Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Datenbanksysteme · Prof. Dr. Grust

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Functional Programming

SS 2017
Benjamin Dietrich

Assignment #4

Submission Deadline: Thu, 25.5.2017

Exercise 1: Polynomials (20 Points)

What is a Number? Haskell's type system answers this question in a simple way. A number—i.e. an instance of type class `Num`—is anything that can be added, subtracted, multiplied, negated, and so on[1]:

```
Prelude> :info Num
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
    -- Defined in 'GHC.Num'
[...]
```

Can a *polynomial* be such a number? Sure! Polynomials can be added, subtracted, and multiplied just like any other number. In this exercise we are going to implement a representation for polynomials and make them an instance of `Num`.

A polynomial is a sequence of *terms*, while each term has a *coefficient* and a *degree*. For example, the polynomial $x^2 + 5x + 3$ has three terms, one of degree 2 with coefficient 1, one of degree 1 with coefficient 5, and one of degree 0 with coefficient 3.

Our representation of a polynomial in Haskell will be a list of coefficients, each of which has degree equal to its position in the list:

```
data Poly a = P [a]
```

For example, the polynomial $x^2 + 5x + 3$ is represented as `P [3,5,1]`.

The type of the coefficients is polymorphic, since we might want to support coefficients of different types.[2] However, most of the rest of this exercise only applies to polynomials with *numeric* coefficients. We thus assume `(Num a =>Poly a)`.

[1] Note that division is not included in this type class, because it is defined differently for integral.

[2] Next to integer or float polynomials there are also some applications for boolean polynomials in Cryptography, so we want to be able to represent those, too.

1. First, define a value `x` representing the polynomial $f(x) = x$.

   ```
   x :: Num a => Poly a
   ```

2. Write an instance of class `Eq` for polynomials with numeric coefficients. Note that it is not possible to simply compare the lists, since omitted coefficients may also be 0. Remember that you don't have to explicitly implement the `(/=)` function; it has a default implementation in terms of `(==)`.

   Examples:

   ```
   P [1,2,3] == P [1,2,3]
   P [1,2] /= P [1,2,3]
   ```

3. Polynomials, e.g. `P [3,2,0,1]`, should be displayed in the following—human readable—form:

   ```
   x^3 + 2*x + (-3)
   ```

   - Terms are displayed as `c*x^e` where `c` is the coefficient and `e` is the exponent. If `e` is 0, only `c` is displayed. If `e` is 1, the exponent is not displayed (`c*x`).
   - Terms are separated by the `+` sign with a single space on each side.
   - Terms are listed in *decreasing* order of their degree.
   - Terms with a coefficient 0 are not displayed, unless the whole polynomial equals 0.
   - The coefficient 1 is also not displayed, unless the degree is 0.
   - For terms with *negative* coefficients (assume `Ord a =>Poly a` to test for $c < 0$), these can either be put in parentheses or the leading operator `+` can be replaced by `-`.[3] For example, `x^3 + 2*x + (-3)` and `x^3 + 2*x - 3` are both fine.

   Make `Poly a` an instance of the `Show` class following this specification. You will need to implement the function `show :: Poly a -> String`.

   Examples:

   ```
   show (P [1,0,0,2]) == "2*x^3␣+␣1"
   show (P [0,-1,2]) == "2*x^2␣+␣(-x)"
   ```

4. We are going to start the implementation of a `Num` instance for polynomials (with numeric coefficients) with the definition for the function `fromInteger`. This function has the type `Num a =>Integer -> Poly a`. An integer c can be thought of, as a polynomial of degree 0 with coefficient c . Remember that you have to convert the `Integer` to a value of type `a` before you can use it as coefficient.

   Start your instance declaration for `Num a =>Num (Poly a)` and define `fromInteger` as a first step. The declaration is to be completed with its other necessary function definitions in the following.

5. Addition on polynomials is fairly simple. All you have to do is to add the coefficients pairwise for each term of the same degree in the two polynomials. For example $(x^2 + 5) + (2x^2 + x + 1) = 3x^2 + x + 6$.

   Write a function `plusP` which adds one polynomial to a second:

   ```
   plusP :: Num a => Poly a -> Poly a -> Poly a
   ```

   Note that the type signature for `plusP` has the constraint that `a` has a `Num` instance. Because of that you can use all of the usual `Num` functions (i.e. `(+)`) on the coefficients of your polynomials.

   Complete the definition of `(+)` in your instance using `plusP`. Examples:

   ```
   P [5,0,1] + P [1,1,2] == P [6,1,3]
   P [1,0,1] + P [1,1] == P [2,1,1]
   ```

[3] If there is no leading operator a simple an unary leading - without parentheses is fine, too.

6. To multiply two polynomials, each term in the first polynomial must be multiplied by each term in the second polynomial. The easiest way to achieve this is to build up a `[Poly a]` where each element is the polynomial resulting from multiplying a single coefficient in the first polynomial by each coefficient in the second polynomial. Since the terms do not explicitly state their exponents, you will have to shift the output before multiplying it by each consecutive coefficient (for example `P [1,1,1] * P [2,2]` will yield the list `[P [2,2], P [0,2,2], P [0,0,2,2]]`). You can then simply `sum` this list.

   Implement a function

   ```
   timesP :: Num a => Poly a -> Poly a -> Poly a
   ```

   Complete the definition of `(*)` in your instance using `timesP`. Examples:

   ```
   P [1,1,1] * P [2,2] == P [2,4,4,2]
   ```

7. Write a definition of `negate` for your instance. This function should return the negation of a polynomial. In other words, the result of negating all of its terms. For example: $3x^2 - x + 6 \equiv -(3x^2 - x + 6) \equiv -3x^2 + x - 6$ or `negate (P [6,-1,3])` $\equiv$ `P [-6,1,-3]`

   Note that with the definition of `(+)` and `negate` we get `(-)` for free, without having to implement it.

8. Write a definition of `abs :: a -> a` for your instance which turns all coefficients to positive numbers. For example: `abs (P [6,-1,3])` $\equiv$ `P [6,1,3]`

   Note: This definition is far away from a mathematical *absolute value* function for polynomials which would have to be a mapping from polynomials to numbers of $\mathbb{R}$. However, it fits the given signature for `abs` and might be a reasonable and practical interpretation of what an `abs`-function for `Poly a` in Haskell might be associated with.

9. Write a definition of `signum :: a -> a` for your instance. The `Prelude` documentation for `signum` says:

   > "The functions `abs` and `signum` should satisfy the law: `abs x * signum x == x` "

   Find and define a function `signum` which satisfies this law, according to the function `abs` defined in the previous task.

   Note: Again, our definition `signum` is—same as for `abs`—not a mathematical one.

Now that we have completed the `Num` instance for polynomials, we can stop using coefficient list syntax. The polynomial $x^2 + 5x + 3$ can now directly be written as

```
x^2 + 5*x + 3
```

which is an expression of type `Poly Int` using the overloaded operators of `Num (Poly Int)` next to the operator `(^)` which is also defined in terms of `Num a`:

```
Prelude> :t (^)
(^) :: (Num a, Integral b) => a -> b -> a
```