5 Polynomials and Power Series

```
{-# LANGUAGE TypeSynonymInstances #-}
module DSLsofMath.W05 where
import DSLsofMath.FunNumInst
```

5.1 Polynomials

From Adams and Essex [2010], page 55:

> A **polynomial** is a function P whose value at x is
>
> $$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$
>
> where a_n, a_{n-1}, …, a_1, and a_0, called the **coefficients** of the polymonial [misspelled in the book], are constants and, if $n > 0$, then $a_n \neq 0$. The number n, the degree of the highest power of x in the polynomial, is called the **degree** of the polynomial. (The degree of the zero polynomial is not defined.)

This definition raises a number of questions, for example "what is the zero polynomial?".

The types of the elements involved in the definition appear to be

$P : \mathbb{R} \to \mathbb{R}$, $x \in \mathbb{R}$, a_0, …, $a_n \in \mathbb{R}$ with $a_n \neq 0$ if $n > 0$

The phrasing should be "whose value at *any* x is". The remark that the a_i are constants is probably meant to indicate that they do not depend on x, otherwise every function would be a polynomial. The zero polynomial is, according to this definition, the *const* 0 function. Thus, what is meant is

> A **polynomial** is a function $P : \mathbb{R} \to \mathbb{R}$ which is either constant zero, or there exist a_0, …, $a_n \in \mathbb{R}$ with $a_n \neq 0$ such that, for any $x \in \mathbb{R}$
>
> $$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

Given the coefficients a_i we can evaluate P at any given x. Assuming the coefficients are given as

$as = [a_0, a_1, ..., a_n]$

(we prefer counting up), then the evaluation function is written

```
evalL :: [ℝ] →  ℝ → ℝ
evalL   []       x  = 0
evalL   (a : as) x  = a + x * evalL as x
```

Note that we can read the type as $evalL :: [\mathbb{R}] \to (\mathbb{R} \to \mathbb{R})$ and thus identify $[\mathbb{R}]$ as the type for the (abstract) syntax (for polynomials) and $(\mathbb{R} \to \mathbb{R})$ as the type of the semantics (for polynomial functions). Exercise: Show that this evaluation function gives the same result as the formula above.

Using the *Num* instance for functions we can rewrite *eval* into a one-argument function (returning a polynomial function):

```
evalL :: Num a ⇒ [a] → (a → a)
evalL []       = const 0
evalL (a : as) = const a + id ∗ evalL as
```

As an example, the polynomial which is usually written just x is represented by the list $[0, 1]$ and the polynomial function $\lambda x \to x\hat{}2 - 1$ is represented by the list $[-1, 0, 1]$.

It is worth noting that the definition of what we call a "polynomial function" is semantic, not syntactic. A syntactic defintion would talk about the form of the expression (a sum of coefficients times natural powers of x). This semantic definition only requires that the function P *behaves like* such a sum. (Has the same value for all x.) This may seem pedantic, but here is an interesting example of a family of functions which syntactically looks very trigonometric:

$$T_n(x) = \cos(n * \arccos(x)) \ .$$

It can be shown that T_n is a polynomial function of degree n. (Exercise 5.4 guides you to a proof. At this point you could just compute T_0, T_1, and T_2 by hand to get a feeling for how it works.)

Not every list of coefficients is valid according to the definition. In particular, the empty list is not a valid list of coefficients, so we have a conceptual, if not empirical, type error in our evaluator.

The valid lists are those *finite* lists in the set

$$\{[0]\} \cup \{(a : as) \mid last\ (a : as) \neq 0\}$$

We cannot express the $last\ (a : as) \neq 0$ in Haskell, but we can express the condition that the list should not be empty:

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

Note that if we drop the requirement of what constitutes a "valid" list of coefficients we can use $[a]$ instead of *Poly* a. Basically, we then use $[\,]$ as the syntax for the "zero polynomial" and $(c : cs)$ for all non-zero polynomials.

The relationship between *Poly* a and $[a]$ is given by the following functions:

```
toList :: Poly a →    [a]
toList (Single a)   = a : []
toList (Cons a as)  = a : toList as

fromList :: Num a ⇒ [a] → Poly a
fromList (a : [])          = Single a
fromList (a0 : a1 : as)    = Cons a0 (fromList (a1 : as))
fromList []                = Single 0   -- to complete the pattern match

instance Show a ⇒ Show (Poly a) where
  show = show ∘ toList
```

Since we only use the arithmetical operations, we can generalise our evaluator:

```
evalPoly :: Num a ⇒ Poly a → (a → a)
evalPoly (Single a)   x = a
evalPoly (Cons a as)  x = a + x ∗ evalPoly as x
```

Since we have *Num* a, there is a *Num* structure on $a \to a$, and *evalPoly* looks like a homomorphism. Question: is there a *Num* structure on *Poly* a, such that *evalPoly* is a homomorphism?

For example, the homomorphism condition gives for $(+)$

$$\mathit{evalPoly}\ \mathit{as} + \mathit{evalPoly}\ \mathit{bs} = \mathit{evalPoly}\ (\mathit{as} + \mathit{bs})$$

Both sides are functions, they are equal iff they are equal for every argument. For an arbitrary x

$$\begin{aligned}
&(\mathit{evalPoly}\ \mathit{as} + \mathit{evalPoly}\ \mathit{bs})\ x = \mathit{evalPoly}\ (\mathit{as} + \mathit{bs})\ x\\
\Leftrightarrow\ &\{\text{- + on functions is defined point-wise -}\}\\
&\mathit{evalPoly}\ \mathit{as}\ x + \mathit{evalPoly}\ \mathit{bs}\ x = \mathit{evalPoly}\ (\mathit{as} + \mathit{bs})\ x
\end{aligned}$$

To proceed further, we need to consider the various cases in the definition of *evalPoly*. We give here the computation for the last case (where *as* has at least one *Cons*), using the traditional list notation (:) for brevity.

$$\mathit{evalPoly}\ (a : \mathit{as})\ x + \mathit{evalPoly}\ (b : \mathit{bs})\ x = \mathit{evalPoly}\ ((a : \mathit{as}) + (b : \mathit{bs}))\ x$$

For the left-hand side, we have:

$$\begin{aligned}
&\mathit{evalPoly}\ (a : \mathit{as})\ x + \mathit{evalPoly}\ (b : \mathit{bs})\ x &&= \{\text{- def. } \mathit{evalPoly} \text{ -}\}\\
&(a + x * \mathit{evalPoly}\ \mathit{as}\ x) + (b + x * \mathit{eval}\ \mathit{bs}\ x) &&= \{\text{- properties of +, valid in any ring -}\}\\
&(a + b) + x * (\mathit{evalPoly}\ \mathit{as}\ x + \mathit{evalPoly}\ \mathit{bs}\ x) &&= \{\text{- homomorphism condition -}\}\\
&(a + b) + x * (\mathit{evalPoly}\ (\mathit{as} + \mathit{bs})\ x) &&= \{\text{- def. } \mathit{evalPoly} \text{ -}\}\\
&\mathit{evalPoly}\ ((a + b) : (\mathit{as} + \mathit{bs}))\ x
\end{aligned}$$

The homomorphism condition will hold for every x if we define

$$(a : \mathit{as}) + (b : \mathit{bs}) = (a + b) : (\mathit{as} + \mathit{bs})$$

This is definition looks natural (we could probably have guessed it early on) but it is still interesting to see that we can derive the definition as the the form it has to take for the proof to go through.

We leave the derivation of the other cases and operations as an exercise. Here, we just give the corresponding definitions.

```
instance Num a ⇒ Num (Poly a) where
  (+) = polyAdd
  (∗) = polyMul
  negate = polyNeg
  fromInteger = Single ∘ fromInteger
polyAdd :: Num a ⇒ Poly a → Poly a → Poly a
polyAdd (Single a)  (Single b)  = Single (a + b)
polyAdd (Single a)  (Cons b bs) = Cons (a + b) bs
polyAdd (Cons a as) (Single b)  = Cons (a + b) as
polyAdd (Cons a as) (Cons b bs) = Cons (a + b) (polyAdd as bs)
polyMul :: Num a ⇒ Poly a → Poly a → Poly a
polyMul (Single a)  (Single b)  = Single (a ∗ b)
polyMul (Single a)  (Cons b bs) = Cons (a ∗ b) (polyMul (Single a) bs)
polyMul (Cons a as) (Single b)  = Cons (a ∗ b) (polyMul as (Single b))
polyMul (Cons a as) (Cons b bs) = Cons (a ∗ b) (polyAdd (polyMul as (Cons b bs))
                                                         (polyMul (Single a) bs))
polyNeg :: Num a ⇒ Poly a → Poly a
polyNeg = mapPoly negate
```

$$
\begin{aligned}
&\mathit{mapPoly} :: (a \to b) \to (\mathit{Poly}\ a \to \mathit{Poly}\ b)\\
&\mathit{mapPoly}\ f\ (\mathit{Single}\ a) \quad = \mathit{Single}\ (f\ a)\\
&\mathit{mapPoly}\ f\ (\mathit{Cons}\ a\ as) = \mathit{Cons}\ (f\ a)\ (\mathit{mapPoly}\ f\ as)
\end{aligned}
$$

Therefore, we *can* define a ring structure (the mathematical counterpart of *Num*) on *Poly a*, and we have arrived at the canonical definition of polynomials, as found in any algebra book (see, for example, Rotman [2006] for a very readable text):

> Given a commutative ring A, the commutative ring given by the set *Poly A* together with the operations defined above is the ring of **polynomials** with coefficients in A.

The functions *evalPoly as* are known as *polynomial functions*.

Caveat: The canonical representation of polynomials in algebra does not use finite lists, but the equivalent

$$\mathit{Poly}'\ A = \{\, a : \mathbb{N} \to A \mid \text{\{- } a \text{ has only a finite number of non-zero values -\}} \,\}$$

Exercise: what are the ring operations on *Poly′ A*? For example, here is addition:

$$a + b = c \Leftrightarrow a\ n + b\ n = c\ n \quad \text{-- } \forall n : \mathbb{N}$$

Observations:

a. Polynomials are not, in general, isomorphic (in one-to-one correspondence) with polynomial functions. For any finite ring A, there is a finite number of functions $A \to A$, but there is a countable number of polynomials. That means that the same polynomial function on A will be the evaluation of many different polynomials.

   For example, consider the ring $\mathbb{Z}_2$ ($\{0, 1\}$ with addition and multiplication modulo 2). In this ring, we have that $p\ x = x + x\hat{\ }2$ is actually a constant function. The only two input values to p are 0 and 1 and we can easily check that $p\ 0 = 0$ and also $p\ 1 = (1 + 1\hat{\ }2)\%2 = 2\%2 = 0$. Thus

   $$\mathit{evalPoly}\ [0,1,1] = p = \mathit{const}\ 0 = \mathit{evalPoly}\ [0] \text{\{- in } \mathbb{Z}_2 \to \mathbb{Z}_2 \text{ -\}}$$

   but

   $$[0,1,1] \neq [0] \text{\{- in } \mathit{Poly}\ \mathbb{Z}_2 \text{ -\}}$$

   Therefore, it is not generally a good idea to confuse polynomials with polynomial functions.

b. In keeping with the DSL terminology, we can say that the polynomial functions are the semantics of the language of polynomials. We started with polynomial functions, we wrote the evaluation function and realised that we have the makings of a homomorphism. That suggested that we could create an adequate language for polynomial functions. Indeed, this turns out to be the case; in so doing, we have recreated an important mathematical achievement: the algebraic definition of polynomials.

   Let

   $$
   \begin{aligned}
   &x :: \mathit{Num}\ a \Rightarrow \mathit{Poly}\ a\\
   &x = \mathit{Cons}\ 0\ (\mathit{Single}\ 1)
   \end{aligned}
   $$

Then (again, using the list notation for brevity) for any polynomial $as = [a_0, a_1, ..., a_n]$ we have

$$as = a_0 + a_1 * x + a_2 * x\hat{}2 + ... + a_n * x\hat{}n$$

Exercise: check this.

This justifies the standard notation

$$as = \sum_{i=0}^{n} a_i * x^i$$

5.2 Polynomial degree as a homomorphism

It is often the case that a certain function is *almost* a homomorphism and the domain or range *almost* a monoid. In the section on *eval* and *eval′* for *FunExp* we have seen "tupling" as one way to fix such a problem and here we will introduce another way.

The *degree* of a polynomial is a good candidate for being a homomorphism: if we multiply two polynomials we can normally add their degrees. If we try to check that $degree :: Poly\ a \to \mathbb{N}$ is the function underlying a monoid morphism we need to decide on the monoid structure to use for the source and for the target, and we need to check the homomorphism laws. We can use $unit = Single\ 1$ and $op = polyMul$ for the source monoid and we can try to use $unit = 0$ and $op = (+)$ for the target monoid. Then we need to check that

$$degree\ (Single\ 1) = 0$$
$$\forall\ x, y.\ degree\ (x\ `op`\ y) = degree\ x + degree\ y$$

The first law is no problem and for most polynomials the second law is also straighforward to prove (exercise: prove it). But we run into trouble with one special case: the zero polynomial.

Looking back at the definition from Adams and Essex [2010], page 55 it says that the degree of the zero polynomial is not defined. Let's see why that is the case and how we might "fix" it. Assume there is a z such that $degree\ 0 = z$ and that we have some polynomial p with $degree\ p = n$. Then we get

$$\begin{aligned}
&z &&= \{\text{- assumption -}\}\\
&degree\ 0 &&= \{\text{- simple calculation -}\}\\
&degree\ (0 * p) &&= \{\text{- homomorphism condition -}\}\\
&degree\ 0 + degree\ p &&= \{\text{- assumption -}\}\\
&z + n
\end{aligned}$$

Thus we need to find a z such that $z = z + n$ for all natural numbers n! At this stage we could either give up, or think out of the box. Intuitively we could try to use $z = -Infinity$, which would seem to satisfy the law but which is not a natural number (not even an integer). More formally what we need to do is to extend the monoid $(\mathbb{N}, 0, +)$ by one more element. In Haskell we can do that using the *Maybe* type constructor:

```
class Monoid a where
  unit :: a
  op   :: a → a → a
instance Monoid a ⇒ Monoid (Maybe a) where
  unit = Just unit
  op   = opMaybe
```

```
opMaybe Nothing  m           = Nothing  -- −Inf + m = −Inf
opMaybe m        Nothing     = Nothing  -- m + (−Inf) = −Inf
opMaybe (Just m1) (Just m2)  = Just (op m1 m2)
```

Thus, to sum up, *degree* is a monoid homomorphism from $(Poly\ a, 1, *)$ to $(Maybe\ \mathbb{N}, Just\ 0, opMaybe)$.

Exercise: check all the Monoid and homomorphism properties.

5.3 Power Series

Power series are obtained from polynomials by removing in $Poly'$ the restriction that there should be a *finite* number of non-zero coefficients; or, in, the case of *Poly*, by going from lists to streams.

$$PowerSeries'\ a = \{f : \mathbb{N} \to a\}$$

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

The operations are still defined as before. If we consider only infinite lists, then only the equations which do not contain the patterns for singleton lists will apply.

Power series are usually denoted

$$\sum_{n=0}^{\infty} a_n * x^n$$

the interpretation of x being the same as before. The simplest operation, addition, can be illustrated as follows:

$$\begin{array}{lcllll}
\sum_{i=0}^{\infty} a_i * x^i & \cong & [a_0, & a_1, & \ldots] \\
\sum_{i=0}^{\infty} b_i * x^i & \cong & [b_0, & b_1, & \ldots] \\
\sum_{i=0}^{\infty} (a_i + b_i) * x^i & \cong & [a_0 + b_0, & a_1 + b_1, & \ldots]
\end{array}$$

The evaluation of a power series represented by $a{:}\mathbb{N}{\to}A$ is defined, in case the necessary operations make sense on A, as a function

$$eval\ a : A \to A$$
$$eval\ a\ x = lim\ s\ \textbf{where}\ s\ n = \sum_{i=0}^{n} a_i * x^i$$

Note that $eval\ a$ is, in general, a partial function (the limit might not exist).

We will consider, as is usual, only the case in which $A = \mathbb{R}$ or $A = \mathbb{C}$.

The term *formal* refers to the independence of the definition of power series from the ideas of convergence and evaluation. In particular, two power series represented by a and b, respectively, are equal only if $a = b$ (as functions). If $a \neq b$, then the power series are different, even if $eval\ a = eval\ b$.

Since we cannot in general compute limits, we can use an "approximative" *eval*, by evaluating the polynomial resulting from an initial segment of the power series.

```
eval :: Num a ⇒ Integer → PowerSeries a → (a → a)
eval n as x = evalPoly (takePoly n as) x
```

```
takePoly :: Integer → PowerSeries 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)
```

Note that *eval n* is not a homomorphism: for example:

```
eval 2 (x ∗ x) 1                    =
evalPoly (takePoly 2 [0, 0, 1]) 1 =
evalPoly [0, 0] 1                   =
0
```

but

```
(eval 2 x 1)                     =
evalPoly (takePoly 2 [0, 1]) 1 =
evalPoly [0, 1] 1                =
1
```

and thus $eval\ 2\ (x * x)\ 1 = 0 \neq 1 = 1 * 1 = (eval\ 2\ x\ 1) * (eval\ 2\ x\ 1)$.

5.4 Operations on power series

Power series have a richer structure than polynomials. For example, we also have division (this is similar to the move from $\mathbb{Z}$ to $\mathbb{Q}$). We start with a special case: trying to compute $p = \frac{1}{1-x}$ as a power series. The specification of $a\ /\ b = c$ is $a = c * b$, thus in our case we need to find a p such that $1 = (1 - x) * p$. For polynomials there is no solution to this equation. One way to see that is by using the homomorphism *degree*: the degree of the left hand side is 0 and the degree of the RHS is $1 + degree\ p \neq 0$. But there is still hope if we move to formal power series.

Remember that p is then represented by a stream of coefficients $[p_0, p_1, ...]$. We make a table of the coefficients of the RHS $= (1 - x) * p = p - x * p$ and of the LHS $= 1$ (seen as a power series).

```
p         == [p₀, p₁,      p₂,      ...
x ∗ p     == [0,  p₀,      p₁,      ...
p − x ∗ p == [p₀, p₁ − p₀, p₂ − p₁, ...
1         == [1,  0,       0,       ...
```

Thus, to make the last two lines equal, we are looking for coefficients satisfying $p_0 = 1$, $p_1 - p_0 = 0$, $p_2 - p_1 = 0, \ldots$. The solution is unique: $1 = p_0 = p_1 = p_2 = \ldots$ but only exists for streams (infinite lists) of coefficients. In the common math notation we have just computed

$$\frac{1}{1-x} = \sum_{i=0}^{\infty} x^i$$

Note that this equation holds when we interpret both sides as formal power series, but not necessarily if we try to evaluate the expressions for a particular x. That works for $|x| < 1$ but not for $x = 2$, for example.

For a more general case of power series division $p\ /\ q$ with $p = a : as$, $q = b : bs$, we assume that $a * b \neq 0$. Then we want to find, for any given $(a : as)$ and $(b : bs)$, the series $(c : cs)$ satisfying

```
(a : as) / (b : bs) = (c : cs)      ⇔ {- def. of division -}
(a : as) = (c : cs) ∗ (b : bs)      ⇔ {- def. of ∗ for Cons -}
```

$$\begin{aligned}
&(a : as) = (c * b) : (cs * (b : bs) + [\,c\,] * bs) \Leftrightarrow \{\text{- equality on compnents, def. of division -}\}\\
&c \;\;= a \,/\, b\{\text{- and -}\}\\
&as = cs * (b : bs) + [\,c\,] * bs \Leftrightarrow \{\text{- arithmetics -}\}\\
&c \;\;= a \,/\, b\{\text{- and -}\}\\
&cs = (as - [\,c\,] * bs) \,/\, (b : bs)
\end{aligned}$$

This leads to the implementation:

```
instance (Eq a, Fractional a) ⇒ Fractional (PowerSeries a) where
  (/) = divPS
  fromRational = Single ∘ fromRational
divPS :: (Eq a, Fractional a) ⇒ PowerSeries a → PowerSeries a → PowerSeries a
divPS as           (Single b)   = as ∗ Single (1 / b)
divPS (Single 0)   (Cons b bs)  = Single 0
divPS (Single a)   (Cons b bs)  = divPS (Cons a (Single 0)) (Cons b bs)
divPS (Cons a as)  (Cons b bs)  = Cons c (divPS (as − (Single c) ∗ bs) (Cons b bs))
                                  where c = a / b
```

The first two equations allow us to also use division on polynomials, but the result will, in general, be a power series, not a polynomial. The first one should be self-explanatory. The second one extends a constant polynomial, in a process similar to that of long division.

For example:

```
ps0, ps1, ps2 :: (Eq a, Fractional a) ⇒ PowerSeries a
ps0 = 1 / (1 − x)
ps1 = 1 / (1 − x)^2
ps2 = (x^2 − 2 ∗ x + 1) / (x − 1)
```

Every *ps* is the result of a division of polynomials: the first two return power series, the third is a polynomial (almost: it has a trailing 0.0).

```
example0  = takePoly 10 ps0
example01 = takePoly 10 (ps0 ∗ (1 − x))
```

We can get a feeling for the definition by computing *ps0* "by hand". We let $p = [1]$ and $q = [1, -1]$ and seek $r = p \,/\, q$.

$$\begin{aligned}
&divPS\ p\ q && =\\
&divPS\ [1]\ \ (1 : [-1]) && = \{\text{- 3rd case -}\}\\
&divPS\ (1 : [0])\ (1 : [-1]) && = \{\text{- 4th case -}\}\\
&(1 \,/\, 1) : divPS\ ([0] - [1] * [-1])\ (1 : [-1])\\
&1 : divPS\ ([0] - [-1])\ (1 : [-1])\\
&1 : divPS\ [1]\ (1 : [-1])\\
&1 : divPS\ p\ q
\end{aligned}$$

Thus, the answer r starts with 1 and continues with r! In other words, we have that $1 \,/\, [1, -1] = [1, 1\,..]$ as infinite lists of coefficients and $\frac{1}{1-x} = \sum_{i=0}^{\infty} x^i$ in the more traditional mathematical notation.

5.5 Formal derivative

Considering the analogy between power series and polynomial functions (via polynomials), we can arrive at a formal derivative for power series through the folowing computation:

$$
\begin{aligned}
\left(\sum_{n=0}^{\infty} a_n * x^n\right)' &= \sum_{n=0}^{\infty}(a_n * x^n)' = \sum_{n=0}^{\infty} a_n * (x^n)' = \sum_{n=0}^{\infty} a_n * (n * x^{n-1}) \\
&= \sum_{n=0}^{\infty}(n * a_n) * x^{n-1} = \sum_{n=1}^{\infty}(n * a_n) * x^{n-1} = \sum_{m=0}^{\infty}((m+1) * a_{m+1}) * x^m
\end{aligned}
\tag{1}
$$

Thus the mth coefficient of the derivative is $(m+1) * a_{m+1}$.

We can implement this, for example, as

```
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))
```

Side note: we cannot in general implement a Boolean equality test for PowerSeries. For example, we know that *deriv ps0* equals *ps1* but we cannot compute *True* in finite time by comparing the coefficients of the two power series.

```
checkDeriv :: Integer → Bool
checkDeriv n = takePoly n (deriv ps0) == takePoly n ps1
```

Recommended reading: the Functional pearl: "Power series, power serious" McIlroy [1999].

5.6 Helpers

```
instance Functor Poly where
  fmap = mapPoly
po1 :: Num a ⇒ Poly a
po1 = 1 + x^2 − 3 * x^4
instance Num a ⇒ Monoid' (Poly a) where
  unit = Single 1
  op = (*)
instance Monoid' Integer where
  unit = 0
  op = (+)
type ℕ = Integer
degree :: (Eq a, Num a) ⇒ Poly a → Maybe ℕ
degree (Single 0) = Nothing
degree (Single x) = Just 0
degree (Cons x xs) = maxd (degree (Single x)) (fmap (1+) (degree xs))
  where maxd x         Nothing = x
        maxd Nothing (Just d) = Just d
        maxd (Just a) (Just b) = Just (max a b)
checkDegree0 = degree (unit :: Poly Integer) == unit
checkDegreeM :: Poly Integer → Poly Integer → Bool
checkDegreeM p q = degree (p * q) == op (degree p) (degree q)
```

5.7 Exercises

The first few exercises are about filling in the gaps in the chapter above.

Exercise 5.1. Polynomial multiplication. To get a feeling for the definition it can be useful to take it step by step, starting with some easy cases.

```
mulP [] p =   -- TODO
mulP p [] =   -- TODO
```

```
mulP [a] p =   -- TODO
mulP p [b] =   -- TODO
```

```
mulP (0 : as) p =   -- TODO
mulP p (0 : bs) =   -- TODO
```

Finally we reach the main case

```
mulP (a : as) q@(b : bs) =   -- TODO
```

Exercise 5.2. Show (by induction) that the evaluation function *evalL* gives the same result as the formula

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

.

Exercise 5.3. Prove that, with the definition of $x = [0, 1]$ we really have

$$as = a_0 + a_1 * x + a_2 * x\hat{}2 + ... + a_n * x\hat{}n$$

Exercise 5.4. Chebyshev polynomials. Let $T_n(x) = \cos(n * \arccos(x))$. Compute T_0, T_1, and T_2 by hand to get a feeling for how it works. Note that they all turn out to be (simple) polynomial functions. In fact, T_n is a polynomial function of degree n for all n. To prove this, here are a few hints:

- $cos(\alpha) + cos(\beta) = 2\cos((\alpha + \beta)/2)\cos((\alpha - \beta)/2)$
- let $\alpha = (n + 1) * \arccos(x)$ and $\beta = (n - 1) * \arccos(x)$
- Simplify $T_{n+1}(x) + T_{n-1}(x)$ to relate it to $T_n(x)$.
- Note that the relation can be seen as an inductive definition of $T_{n+1}(x)$.
- Use induction on n.

Exercise 5.5. Another view of T_n from Exercise 5.4 is as a homomorphism. Let $H_1\ (h, F, f) = \forall\ x.\ h\ (F\ x) == f\ (h\ x)$ be the predicate that states "$h : A \to B$ is a homomorphism from $F : A \to A$ to $f : B \to B$". Show that $H_1\ (cos, (n*), T_n)$ holds, where $cos : \mathbb{R}_{\geq 0} \to [-1, 1]$, $(n*) : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$, and $T_n : [-1, 1] \to [-1, 1]$.

Exercise 5.6. Complete the following definition for polynomials represented as a plain list of coefficients:

```
instance Num a ⇒ Num [a] where
  (+) = addP
  (∗) = mulP
```

```
    -- ... TODO
addP :: Num a ⇒ [a] → [a] → [a]
addP = zipWith' (+)
mulP :: Num a ⇒ [a] → [a] → [a]
mulP =    -- TODO
```

Note that *zipWith'* is almost, but not quite, the definition of *zipWith* from the standard Haskell prelude.

Exercise 5.7. What are the ring operations on *Poly' A* where

$$Poly'\ A = \{ a : \mathbb{N} \to A \mid \text{\{- } a \text{ has only a finite number of non-zero values -\}} \}$$

Exercise 5.8. Prove the *degree* law

$$\forall\, x, y.\ degree\ (x\ `op`\ y) = degree\ x + degree\ y$$

for polynomials.

Exercise 5.9. Check all the *Monoid* and homomorphism properties in this claim: "*degree* is a monoid homomorphism from $(Poly\ a, 1, *)$ to $(Maybe\ \mathbb{N}, Just\ 0, opMaybe)$".

Exercise 5.10. The helper function $mapPoly :: (a \to b) \to (Poly\ a \to Poly\ b)$ that was used in the implementation of *polyNeg* is a close relative of the usual $map :: (a \to b) \to ([a] \to [b])$. Both these are members of a typeclass called *Functor*:

```
class Functor f where
  fmap :: (a → b) → (f a → f b)
```

Implement an instance of *Functor* for *Maybe* and *ComplexSyn* from Chapter 1 and for *Rat* from Chapter 2.

Is *fmap f* a homomorphism?