunExp :*: FunExp
             |  FunExp :/: FunExp
             |  Exp FunExp
             |  Sin FunExp
```

```
             |  Cos FunExp
             -- and so on
             deriving Show
```

What is the expression `e` for which `f = eval e`?

We have

```
   eval e x = f x

<=>

   eval e x = sin x + 2 * x

<=>

   eval e x = eval (Sin Id) x + eval (Const 2 :*: Id) x

<=>

   eval e x = eval ((Sin Id) :+: (Const 2 :*: Id)) x

<=

e = Sin Id :+: (Const 2 :*: Id)
```

Finally, we can apply `derive` and obtain

```
f' 2 = eval (derive e) 2
```

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

However, consider this:

```
 e = f Id
```

(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
```

2. Using `FD`

Recall

```
type FD a = (a -> a, a -> a)
```

```
applyFD (f, g) x = (f x, g x)
```

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')    (*)
```

so we can then do

```
f' 2 = snd (applyFD (f (g, g')) 2)
```

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 . g, (f' . g) * g')
```

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

Finally

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

In general

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

computes the derivative of `f` at `x`.

```
f1 :: FD Double -> FD Double
f1  = f
```

3. Using pairs

We have `instance Floating a => Floating (a, a)`, moreover, the instance declaration looks exactly the same as that for `FD a`:

```
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')

instance Floating a => Floating (a, 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')
```

In fact, the latter represents a generalisation of the former. It 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

```
  f (g, g') = (f 2, f' 2)

<=>

  (f g, (f' g) * g') = (f 2, f' 2)

<=

  g = 2, g' = 1
```

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`.

```
f2 :: (Double, Double) -> (Double, Double)
f2  = f
```

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 (e1 :*: e2) = evalAll e1 * evalAll e2
```

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

```
    evalAll (e1 :*: e2) = evalAll e1 * evalAll e2
<=>
    eval (e1 :*: e2) : evalAll (derive (e1 :*: e2)) =
    eval e1 : evalAll (derive e) * eval e1 : evalAll (derive e2)
<=>
    (eval e1 * eval e2) : evalAll (derive (e1 :*: e2)) =
    eval e1 : evalAll (derive e) * eval e1 : evalAll (derive e2)
<=>
    (eval e1 * eval e2) : evalAll (derive e1 :*: e2 :+: e1 * derive e2) =
    eval e1 : evalAll (derive e) * eval e1 : evalAll (derive e2)
<=
    (a : as) * (b : bs) = (a * b) : (as * (b : bs) + (a : as) * bs)
```

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`:

```
instance Num a => Num [a] where
  (a : as) + (b : bs) = (a + b) : (as + bs)
  (a : as) * (b : bs) = (a * b) : (as * (b : bs) + (a : as) * bs)
```

Exercise: complete the instance declarations for `Fractional` and `Floating`. Write a general derivative computation, similar to `drv` functions above:

```
drvList k f x = undefined -- kth derivative of f at x
```

This is a very inefficient way of computing derivatives!

Polynomials

```
data Poly a  =  Single a  |  Cons a (Poly a)
                deriving (Eq, Ord)
```

```
evalPoly ::  Num a => Poly a -> a -> a
evalPoly (Single a)     x   =  a
evalPoly (Cons a as)    x   =  a + x * evalPoly as x
```

Power series

No need for a separate type in Haskell

```
type PowerSeries a = Poly a -- finite and infinite non-empty lists
```

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

We can also derive:

```
deriv (Single a)    =  Single 0
deriv (Cons a as)   =  deriv' as 1
                       where deriv' (Single a)  n  =  Single (n * a)
                             deriv' (Cons a as) n  =  Cons (n * a) (deriv' as (n+1))
```

and integrate:

```
integ  ::  Fractional a => PowerSeries a -> a -> PowerSeries a
integ  as a0  =  Cons a0 (integ' as 1)
  where integ' (Single a) n   =  Single (a / n)
        integ' (Cons a as) n  =  Cons (a / n) (integ' as (n+1))
```

Everything here makes sense, irrespective of convergence, hence "formal".

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

```
eval as + eval bs = eval (as + bs)
eval as * eval bs = eval (as * bs)

eval (derive as)   =  (eval as)'
eval (integ as c) x   =  S_0^x (eval as t) dt  +  c  -- S stands for "snakey integral sign"
```

Simple differential equations

Many first-order differential equations have the structure

```
f' x = g f x,  f 0 = f0
```

i.e., they are defined in terms of `f`.

The fundamental theorem of calculus gives us

```
f x = S_0^x (g f t) dt + f0
```

If `f = eval as`

```
eval as x = S_0^x (g (eval as) t) dt + f0
```

Assuming that `g` is a polymorphic function that commutes with `eval`

```
eval as x = S_0^x (eval (g as) t) dt + f0
```

```
eval as x = eval (integ (g as) f0) x
```

```
as = integ (g as) f0
```

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' x = g f, f 0 = f0
  where f = integ (g f) f0
```

```
idx  = solve (\ f -> 1) 0
idf  = eval 100 idx
```

```
expx = solve (\ f -> f) 1
expf = eval 100 expx
```

```
sinx = solve (\ f -> cosx) 0
cosx = solve (\ f -> -sinx) 1
sinf = eval 100 sinx
cosf = eval 100 cosx
```

The `Floating` structure of `PowerSeries`

Can we compute `exp as`?

Specification:

```
eval (exp as) = exp (eval as)
```

Differentiating both sides, we obtain

```
(eval (exp as))' = exp (eval as) * (eval as)'

=>  { `eval` morphism }

eval (deriv (exp as)) = eval (exp as * deriv as)

=

deriv (exp as) = exp as * deriv as
```

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

```
exp as = integ (exp as * deriv as) (exp (val as))
```

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

```
instance (Eq a, Floating a) => Floating (PowerSeries a) where
  pi       =  Single pi
  exp fs   =  integ (exp fs * deriv fs)  (exp (val fs))
  sin fs   =  integ (cos fs * deriv fs)  (sin (val fs))
  cos fs   =  integ (-sin fs * deriv fs) (cos (val fs))
```

```
val              ::  PowerSeries a -> a
val (Single a)    =  a
val (Cons a as)   =  a
```

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

```
evalP :: FunExp -> PowerSeries Double
evalP (Const x) = Single x
evalP (e1 :+: e2)  =  evalP e1 + evalP e2
evalP (e1 :*: e2)  =  evalP e1 * evalP e2
evalP (e1 :/: e2)  =  evalP e1 / evalP e2
evalP Id           =  idx
evalP (Exp e)      =  exp (evalP e)
evalP (Sin e)      =  sin (evalP e)
evalP (Cos e)      =  cos (evalP e)
```

Taylor series

If `f = eval [a0, a1, ..., an, ...]`, then

```
f 0    =  a0
f'     =  eval (deriv [a0, a1, ..., an, ...])
       =  eval ([a1, 2 * a2, 3 * a3, ..., n * an, ...])
=>
f' 0   =  a1
f''    =  eval (deriv [a1, 2 * a2, ..., n * an, ...])
       =  eval ([2 * a2, 3 * 2 * a3, ..., n * (n - 1) * an, ...])
=>
f'' 0  =  2 * a2
```

In general:

```
f^(k) 0  =  fact k * ak
```

Therefore

```
f      =  eval [f 0, f' 0, f'' 0 / 2, ..., f^(n) 0 / (fact n), ...]
```

The series `[f 0, f' 0, f'' 0 / 2, ..., f^(n) 0 / (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     -- series n n!
  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)
```

```
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 /= 0`?

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

```
 g x  =  f (x + a)  <=>  g = f . (+ a)
```

If we can represent g as a power series, say `[b0, b1, ...]`, then we have

```
g^(k) 0  =  fact k * bk  =  f^(k) a
```

In particular, we would have

```
f x  =  g (x - a)  =  Sum bn * (x - a)^n
```

which is called the Taylor expansion of `f` at `a`.

Example:

We have that `idx = [0, 1]`, thus giving us indeed the values

```
[id 0, id' 0, id'' 0, ...]
```

In order to compute the values of

```
[id a, id' a, id'' a, ...]
```

for `a /= 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 = sin x + 2 * x` above.

As before, we can use directly power series:

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

```
instance Num a => Num (x -> a) where
  f + g        =  \x -> f x + g x
  f - g        =  \x -> f x - g x
```

```haskell
  f * g        =  \x -> f x * g x
  negate f     =  negate . f
  abs f        =  abs . f
  signum f     =  signum . f
  fromInteger  =  const . fromInteger

instance Fractional a => Fractional (x -> a) where
  recip  f        =  recip . f
  fromRational    =  const . fromRational

instance Floating a => Floating (x -> a) where
  pi       =  const pi
  exp f    =  exp . f
  sin f    =  sin . f
  cos f    =  cos . f
  f ** g   =  \ x -> (f x)**(g x)
  -- and so on
```

```haskell
evalFunExp  ::  FunExp          ->  Double -> Double
evalFunExp      (Const alpha)   =   const alpha
evalFunExp      Id              =   id
evalFunExp      (e1 :+: e2)     =   evalFunExp e1  +  evalFunExp e2    -- note the use of ``
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 alpha)  =  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
  (/)  =  (:/:)

instance Floating FunExp where
  exp       =  Exp
  sin       =  Sin
```

```haskell
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')
```

```haskell
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)

instance Fractional a => Fractional (a, a) where
  (f, f') / (g, g') = (f / g, (f' * g - g' * f) / (g * g))

instance Floating a => Floating (a, 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')
```

```haskell
toList :: Poly a -> [a]
toList (Single a)    =  [a]
toList (Cons a as)   =  a : toList as

fromList :: [a] -> Poly a
fromList [a] = Single a
fromList (a0 : a1 : as) = Cons a0 (fromList (a1 : as))

instance Show a => Show (Poly a) where
  show = show . toList

instance Num a => Num (Poly a) where
  Single a   +  Single b    =  Single (a + b)
  Single a   +  Cons b bs   =  Cons (a + b) bs
  Cons a as  +  Single b    =  Cons (a + b) as
  Cons a as  +  Cons b bs   =  Cons (a + b) (as + bs)

  Single a   *  Single b    =  Single (a * b)
  Single a   *  Cons b bs   =  Cons (a * b) (Single a * bs)
  Cons a as  *  Single b    =  Cons (a * b) (as * Single b)
```

```
  Cons a as  *  Cons b bs  =  Cons (a * b) (as * Cons b bs + Single a * bs)

  negate (Single a)        =  Single (negate a)
  negate (Cons a as)       =  Cons (negate a) (negate as)

  fromInteger              =  Single . fromInteger
```

```
eval n as x = evalPoly (takePoly n as) x

takePoly :: Integer -> Poly a -> Poly a
takePoly n (Single a)   =  Single a
takePoly n (Cons a as)  =  if n <= 1
                              then  Single a
                              else  Cons a (takePoly (n-1) as)
```

```
instance (Eq a, Fractional a) => Fractional (PowerSeries a) where
  as / Single b            =  as * Single (1 / b)
  Single a / Cons b bs     =  if a == 0 then Single 0 else Cons a (Single 0) / Cons b bs
  Cons a as / Cons b bs    =  let  q = a / b
                              in   Cons q  ((as - Single q * bs) / Cons b bs)
  fromRational             =  Single . fromRational
```