JPP 2022/23 — Program zaliczeniowy (Haskell)

Wielomiany

Przedmiotem tego zadania są operacje na wielomianach w różnych reprezentacjach.

Operacje te mają być uniwersalne, pozwalające działać na wielomianach nad dowolnym pierścieniem bądź ciałem (czyli nie tylko R, ale również Z, Q, czy Z/pZ).

Interfejs wielomianu określa następująca klasa `Polynomial`:

```
class Polynomial p where
  zeroP  :: p a                             -- zero polynomial
  constP :: (Eq a, Num a) => a -> p a       -- constant polynomial
  varP   :: Num a => p a                    -- p(x) = x
  evalP  :: Num a => p a -> a -> a          -- value of p(x) at given x
  shiftP :: (Eq a, Num a) => Int -> p a -> p a    -- multiply by x^n
  degree :: (Eq a, Num a) => p a -> Int     -- highest power with nonzero coefficient
  nullP  :: (Eq a, Num a) => p a -> Bool    -- True for zero polynomial
  nullP p = degree p < 0
  x      :: Num a => p a                    -- p(x) = x
  x       = varP
```

dla wielomianu zerowego `degree` powinno dawąc wynik `(-1)`

A. Reprezentacja gęsta

Reprezentacja w postaci listy współczynników, poczynając od najniższej potęgi:

```
-- | Polynomial as a list of coefficients, lowest power first
-- e.g. x^3-1 is represented as P [-1,0,0,1]
-- canonical: no trailing zeros
newtype DensePoly a = P { unP :: [a] } deriving Show

sampleDP = P [-1,0,0,1]
```

Kanoniczna reprezentacja nie ma zer na końcu listy (w szczególności wielomian zerowy reprezentowany jest jako lista pusta)

Pomocniczo może być użyteczna reprezentacja w odwrotnej kolejności:

```
-- | Polynomial as a list of coefficients, highest power first
-- e.g. x^3-1 is represented as R [1,0,0,-1]
-- canonical: no leading zeros
newtype ReversePoly a = R { unR :: [a] } deriving Show

sampleRP = R [1,0,0,-1]
```

Uzupełnij instancje klas:

```
instance Functor DensePoly where

instance Polynomial DensePoly where

instance (Eq a, Num a) => Num (DensePoly a) where

instance (Eq a, Num a) => Eq (DensePoly a) where
```

Zaimplementowane metody klas `Num` i `Polynomial` powinny dawać w wyniku reprezentację kanoniczną, także dla argumentów niekanonicznych.

W klasie `Num` metody `abs` i `signum` mogą użyć `undefined`.

Przykładowe własności (patrz sekcja **E. Własności i testy**)

```
-- >>>  P [1,2] == P [1,2]
-- True

-- >>> fromInteger 0 == (zeroP :: DensePoly Int)
-- True

-- >>>  P [0,1] == P [1,0]
-- False
```

B. Reprezentacja rzadka

W przypadku wielomianów rzadkich (np. `1+x^2023`) tudzież przy operacjach takich jak dzielenie, lepsza reprezentacja wielomianu w postaci listy par
`(potęga, współczynnik)`:

```
-- | Polynomial as a list of pairs (power, coefficient)
-- e.g. x^3-1 is represented as S [(3,1),(0,-1)]
-- invariant: in descending order of powers; no zero coefficients
newtype SparsePoly a = S { unS :: [(Int, a)] } deriving Show

sampleSP = S [(3,1),(0,-1)]
```

Kanoniczna reprezentacja wielomianu nie ma współczynników zerowych i jest uporządkowana w kolejności malejących wykładników.

Uzupełnij instancje klas w module SparsePoly:

```
instance Functor SparsePoly where

instance Polynomial SparsePoly where

instance (Eq a, Num a) => Num (SparsePoly a) where

instance (Eq a, Num a) => Eq (SparsePoly a) where
```

W klasie `Num` metody `abs` i `signum` mogą użyć `undefined`.

Zaimplementowane metody klas `Num` i `Polynomial` powinny dla argumentów w postaci kanonicznej dawać wyniku w postaci kanonicznej.

Wskazówka do `Functor`: napisz funkcje

```
first :: (a -> a') -> (a, b) -> (a', b)
second :: (b -> b') -> (a, b) -> (a, b')
```

C. Konwersje

Zaimplementuj w module SparsePoly konwersje pomiędzy powyższymi reprezentacjami:

```
fromDP :: (Eq a, Num a) => DensePoly a -> SparsePoly a
toDP :: (Eq a, Num a) => SparsePoly a -> DensePoly a
```

Konwersje powinny dawać w wyniku reprezentację kanoniczną.

Na zbiorach reprezentacji kanonicznych, złożenia powyższych funkcji mają być identycznościami.

Uwaga: niezależnie od istnienia konwersji, operacje arytmetyczne należy implementować na właściwych reprezentacjach, bez używani konwersji.

D. Dzielenie z resztą

Jeżeli `p, s` są wielomianami nad pewnym ciałem i `s` nie jest wielomianem zerowym, istnieje dokładnie jedna para wielomianów `q,r` taka, że `q` jest ilorazem, zaś `r` resztą z dzielenia `p` przez `s`, to znaczy stopień `r` jest mniejszy niż stopień `s` oraz

```
p = q*s + r
```

Napisz funkcję

```
qrP :: (Eq a, Fractional a)
    => SparsePoly a -> SparsePoly a -> (SparsePoly a, SparsePoly a)
```

realizującą takie dzielenie.

Wskazówka: klasa `Fractional` definiuje dzielenie:

```
(/) :: Fractional a => a -> a -> a
```

E. Własności i testy

Oprócz własności opisanych powyżej, operacje arytmetyczne na wielomianach powinny spełniać wszystkie typowe własności (przemienność, łączność, rozdzielność, elementy neutralne, element odwrotny względem dodawania itp).

Niektóre własności są objęte testami. testy są dwojakiego rodzaju:

- przykłady wartości wyrażeń, np.

```
-- >>> degree (zeroP :: DensePoly Int)
-- -1
```

tego typu testy można sprawdzić przy pomocy narzędzia doctest albo obliczając wartości wyrażeń w GHCi (bądź VS Code)

- własności QuickCheck, np.

```
type DPI = DensePoly Integer

propAddCommDP :: DPI -> DPI -> Property
propAddCommDP p q = p + q === q + p
```

taką własność należy rozumieć jako „dla dowolnych p, q typu `DensePoly Integer` wartość `p + q` jest równa wartości `q + p`". Spełnienie takich własności można wyrywkowo sprawdzić przy pomocy biblioteki QuickCheck (np. uruchamiając program `THTestPoly.hs`).

Jak to zwykle z testami, pomyślne przejscie testów nie gwarantuje, że rozwiązanie jest poprawne, natomiast niepomyślne wskazuje, że jest niepoprawne.

Oddawanie

Należy oddać tylko pliki `SparsePoly.hs` i `DensePoly.hs` stworzone poprzez uzupełnienie dostarczonych plików `*-template.hs`

Elementów istniejących już w tych plikach nie wolno modyfikować ani usuwać (oczywiście za wyjątkiem `undefined`). Dotyczy to także (a nawet szczególnie) komentarzy zawierających testy.