EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Functional Programming
Wintersemester 2021/2022
Prof. Dr. T. Grust, D. Hirn

Assignment 5 (26.11.2021)

Handin until: Friday, 03.12.2021, 00:00

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 will implement a representation for polynomials and make them an instance of `Num`.
A polynomial is a sequence of *terms*. 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, whose degrees corresponds to their position $(0,1,\dots)$ 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 coefficients is polymorphic; in particular, polynomials over integers, floats, or booleans can be represented. However, most of the rest of this exercise only applies to polynomials with *numeric coefficients*. You may thus restrict the type to `Num a => Poly a` whenever necessary.

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. Remember that you don't have to implement function `(/=)` explicitly; it has a default implementation in terms of `(==)`.

   Examples:

   ```
   P [1,2,3,0] == P [1,2,3] ≡ True
   P [1,2]     /= P [1,2,3] ≡ True
   ```

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

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

[1] Note that division is not included. There is another type class, `Integral`, for that.

- Terms with a coefficient 0 are not displayed, unless the whole polynomial equals 0.
- The coefficient 1 is not displayed, unless the degree is 0.
- Terms with negative coefficients (assume `Ord a => Poly a` to test for $c < 0$) can either be put in parentheses, or the leading operator `+` can be replaced by `-`. [2] Both versions are fine.
  Example: `(-x^3) + 2x + (-3)` or `-x^3 + 2x - 3`.

Make `Poly a` an instance of class `Show` (implement function `show :: Poly a -> String`), following the specification above.

Examples:

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

4. The first function of a `Num` instance for polynomials (with numeric coefficients) is `fromInteger :: Num a => Integer -> Poly a`. An integer c is a polynomial of degree 0 with coefficient c. Remember that you have to convert the `Integer` to a value of type `Num a => a` before you can use it as coefficient.

   Begin with an instance declaration for `Num a => Num (Poly a)` and define `fromInteger`. The declaration is to be completed with all 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]
```

6. To multiply two polynomials, each term in the first polynomial must be multiplied by each term in the second polynomial.

   Implement a function

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

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

```
P [1,2,3] * P [2,2] ≡ P [2,6,10,6]
```

   Proceed as follows:

   (a) Multiply the second polynomial with each coefficient $c_0, c_1, \ldots$ in the first polynomial separately.

```
P [1,2,3] * P [2,2]: 1 * P [2,2] = P [2,2],
                     2 * P [2,2] = P [4,4],
                     3 * P [2,2] = P [6,6]
```

   (b) Shift the result of c_i i decimal places to the right.

```
P [1,2,3] * P [2,2]: shift 0 $ 1 * P [2,2] = P [2,2],
                     shift 1 $ 2 * P [2,2] = P [0,4,4],
                     shift 2 $ 3 * P [2,2] = P [0,0,6,6]
```

   (c) Calculate the sum of all intermediate results.

```
P [1,2,3] * P [2,2] = P [2,2] + P [0,4,4] + P [0,0,6,6] = P [2,4,10,6]
```

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

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.[3] For example: `abs (P [6,-1,3])` $\equiv$ `P [6,1,3]`

9. Write a definition of `signum :: Poly a -> Poly a` for your instance.
   The "sign" of a polynomial $P = c_n x^{e_n} + \cdots + c_1 x^{e_1}$ ($c_n \neq 0$, if $n > 1$) shall be defined as:

$$signum(P) = \begin{cases} +1 & , \quad c_n > 0 \\ 0 & , \quad c_n = 0 \\ -1 & , \quad c_n < 0 \end{cases}$$

   Note: You may have to add more type class constraints to the context of your instance declarations.

   Examples:

```haskell
signum $ P [3,-2,0,1]  ≡ P [1]
signum $ P [3,-2,0,-1] ≡ P [-1]
```

10. **Bonus (optional):** Note that the `Prelude` documentation for `signum` says:

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

    Can you give an alternative implementation of `abs`—possibly different to the one we implemented in the previous subtask—such that the law is satisfied?

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

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

This is a composition of expressions of type `Poly Int`, using the overloaded operators of `Num (Poly Int)`, next to value `x` (recall your definition in 1.) and the operator `(^)`, which is also defined in terms of the `Num` instance:

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

[3]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.