PFL - 1^{st} Project

João Pereira, Nuno Pereira

October 23, 2022

1 Introduction

The goal of this project was to implement polynomials and common operations performed on them, using the Haskell programming language.

All requested functionality was implemented, including:

- Normalizing polynomials;
- Adding polynomials;
- Multiplying polynomials;
- Differentiating polynomials;
- Parsing polynomials;
- Outputting polynomials.

2 Internal Representation

For the internal representation of the `Polynomial` data structure, we implemented the following:

```haskell
data Natural = One | Suc Natural
    deriving (Eq, Ord)

type Variable = Char

type Exponent = Natural

type Coefficient = Double

data Monomial = Monomial Coefficient (Map Variable Exponent)
    deriving (Eq)

newtype Polynomial = Polynomial [Monomial]
    deriving (Eq)
```

This allows us to represent `Polynomial`s and `Monomial`s in a way that naturally represents what they are, while also ensuring that the operations performed on them are efficient:

- doing work on a `Polynomial` is (almost) the same as doing the same work to each of its `Monomial`s;

- working with the variables and degrees of each Monomial is not only simplified but also more efficient because of the nature of the underlying Map data structure.

  For example, normalizing a Polynomial is done in $\mathcal{O}\left(k'km \times \log\left(\frac{n+1}{m+1} + k'\right)\right), m \leq n$ time instead of $\mathcal{O}\left(mk^2\right)$, like we had in a previous implementation:

  - $\mathcal{O}\left(m \times \log\left(\frac{n+1}{m+1}\right)\right), m \leq n$ for aggregating any 2 Monomials, where n and m are the sizes of the Monomials' "exponent map";
  - $\mathcal{O}(k-1) = \mathcal{O}(k)$, where k is the number of Monomials in the original Polynomial;
  - $\mathcal{O}(k' \times \log(k'))$ for sorting the aggregated Monomials, where k' is the number of Monomials in the Polynomial that resulted from the previous step;

- ensuring that the exponents used are Natural numbers better models the mathematical definition of a Monomial as well as allowing to catch unexpected bugs arising from the use of negative exponents.

3 Implementation

3.1 Polynomial

- ```
  normalize :: Polynomial -> Polynomial
  normalize (Polynomial p) = Polynomial $ sortOn Down [Monomial c
     exps | (exps, c) <- toList (normalizeHelper p), c /= 0]
    where
      normalizeHelper :: [Monomial] -> Map (Map Variable Exponent)
         Coefficient
      normalizeHelper [] = empty
      normalizeHelper ((Monomial c exps) : xs) = unionWith (+)
         (fromList [(exps, c)]) (normalizeHelper xs)
  ```

  In the normalize function we process each Monomial that composes the Polynomial only once, "accumulating" the desired results, which are then re-ordered, converted back into Monomials and sorted. This strategy works because, by mapping an "exponent map" to a coefficient and exploiting the unionWith (+) function, we only need to calculate the union of all the Monomials: Monomials with the same "exponent map" simply have their coefficients added.

- ```
  instance Show Polynomial where
    show (Polynomial []) = ""
    show (Polynomial [m]) = show m
    show (Polynomial (m : (Monomial c e) : ms)) = show m ++ showSign
       c ++ show (Polynomial (abs (Monomial c e) : ms))
      where
        showSign c = if c < 0 then " - " else " + "
  ```

  In the show function the Monomials are printed one at a time using the coefficient of the next Monomial in the list to define the sign to show: '-' if it is negative, '+' otherwise.

- ```
  instance Differentiable Polynomial where
    p // v = normalize $ p !// v
  ```

  Makes a Polynomial inherently differentiable. This functions basically differentiates each Monomial and then normalizes the output.
  ```

- `instance Num Polynomial where`
  ```
  x + y = normalize $ x !+ y

  x * y = normalize $ x !* y

  abs (Polynomial x) = Polynomial $ map abs x

  signum = error "Not implemented"

  fromInteger x = Polynomial [fromInteger x]

  negate (Polynomial x) = Polynomial $ map negate x
  ```

  Makes a `Polynomial` be treated as a `Num` so that common number operations can be performed on `Polynomial`s. This is basically a way to perform the same operations on all the `Monomial`s that make up a `Polynomial`.

- `instance Read Polynomial where`
  ```
  readsPrec _ s = [(Polynomial $ read s, "")]
  ```

  Utility instantiation of `Read` so that `Polynomial`s can be parsed directly from an input string.

- `(!+) :: Polynomial -> Polynomial -> Polynomial`
  ```
  Polynomial x !+ Polynomial y = Polynomial $ x ++ y
  ```

  Adds two `Polynomial`s, without normalizing the output. Since `Polynomial`s are just a list of `Monomial`s, this just creates a `Polynomial` that results from the concatenation of all the `Monomial`s in the two input `Polynomial`s.

- `(!-) :: Polynomial -> Polynomial -> Polynomial`
  ```
  x !- y = x !+ negate y
  ```

  Subtracts two `Polynomial`s, without normalizing the output. This just calls `(!+)` but having the second `Polynomial` negated.

- `(!*) :: Polynomial -> Polynomial -> Polynomial`
  ```
  Polynomial x !* Polynomial y = Polynomial [i * j | i <- x, j <- y]
  ```

  Multiplies both `Polynomial`s, without normalizing the output. This is basically performing a Cartesian product between the inputs' `Monomial`s.

- `(!//) :: Polynomial -> Variable -> Polynomial`
  ```
  Polynomial p !// v = Polynomial $ map (// v) p
  ```

  Differentiates a `Polynomial` without normalizing the output. This just applies a mapping to differentiate each of this `Polynomial`'s `Monomial`s.

3.2 Monomial

- `instance Differentiable Monomial where`
  ```
  Monomial n m // v = case Data.Map.lookup v m of
    Nothing -> 0
    Just One -> Monomial n (delete v m)
    Just exp -> Monomial (n * fromInteger (toInteger exp)) (adjust
       (\x -> x - One) v m)
  ```

Makes a `Monomial` inherently differentiable. If the new exponent of the variable of differentiation in the `Monomial` is 1, then the exponent is removed from this `Monomial`'s "exponent map". Otherwise, the model remains the same, having in mind the differentiation rules of mathematical monomials.

- ```
  instance Num Monomial where
    Monomial c1 m1 + Monomial c2 m2
      | m1 == m2 = Monomial (c1 + c2) m1
      | otherwise = error "Can't add monomials with different
        degrees"

    Monomial c1 m1 * Monomial c2 m2 = Monomial (c1 * c2) (unionWith
        (+) m1 m2)

    abs (Monomial n m) = Monomial (abs n) m

    signum (Monomial n m) = Monomial (signum n) empty

    fromInteger n = Monomial (fromInteger n) empty

    negate (Monomial n m) = Monomial (-n) m
  ```

  Makes a `Monomial` be treated as a `Num`, so that common number operations can be performed on them.

- ```
  instance Ord Monomial where
    Monomial ca ea <= Monomial cb eb
      | ea == eb = ca <= cb
      | otherwise = or $ zipWith compare (toList ea) (toList eb)
      where
        compare (v1, e1) (v2, e2)
          | v1 == v2 = e1 <= e2
          | otherwise = v1 >= v2
  ```

  Makes it so that `Monomial`s can be ordered. This is especially useful when normalizing `Polynomial`s.

- ```
  instance Show Monomial where
    show (Monomial c m)
      | Data.Map.null m = showReal c
      | c == 1 = showVars m
      | c == (-1) = '-' : showVars m
      | otherwise = showReal c ++ showVars m
      where
        showVars m = concat $ mapWithKey showVar m
        showVar v 1 = [v]
        showVar v e = v : showSuperscript (toInteger e)
  ```

  Utility instanciation of `Show` for easier printing of `Monomial`s.

- ```
  instance Read Monomial where
    readsPrec _ s = [readMonomial s]
      where
        readMonomial :: String -> (Monomial, String)
        readMonomial s' = (Monomial coeff vars, s''')
          where
            (coeff, s'') = readCoefficient s'
  ```

```
          (vars, s''') = readVars s''

      readCoefficient :: String -> (Coefficient, String)
      readCoefficient ('-' : cs) = (-f, s)
        where
          (f, s) = readCoefficient cs
      readCoefficient ('+' : cs) = readCoefficient cs
      readCoefficient (' ' : cs) = readCoefficient cs
      readCoefficient cs = case f of
        "" -> (1, s)
        f -> (read f, s)
        where
          (f, s) = span (\c -> isDigit c || c == '.') cs

      readVars :: String -> (Map Variable Exponent, String)
      readVars vs = (fromList $ readVarsHelper f, s)
        where
          (f, s) = span (\c -> isSpace c || isDigit c ||
             isAsciiLower c || c == '*' || c == '^') vs

          readVarsHelper :: String -> [(Variable, Exponent)]
          readVarsHelper "" = []
          readVarsHelper (' ' : s) = readVarsHelper s
          readVarsHelper s = (v, e) : readVarsHelper s''
            where
              readVar :: String -> (Variable, String)
              readVar "" = error "No read"
              readVar (v : vs)
                | v == '*' || v == ' ' = readVar vs
                | isAsciiLower v = (v, vs)
                | otherwise = error "No read"

              (v, s') = readVar s

              readExponent :: String -> Exponent
              readExponent "" = One
              readExponent ('^' : es) = readExponent es
              readExponent (' ' : es) = readExponent es
              readExponent e = fromInteger $ read e

              (e, s'') = (readExponent es, s'')
                where
                  (es, s'') = span (\c -> isSpace c || isDigit c
                     || c == '^') s'

  readList s = [(helper s, "")]
    where
      helper "" = []
      helper s = m : helper s'
        where
          (m, s') = head (reads s :: [(Monomial, String)])
```

Utility instanciation of `Read` that allows a `Monomial` to be parsed from an input string.

3.3 Natural

- `instance Enum Natural where`

```haskell
  toEnum n
    | n == 1 = One
    | n > 1 = Suc $ toEnum (n - 1)
    | otherwise = error "Cannot be negative"

  fromEnum One = 1
  fromEnum (Suc n) = 1 + fromEnum n
```

Utility instantiation of `Enum` that allows `Natural`s to be converted from and to `Integer`s.

- ```haskell
  instance Num Natural where
    a + b = toEnum $ fromEnum a + fromEnum b

    a * b = toEnum $ fromEnum a * fromEnum b

    a - b
      | number <= 0 = error "Negative difference"
      | otherwise = toEnum number
      where
        number = fromEnum a - fromEnum b

    abs n = n

    signum p = 1

    fromInteger n
      | n < 1 = error "Must be positive"
      | n == 1 = One
      | otherwise = Suc (fromInteger (n Prelude.- 1))

    negate = error "Cannot negate value"
  ```

  Makes it so that `Natural`s are treated as native number types.

- ```haskell
  instance Show Natural where
    show = show . fromEnum
  ```

  Utility instantiation of `Show` that facilitates printing `Natural` numbers.

- ```haskell
  instance Real Natural where
    toRational x = toInteger x % 1
  ```

  Utility instantiation of `Real` that is needed to be able to instantiate `Integral`.

- ```haskell
  instance Integral Natural where
    quotRem x y = (toEnum q, toEnum r)
      where
        (q, r) = quotRem (fromEnum x) (fromEnum y)

    toInteger = toInteger . fromEnum
  ```

  Utility instantiation of `Integral` that makes it so that `Natural` numbers are treated as `Integer` numbers.

4 Usage examples

As the program is designed to be used inside of `ghci`, all examples will assume the program has been ran as `ghci Main`.

4.1 Monomials

4.1.1 Input and output

A `Monomial` can be inputted directly as its internal representation:

```
ghci> Monomial 7 (fromList [('y', 2), ('x', 4)])
7x⁴y²
```

Or by using `read` and inputting a properly formatted string:

```
ghci> read "7*y^2*x^4" :: Monomial
7x⁴y²
```

A simplified format is also supported:

```
ghci> read "7y2x4" :: Monomial
7x⁴y²
```

This function will only accept `Monomial`s with their coefficient as the first term, and will not accept parenthesis. The syntax accepted is roughly equivalent to the regular expression `[+-]?\d*\.?\d*(\*?[a-z]\^?\d*)*`, but whitespace is also accepted between terms.

Monomials will always be outputted in their formatted form by `ghci`, as the class `Show` has been instanced. As so, `show` can also be used to get a string with the formatted `Monomial`.

```
ghci> show (read "7y2x4" :: Monomial)
"7x\8308y\178"
```

4.1.2 Normalization

A `Monomial` is always inherently normalized because of the internal data structure used.

4.1.3 Addition, subtraction and multiplication

The following examples will use the `Monomial`s:

$$M_1 = 7x^2$$
$$M_2 = 12y$$
$$M_3 = 5y$$

To add `Monomial`s, use the `(+)` operator. Only `Monomial`s with matching degrees can be added.

```
ghci> m1 + m2
*** Exception: Can't add monomials with different degrees
CallStack (from HasCallStack):
  error, called at ./Data/Monomial.hs:27:19 in main:Data.Monomial
ghci> m2 + m3
17y
```

To subtract `Monomial`s, use the `(-)` operator. Only `Monomial`s with matching degrees can be subtracted.

```
ghci> m1 - m2
*** Exception: Can't add monomials with different degrees
CallStack (from HasCallStack):
  error, called at ./Data/Monomial.hs:27:19 in main:Data.Monomial
ghci> m2 - m3
7y
```

To multiply `Monomial`s, use the `(*)` operator.

```
ghci> m1 * m2
84x²y
```

4.1.4 Differentiation

To differentiate a `Monomial`, use the `(//)` operator. The second argument is the variable to differentiate by.

```
ghci> (read "7y2x4" :: Monomial) // 'x'
28x³y²
ghci> (read "7y2x4" :: Monomial) // 'y'
14x⁴y
```

4.1.5 Other operations

To get the absolute value of a `Monomial`, use the `abs` function. This will make the coefficient non-negative.

```
ghci> abs $ read "-7y2x4" :: Monomial
7x⁴y²
```

To get the sign of a `Monomial`, use the `signum` function. This will return the sign of the coefficient.

```
ghci> signum $ read "7y2x4" :: Monomial
1
ghci> signum $ read "-7y2x4" :: Monomial
-1
```

To negate a `Monomial`, use the `negate` function. This will flip the sign of the coefficient.

```
ghci> negate $ read "7y2x4" :: Monomial
-7x⁴y²
ghci> negate $ read "-7y2x4" :: Monomial
7x⁴y²
```

To check if two `Monomial`s have the same degree, use the `(~=)` operator, or the `(~/=)` operator for the opposite.

```
ghci> (read "7y2x4" :: Monomial) ~= (read "4x" :: Monomial)
False
ghci> (read "7y2x4" :: Monomial) ~/= (read "4x" :: Monomial)
True
```

4.2 Polynomials

4.2.1 Input and output

A `Polynomial` can be inputted directly as its internal representation:

```
ghci> Polynomial [Monomial 0 (fromList [('x', 2)]), Monomial 2
    (fromList [('y', 1)]), Monomial 5 (fromList [('z', 1)]), Monomial 1
    (fromList [('y', 1)]), Monomial 7 (fromList [('y', 2)])]
0x² + 2y + 5z + y + 7y²
```

Or by using read and inputting a properly formatted string:

```
ghci> read "0*x^2 + 2*y + 5*z + y + 7*y^2" :: Polynomial
0x² + 2y + 5z + y + 7y²
```

A simplified format is also supported:

```
ghci> read "0x2 + 2y + 5z + y + 7y2" :: Polynomial
0x² + 2y + 5z + y + 7y²
```

This function has the same limitations as the version for `Monomial`s.

Polynomials will always be outputted in their formatted form by `ghci`, as the class `Show` has been instanced. As so, `show` can also be used to get a string with the formatted `Polynomial`.

```
ghci> show (read "0x2 + 2y + 5z + y + 7y2" :: Polynomial)
"0x\178 + 2y + 5z + y + 7y\178"
```

4.2.2 Normalization

A `Polynomial` can be normalized using the `normalize` function. Polynomials will also get normalized after most operations, but this can be skipped.

```
ghci> normalize $ read "0*x^2 + 2*y + 5*z + y + 7*y^2" :: Polynomial
7y² + 3y + 5z
```

4.2.3 Addition, subtraction and multiplication

The following examples will use the `Polynomial`s:

$$P_1 = x^3 + x^2 + 12y + 0$$

$$P_2 = 4x + 5y + 8 + 10z$$

To add `Polynomial`s, use the `(+)` operator, or the `(!+)` operator, if normalization is to be skipped.

```
ghci> p1 + p2
x³ + x² + 4x + 17y + 10z + 8
ghci> p1 !+ p2
x³ + x² + 12y + 0 + 4x + 5y + 8 + 10z
```

To subtract `Polynomial`s, use the `(-)` operator, or the `(!-)` operator, if normalization is to be skipped.

```
ghci> p1 - p2
x³ + x² - 4x + 7y - 10z - 8
ghci> p1 !- p2
x³ + x² + 12y + 0 - 4x - 5y - 8 - 10z
```

To multiply `Polynomial`s, use the `(*)` operator, or the `(!*)` operator, if normalization is to be skipped.

```
ghci> p1 * p2
4x⁴ + 12x³ + 5x³y + 10x³z + 8x² + 48xy + 5x²y + 10x²z + 60y² + 96y + 120yz
ghci> p1 !* p2
4x⁴ + 5x³y + 8x³ + 10x³z + 4x³ + 5x²y + 8x² + 10x²z + 48xy + 60y² + 96y + 120yz + 0x +
    0y + 0 + 0z
```

4.2.4 Differentiation

To differentiate a `Polynomial`, use the `(//)` operator, or the `(!//)` operator, if normalization is to be skipped. The second argument is the variable to differentiate by.

```
ghci> (read "x3 + x2 + 12y + 5" :: Polynomial) // 'x'
3x² + 2x
ghci> (read "x3 + x2 + 12y + 5" :: Polynomial) !// 'x'
3x² + 2x + 0 + 0
ghci> (read "x3 + x2 + 12y + 5" :: Polynomial) // 'y'
12
ghci> (read "x3 + x2 + 12y + 5" :: Polynomial) !// 'y'
0 + 0 + 12 + 0
```

4.2.5 Other operations

To get the absolute value of a `Polynomial`, use the `abs` function. This will make all coefficients non-negative.

```
ghci> abs $ read "-x3 + x2 - 12y + 5" :: Polynomial
x³ + x² + 12y + 5
```

To negate a `Polynomial`, use the `negate` function. This will flip the sign of all coefficients.

```
ghci> negate $ read "-x3 + x2 - 12y + 5" :: Polynomial
x³ - x² + 12y - 5
```