Programowanie w logice i funkcyjne (Haskell), leniwa ewaluacja, typy itp. (prezentacja niekompletna i przed korektą)

Konrad Zdanowski

Outline

Weak Head Normal Form (WHNF)

- Wyrażenie jest w postaci normalnej (NF), jeśli wszystkie jest w pełni zredukowane (nie można wykonać już żadnej redukcji).
- Wyrażenie jest w słabej głowowej postaci normalnej (WHNF) jeśli jest w nim zredukowany główny operator.
- Każdy term w NF jest też w WHNF.
- Główny operator termu w WHNF to
  - konstruktor danych (data constructor),
  - lambda abstrakcja (funkcja bez argumentów).

Przykłady

- Termy w postaci normalnej:
  `31, 'a', sin, (\ x y -> x+y)`
- Podobnie `"ala", (3,"ala")`
- Termy w WHNF (lecz nie w postaci normalnej):
  - `(pi, 3-1), (\x -> x+ (3+5))`
  - `x:"la ma kota", 'a':"la ma kota".`
- Term nie jest w WHNF jeśli możemy zredukować jego główny operator:
  - `2+2, "a"++"la", sin 4, (\x -> x+y) 3`

Leniwa ewaluacja

- Haskell stosuje leniwą ewaluację tzn oblicza tylko te elementy termu, które są potrzebne do wykonania obliczenia.
- Term, którego ewaluacja jest, aktualnie, tylko częściowa nazywa się *thunk*.
- Jeśli fragment termu nie będzie wykorzystany w obliczeniu, to Haskell go nie zredukuje (unikając, być może, błędów).
- Powoduje to, że rozmiar termu podczas obliczenia programu może być duży, gdyż Haskell nie wykonuje niepotrzebnych redukcji (koszt pamięciowy).
- Pewne operacje, jak wypisanie, wymuszają pełną ewaluację termu.

Przykład – leniwa ewaluacja

- `let (x,y)=(1,[]!!1) in x+3` nie generuje wyjątku,
- `length [[]!!1, undefined]` ewaluuje się do 2 bo Haskell nie potrzebuje zredukować elementów listy, żeby policzyć jej długość.
- Pomimo leniwej ewaluacji Haskell kontroluje typy. Następująca konstrukcja wygeneruje błąd `length [1,"ala"]`

Zalety leniwej ewaluacji

- Liczymy tylko to, co konieczne.
  - Pewne obliczenia wykonamy szybciej, `take 2 (selectionsort xs)`,
  - Pewne obliczenia wykonamy, np. `take 2 [1..]`
- Wykorzystywane funkcje są obliczane tylko do takiej głębokości jaka jest konieczna (przykład za Wikibooks, Haskell, Laziness). Pozwala to na efektywniejsze wykorzystanie istniejącego kodu:

```
isInfixOf :: Eq a => [a] -> [a] -> Bool
isInfixOf x y = any (isPrefixOf x) (tails y)
```

- Pozwala manipulować na (potencjalnie) nieskończonych strukturach.
- Pozwala tworzyć struktury odnoszące się do siebie.
  - `let x=[1,2,3]++x in x!!8`

foldl'

- Czasem chcemy zredukować coś szybciej.
- `seq::a->b->b`
- `seq x y` najpierw redukuje `x` a potem zwraca `y`
- `Prelude> import Data.List`
- ```
  foldl' f z []     = z
  foldl' f z (x:xs) = let z' = z `f` x
                      in seq z' $ foldl' f z' xs
  ```
-
  ```

Outline

Funkcje typów w Haskellu

- Kontrola kodu.
- Polimorizm.
- ...

- Silny system typów:
  - kontrola poprawności wyrażen,
  - brak ukrytych rzutowań.
- Typy statyczne.
- Wnioskowanie o typach.
- Jaki jest typ

```
foldr (\x y -> x && y) ?
```

- Polimorfizm:
  - polimorficzne funkcje, np. f::[a]->[a],
  - polimorficzne typy, np. [a].

Deklaracja typu

```
data Bool = False | True
```

Typy numeryczne

- Int, Integer
- Float, Double
- Rational (1/2::Rational) – reprezentowany przez dwie liczby Ingeger, dowolna precyzja
- Scientific (biblioteka Scientific) – format reprezentujący ma postać mantysy m::Integer i wykładnika w::Int, (m* exp(10,w)). Np. 2030482e-4 reprezentuje 203,0482.
- Scientific nie zużywa tyle pamięci co Rational (np. 1e100000).
- Typy ograniczone mają minBound i maxBound.

Typy numeryczne

- Obsługa typu Rational

```
Prelude> import Data.Ratio
Prelude> 2% 12 * (3 % 7 )
1%14
```

- Możemy też po prostu rzutowac na typ Rational

```
Prelude (1/3)::Rational
1%3
```

Typy numeryczne

- Instalacja biblioteki scientific:
  - Instalujemy program do zarządzania pakietami Haskella (cabal).
  - ```
    $ cabal update
    $ cabal install scientific
    ```
  - ```
    Prelude> import Data.Scientific
    Prelude> scientific 123 (-1)
    12.3
    Prelude> 123e-1
    12.3
    Prelude> 123 e -1
    <interactive>:90:1: error:
        • Non type-variable argument in the constraint: Num
    ```

- Uwaga. Scientific reprezentuje wartości dokładnie, więc nie radzi sobie z wielkościami, które nie mają skończonego rozwinięcia dziesiętnego.

```
Prelude> 1/3
0.333333333333333
Prelude> 1/3::Rational
1 % 3
Prelude> 1/3::Scientific
*** Exception: fromRational has been applied
to a repeating decimal which can't be represented
as a Scientific!
It's better to avoid performing fractional operations
on Scientifics and convert them to other fractional t
like Double as early as possible.
CallStack (from HasCallStack):
  error, called at src\\Data\\Scientific.hs:311:23 in
```

- Typ Scientific nie jest domknięty na dzielenie!

- Kontrola typów sprawia, że rzutowania musimy wykonywać jawnie.
- Tutaj Haskell domyślnie przyjmuje, że argumenty dzielenia są typu ułamkowego:

```
Prelude> :t 2/3
2/3 :: Fractional a => a
```

- Wymuszenie typu Int spowoduje błąd:

```
Prelude> 2/(2::Int)
<interactive>:53:1: error:
    • No instance for (Fractional Int) arising from a
    • In the expression: 2 / (2 :: Int)
      In an equation for ‘it’: it = 2 / (2 :: Int)
```

- Argument `/` musi być z klasy Fractional bo

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

- ```
  Prelude> 2 / (length [1,2,3])
  <interactive>:55:1: error:
      • No instance for (Fractional Int) arising from a
      • In the expression: 2 / length [1, 2, 3]
        In an equation for ‘it’: it = 2 / length [1, 2,
  ```

- Musimy wykonać jawne rzutowanie

  ```
  Prelude>  2 / fromIntegral (length [1,2,3])
  0.6666666666666666
  ```
  ```

Typy złożone – n-tki i listy

- ```
  Prelude> :info (,)
  data (,) a b = (,) a b  -- Defined in ‘GHC.Tuple’
  ```
- fst (x,y), snd (x,y)
- Mamy też n-tki, dla n>2: (1,2,3), ...
- Listy były opisane wcześniej.
  ```

Currying, Uncurrying

- Funkcje w Haskellu są jednoargumentowe.
- Np. `foldr::(a->b->b) -> (b->[a]->b)`
- ```
  Prelude> :t curry
  curry :: ((a, b) -> c) -> a -> b -> c
  Prelude> :t uncurry
  uncurry :: (a -> b -> c) -> (a, b) -> c
  ```
- ```
  Prelude> let ucFoldr = uncurry foldr
  Prelude> ucFoldr ((+),0) [1,2,3]
  6
  Prelude> let ucucFoldr = uncurry (uncurry foldr)
  Prelude> ucucFoldr (((+),0), [1,2,3])
  6
  ```
- Jaki jest typ `uncurry uncurry`?

Konstruktory danych

- Sposób na tworzenie danych danego typu
- Konstruktor danych zaczyna się dużą literą.
- ```
  type Imie = String
  data Zwierzak = Kot | Pies Imie

  Prelude> :t Kot
  Kot :: Zwierzak
  Prelude> :t Pies
  Pies :: Imie -> Zwierzak
  ```
  ```

Konstruktory typów

- Pozwalają na tworzenie nazw typów.
- Np. ([]), (->)
- Funkcja fst ma typ (a,b)-> a (użyliśmy dwóch konstruktorów typów)

Typy struktur dynamicznych – przykład

Klasa typów

- Klasa typów definiuje zbiór operacji na typie, np. Eq a, Show a, Ord a, ...

Możemy sprawdzić do jakich klas należy dany typ:

```
Prelude> :info Bool
data Bool = False | True  -- Defined in ‘GHC.Types’
instance Eq Bool -- Defined in ‘GHC.Classes’
instance Ord Bool -- Defined in ‘GHC.Classes’
instance Show Bool -- Defined in ‘GHC.Show’
instance Read Bool -- Defined in ‘GHC.Read’
instance Enum Bool -- Defined in ‘GHC.Enum’
instance Bounded Bool -- Defined in ‘GHC.Enum’
```

Możemy sprawdzić, co oferuje dana klasa:

```
Prelude> :info Eq
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  {-# MINIMAL (==) | (/=) #-}
   -- Defined in ‘GHC.Classes’
instance (Eq a, Eq b) => Eq (Either a b)
  -- Defined in ‘Data.Either’
instance Eq a => Eq [a] -- Defined in ‘GHC.Classes’

instance Eq Word -- Defined in ‘GHC.Classes’
instance Eq Ordering -- Defined in ‘GHC.Classes’
instance Eq Int -- Defined in ‘GHC.Classes’
instance Eq Float -- Defined in ‘GHC.Classes’
....
```

```
instance Eq Zwierzak where
 Kot == Kot = True
 Pies imie1 == Pies imie2 = imie1 == imie2
 _ == _ = False
```

```
Prelude> :info Ord
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
  {-# MINIMAL compare | (<=) #-}
   -- Defined in ‘GHC.Classes’
instance (Ord a, Ord b) => Ord (Either a b)
  -- Defined in ‘Data.Either’
instance Ord a => Ord [a] -- Defined in ‘GHC.Classes’
instance Ord Word -- Defined in ‘GHC.Classes’
instance Ord Ordering -- Defined in ‘GHC.Classes’
instance Ord Int -- Defined in ‘GHC.Classes’
...
```

W szczególności klasa Ord zawiera się w klasie Eq.

- Haskell sam tworzy odpowiednie funkcje dla typów złożonych
- ```
  Prelude> (1,2)<(2,2)
  True
  Prelude> [1,2,3]<[1,2]
  False
  ```
  ```

Outline

Definicja drzewa

- Deklarujemy typ danych: drzewo elementów typu a

```
data Drzewo a = Puste | Wezel a
                        (Drzewo a) (Drzewo a)
                        deriving (Eq, Ord)
```

- `Drzewo` jest konstruktorem typu.
- Inny taki konstruktor to `data [] a = [] | [a]`

- `Drzewo` jest kind'em, dopiero gdy ukonkretnimy typ `a` to otrzymamy typ drzew zawierających elementy typu `a`.
- ```
  Prelude> :kind Drzewo
  Drzewo :: * -> *
  Prelude> :kind Drzewo Int
  Drzewo Int :: *
  ```
- ```
  Prelude> :t Wezel 2 Puste Puste
  Wezel 2 Puste Puste :: Num a => Drzewo a
  Prelude> :t Wezel (2::Int) Puste Puste
  Wezel (2::Int) Puste Puste :: Drzewo Int
  ```

Dodawanie elementu do drzewa BST

- Możemy teraz napisać funkcję dodającą element:

```
dodajDoDrzewa::Ord a => a-> Drzewo a -> Drzewo a

dodajDoDrzewa x Puste = Wezel x Puste Puste
dodajDoDrzewa x t@(Wezel y lDrzewo pDrzewo)
  | x<y = (Wezel y
           (dodajDoDrzewa x lDrzewo) (pDrzewo))
  | x==y = t
  | x>y = (Wezel y
            (lDrzewo) (dodajDoDrzewa x pDrzewo))
```

Wypisywanie drzewa za pomoca show

- Napiszemy definicję metody z klasy Show

```
instance (Show a) =>Show (Drzewo a) where
  show Puste = "()"
  show (Wezel x lDrzewo pDrzewo) =
         "("++(show x )++", "++
           (show lDrzewo)++", "++
           (show pDrzewo)++")"
```

```
Prelude> dodajDoDrzewa 4 (dodajDoDrzewa 3
           (dodajDoDrzewa 2 Puste))
(2, (), (3, (), (4, (), ())))
```

Outline

Maybe

-
- `data Maybe a = Nothing | Just a`
-
```
data MyMaybe a = MyNothing | MyJust a
           deriving Show

dzielenie:: (Eq a, Fractional a)=> (a) -> (a) -> (MyM
dzielenie (x) (y)
    | x==0&&y==0 = MyNothing
    | True = MyJust (x/y)
```

-
```
Prelude> dzielenie 0 0
MyNothing
Prelude> dzielenie 1 0
MyJust Infinity
Prelude> dzielenie 2 3
MyJust 0.6666666666666666
```

Przykład: lookup

- `lookup::Eq a => a -> [(a,b)] -> Maybe b`
- ```
  Prelude> lookup 3 [(2,"ala"), (3,"ela")]
  Just "ela"
  Prelude> lookup 4 [(2,"ala"), (3,"ela")]
  Nothing
  ```
  ```

Outline

Polimorfizm

- Opisując typ funkcji możemy użyć zmiennych typowych (małe litery).
- `id::a->a` opisuje, że id może pobrać argument dowolnego typu i zwrócić wartość tego typu.
- Jeśli definicja funkcji wymaga pewnych specyficznych operacji, to wyrażamy to nakładając na typ pewne warunki.

```
Prelude> let f x = if (x==x) then x+1 else x
Prelude> :t f:: (Eq p, Num p) => p -> p
```

Polimorfizm

- W jednym wyrażeniu funkcja może być użyta jako funkcja o różnych typach.

```
Prelude> id succ (id 3)
```

- Pierwsze id ma typ Enum a => (a->a) -> (a->a)
- Drugie id ma typ Num a => a->a
- ```
  Prelude>:t id succ (id 3)
  id succ (id 3) :: (Enum a, Num a) => a
  ```
- Ograniczenia są dziedziczone z funkcji succ oraz z argumentu 3.
  ```

Outline

Wnioskowanie o typach

- Haskell ma statyczne typowanie wszystkich wyrażen.
- To znaczy, że w programowanie każde wyrażenie musi mieć określony typ.
- Haskell posiada mechanizm, który wyprowadza typ wyrażenia i typ ten jest najbardziej ogólny.
- Programista może narzucić wyrażeniu (tylko) typ bardziej szczegółowy.
- Dobrą praktyką jest deklarowanie typów wyrażeń aby zwiększyć czytelność i kontrolę nad kodem.

Wnioskowanie o typach

- Jeśli `f:: a->b` oraz `x::a`, to `f x:: b`.
- Jeśli `f::t->s` oraz `x::r`, gdzie `r` można otrzymać przez podstawienie σ z `t`, to `f x :: s`σ.
- Np. `map:: (a->b)->[a]->[b]` i `and::Bool->Bool->Bool`. Wtedy `a=Bool`, `b=(Bool->Bool)` i

```
map and :: [Bool]->[Bool->Bool]
```

- Jeśli `f:: a->b` a `g:: b->c`, to `g.f:: a->c`
- Poprzedni punkt wynika też z faktu, że `(.):: (b->c)->(a->b)->(a->c)`

Przykład

- Przykładowe typy wyrażeń
- `(++)::[a]->[a]->[a]`
- ```
  Prelude> let hello x = "Hello" ++ x
  hello::[Char]->[Char]
  ```
- ```
  Prelude> let f x@(z:zs) y = if z
                    then y++y else x++x
  f :: [Bool]->[Bool]->[Bool]
  ```

Przykład

- Jaki jest typ `uncurry uncurry`?
- Dla ułatwienia napiszmy `uncurry1 uncurry2`.
- Typ (polimorficzny) `uncurry2` to `(a->b->c)-> ((a,b)->c)`
- Możemy potraktować więc uncurry jako funkcję przyjmującą dwa argumenty typu `a->b->c` oraz `(a,b)`
- Typ `uncurry1` musi pasować do argumentu więc

```
uncurry1:: ((a->b->c)-> ((a,b)->c))->
                        ((a->b->c),(a,b))->c
```

- Po podstawieniu argumentu `uncurry2` do funkcji `uncurry1` otrzymujemy wyrażenie typu wartości funkcji `uncurry1` czyli

```
uncurry uncurry::(a->b->c,(a,b))->c
```

Koniec