Programowanie w logice i funkcyjne (Haskell), rachunek lambda (prezentacja niekompletna i przed korektą)

Konrad Zdanowski

Outline

Materiały

- Haskell 2010 – specyfikacja języka, ed. Simon Marlow, https://wiki.haskell.org/Language_and_library_specification,
- Miran Lipovača, Learn You a Haskell for Great Good,http://learnyouahaskell.com/chapters,
- Bryan O'Sullivan, Don Stewart, John Goerzen, Real World Haskell, http://book.realworldhaskell.org/read/
- Richard Bird, Thinking Functionally with Haskell, 2015,
- Alejandro Serrano Mena, Beginning Haskell, A Project-Based Approach, Apress, https://pulpit.uksw.edu.pl/go/link.springer.com~ssl/book/10.1007/978-1-4302-6251-0.

Programowanie funkcyjne – motywacje

- Paradygmat, w którym definiujemy funkcje zamiast opisywać jak je obliczyć.
- Ma jasno określone matematyczne podstawy, które pozwalają traktować funkcje Haskella jako obiekty matematyczne i ułatwiają rozumowanie o programie.
- Łatwiejsza analiza kodu i dowodzenie własności programu.

Cechy paradygmatu funkcyjnego

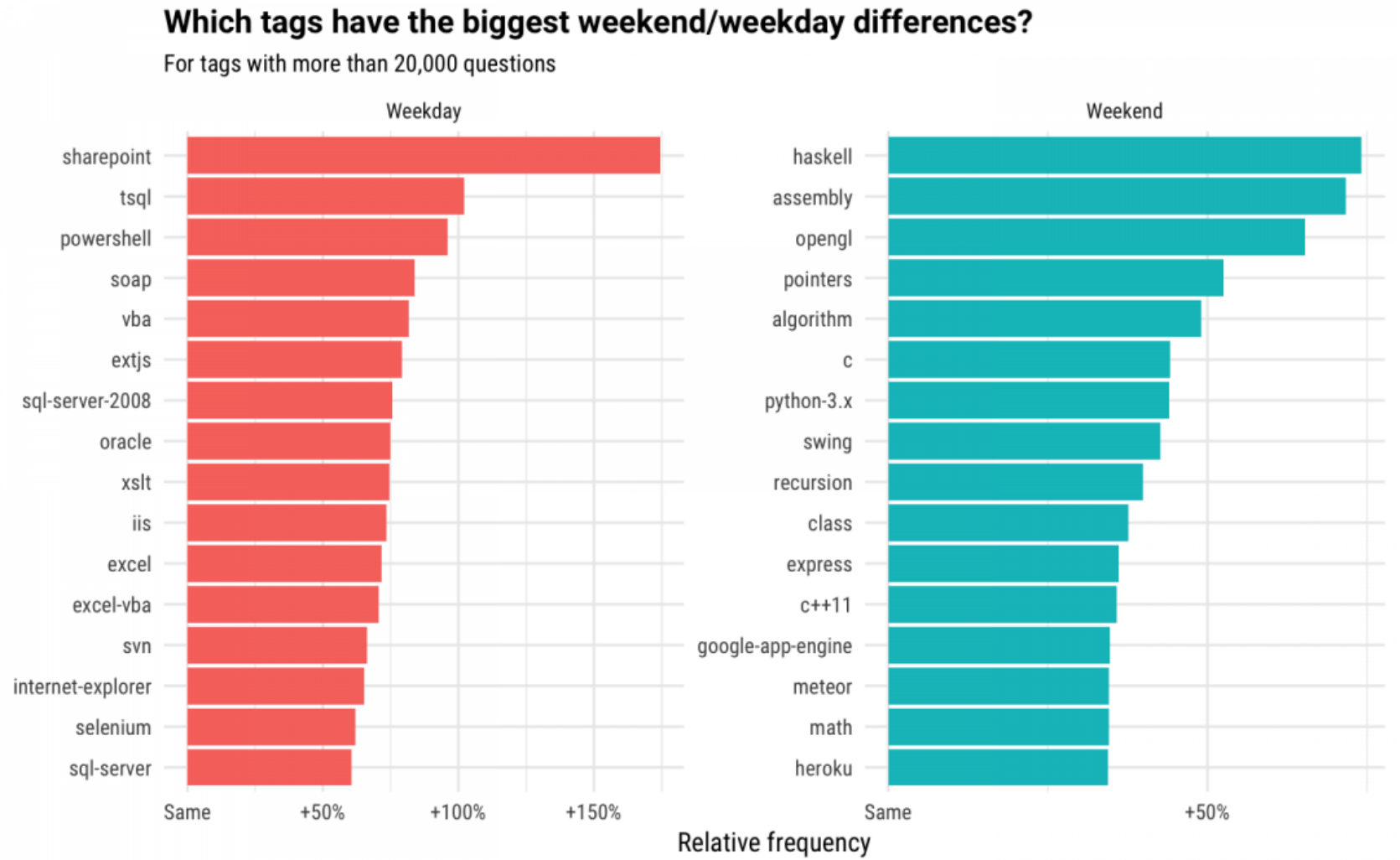
- Program w paradygmacie imperatywnym opisuje, krok po kroku, jak wykonać obliczenie.
- Na wynik obliczenia ma wpływ stan maszyny (wartość zmiennych, ...), dwukrotne wykonanie tej samej funkcji może dać różne wyniki.
- W paradygmacie funkcyjnym defniujemy funkcje, wynik funkcji jest zawsze ten sam.

Cechy paradygmatu funkcyjnego – ciąg dalszy

- Funkcje są obiektami podstawowymi, mogą występować jako argumenty, łatwo tworzy się funkcje wyższych rzędów.
- Przejrzystość referencyjna, zmienna może być zawsze zastąpiona przez swoją wartość.
- Funkcje nie mają efektów ubocznych (są „czyste”). Mogą być obliczane niezależnie od siebie, jeśli nie przekazują sobie swoich wyników. (Wyjątek: operacje wejścia i wyjścia).
- Ważnym sposobem definiowania funkcji jest rekurencja. Rekurencja zastępuje pętle.
- Leniwa ewaluacja (vs. zachłanna ewaluacja), system typów, polimorfizm, ...

Popularność Haskella

Julia Silge zbadała różnice w popularności pytań pomiędzy dniami pracującymi i weekendem na StackOverflow (7.2.2017, `https://stackoverflow.blog/2017/02/07/what-programming-languages-weekends/`)



Zastosowanie Haskella

- Zwięzły kod pozwala na pisanie szybkie pisanie prototypów i ich testowanie.
- Brak efektów ubocznych, przejrzystość referencyjna, kontrola typów, „matematyczny” styl programowania pozwala na precyzyjną analizę i kontrolę kodu, dowodzenie jego własności.
- Przykłady wykorzystania `https://wiki.haskell.org/Haskell_in_industry`

Przykład – „quicksort”

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (p:xs) = (quicksort lesser) ++ [p] ++ (quicksor
    where
        lesser = filter (< p) xs
        greater = filter (>= p) xs
```

- Oczywiście ten kod nie jest równoważny implementacji w C.
- Implementacja w C gwarantuje, że funkcja `partition` wykona się efektywnie; co więcej program w C działa na jednej tablicy.
- Kod w Haskellu nie gwarantuje (i nie daje) tej samej szybkości operacji podziału.
- Co więcej nie działamy w jednej tablicy (zmienne nie mogą zmieniać wartości).

Outline

Outline

Podstawy GHCi

- GHCi – interactive Glasgow Haskell Compiler
  - sprawdza poprawność syntaktyczną,
  - określa typy wyrażeń,
  - oblicza.
- `:load plik` ładuje plik Haskella, (po jego zmianie trzeba wykonać `:reload`.

- Program zapisujemy w pliku .hs (tradycyjne rozszerzenie).
- Biblioteki Haskella

Outline

- Nazwy funkcji zaczynają się małą literą (wyjątek: konstruktory danych).
- Z dużej litery zaczynają się nazwy typów, klasy typów, moduły.
- Funkcje zapisujemy bez nawiasów.
  - `foo1 3 4` to wynik funkcji `foo1` na argumentach 3 i 4,
  - `foo2 (3,4)` to wynik funkcji `foo2` na parze uporządkowanej (3,4),
- Funkcja ma wyższy priorytet niż operator.
  - `foo 2 + 3` to `(foo 2)+3`,
  - `foo (2+3)` to `foo 5`.
- Silniej wiąże funkcja po lewej stronie
  - `sin sin pi` nie jest dobrze zdefniowane,
  - `sin (sin pi)` jest zdefiniowane.
- Złożenie funkcji wyrażamy operatorem .
  - `(sin . cos) pi` to `sin (cos pi)`,
  - `sin . cos pi` nie jest poprawne (dlaczego?).

- Podstawowe informacje o operatorze (funkcji) możemy uzyskać przez `:info operator`
- Operator możemy zapisać w notacji funkcyjnej biorąc go w nawiasy:
  - `(+) 3 4` to `3+4`.
- Podobnie funkcję możemy zapisać jako operator przez użycie cudzysłowów:
  - ```
    ghci>let foo x y = 2*(x - y)
    ghci>foo 2 3
    5
    ghci>2 `foo` 3
    5
    ```
- Z powodów, które staną się jaśniejsze później, możemy określić część argumentów funkcji, np.
  - `(+2) 3`.
- Jednak `(-1)` to liczba -1, nie funkcja odejmująca 1. Wynika to z dwuznaczności symbolu '-'.
    ```

- Funkcje możemy też definiować stosując wyrażenia lambda.
- `\n -> n*2` to funkcja, która mnoży argument razy 2.
- `(\n -> n*2) 3` to 6.
- `(\n -> \k -> n+k` to funkcja dwuargumentowa.
- Wyrażenia lambda stosujemy, kiedy chcemy tylko raz użyć jakiejś funkcji, nie chcemy rezerwować dla niej specjalnej nazwy.

Definiowanie funkcji

- W pliku z opisem funkcji opisujemy (opcjonalnie) typ funkcji:
  ```
  pierwsza:: OpisTypu
  ```
- Opisujemy zachowanie funkcji stosując wzorce wejścia

```
pierwsza 'a' 0 = "Zwrocona wartosc"
pierwsza _ 1 = "Drugi argument to 1"
pierwsza a b = "Argumenty: "++show a
                        ++", " ++ show b
```

- Podobnie możemy definiować funkcje działające na listach (rekurencyjny typ danych), ...

Definiowanie funkcji

- Wyrażenie

```
if Warunek then Wyrazenie1 else Wyrazenie2
```

- Wyrażenie

```
case Wyrazenie of Wzor1 -> Wyrazenie1
                  Wzor2 -> Wyrazenie2
```

Definiowanie funkcji przez przypadki

- ```
  przedzialy:: Int -> String
  przedzialy x =
       | x<=0 = "Mala liczba"
       | x<=10 = "Srednia liczba"
       | x<=100 = "Ho ho ho!"
       | otherwise = "Duza liczba"
  ```
  ```

Ewaluacja wyrażeń – obliczenie Haskella

- Funkcje Haskella określone są przez defniujące je równania.
- Obliczenie Haskella polega na przepisaniu wyrażeń do najprostszej (nieredukowalnej) postaci korzystając z tych równań.
- ```
  kwadrat x = x*x

  kwadrat (3 + 4) -7
  = kwadrat (7) - 7
  = (7*7) - 7
  = 49 - 7
  = 42
  ```
  ```

Leniwa ewaluacja

- Aby obliczyć wartość wyrażenia Haskell wykonuje tylko niezbędne redukcje.
- ```
  Prelude> [0,1,2]!!3
  *** Exception: Prelude.!!: index too large
  Prelude> fst ("ala",[0,1,2]!!3)
  "ala"
  ```

- Aby obliczyć pierwszy element pary uporządkowanej nie trzeba obliczać jej drugiego elementu.
- Haskell oblicza wartość dopiero wtedy, gdy jest to konieczne.
  ```

Leniwa ewaluacja – przykład

- Nie musimy wyliczać całej listy, żeby wyliczyć jej n-ty element.
- ```
  Prelude> [1,[1,2,3]!!5, 3,4,5]
  [1,*** Exception: Prelude.!!: index too large
  Prelude> [1,[1,2,3]!!5, 3,4,5]!!4
  5
  ```
  ```

Leniwa ewaluacja – przykład

- Leniwa ewaluacja pozwala tworzyć struktury potencjalnie nieskończone.
- ```
  Prelude> let x = [1,2,3]++x
  Prelude> x!! 8
  3
  ```

- Zmienna `x` ma wartość nieskonczonej listy [1,2,3,1,2,3,1,2,3,1,...]
- Haskell oblicza wartość `x` tylko do indeksu 8.
  ```

Typy

- Każdy obiekt w Haskellu posiada swój typ.
- Możemy go sprawdzić poleceniem `:t nazwa`
- Typy wbudowane: Int, Float, Double, Char.
- Integer – liczby całkowite bez ograniczenia ich wielkości,
- Bool

Przykład

```
> let silnia n = product [1..n]
> :t silnia
silnia :: (Num a, Enum a) => a -> a
> silnia 50
30414093201713378043612608166064768844377641568960512000
> maxBound::Int
9223372036854775807
```

Typy złożone

- Możemy konstruować typy bardziej złożone z prostszych.
  - [a] – lista elementów typu a,
  - (a,b) – para uporządkowana typów a i b,
  - (a,b,c) – trójka
  - () – pusta krotka i jej typ,
  - a -> b – typ funkcji z typu a w typ b.
- Haskell potrafi sam wywnioskować typ funkcji ale dobrym zwyczajem jest podanie tego typu.

Typy – przykłady

- ```
  Prelude> :t "ala"
  "ala" :: [Char]
  Prelude> :t ['a','l', 'a']
  ['a','l', 'a'] :: [Char]
  Prelude> "ala"==['a','l','a']
  True
  ```

- n-ty element listy:

  ```
  Prelude> :t (!!)
  (!!) :: [a] -> Int -> a
  ```

- Konkatenacja `(++)::[a]->[a]->[a]`
- Złożenie funkcji `(.):: (b->c) -> (a->b)->(a->c)`
- Typy tych funkcji są polimorficzne, tzn. `a,b,c` są dowolnymi typami.
  ```

Typy – przykłady

- `(+) :: Num a => a -> a -> a`
- (+) ma typ a->a->a, dla każdego typu *numerycznego* a.
- Klasa typów zawiera zbiór funkcji, które muszą być zdefiniowane dla typów z tej klasy, ale mogą być zdefiniowane w różny sposób.

Klasy typów

- ```
  class Eq a where
    (==),(/=) :: a -> a -> Bool
    x /= y = not (x == y)
  ```

- ```
  instance Eq Bool where
     x == y = if x then y else not y
  ```

Wcięcia

- Jeśli definiujemy używając `let, do, where` i potrzebujemy więcej niż jednej linii musimy uważać na wcięcia.
- Załóżmy, że piszemy `where x = 2`
  - nowa linia zaczynjąca się za `x` to kontynuacja poprzedniej linii,
  - nowa linia zaczynająca się tam gdzie `x` to nowy element definicji,
  - nowa linia zaczynająca się przed `x` to koniec definicji.
- W pliku wszystkie definicje powinny zaczynać się w tej samej kolumnie.

Wcięcia

- Definicja używająca wcięć:

```
where r = 4
      pole = r^2*pi
      obwod = 2*pi*r
```

- Zamiast wcięć można użyć nawiasów i średników:

```
where {r = 4; pole= r^2*pi; obwod=2*pi*r}
```

Operatory (.) oraz ($)

- Czasem priorytety operatorów wymuszają używanie dużej ilości nawiasów.
- `f g h x` domyślnie jest interpretowany jako `((f g) h) x`
- Jeśli chcemy to zmienić możemy użyć ($) lub (.)
- `(f.g.h) x` to `f(g(h x))`.

Operatory (.) oraz ($)

- `ghci>:info ($)`
  `($) :: (a -> b) -> a -> b  -- Defined in ‘GHC.Base’`
  `infixr 0 $`
- Napis `f $ g $ h $ x` oznacza `f(g(h x))`
- Możemy też użyć operatora ($) aby oddzielić złożony term opisujący argument od funkcji:
  - `sin $ 2+pi`

Hello world!

- W pliku Hello_Haskell.hs zapisujemy

```
main={ putStrLn "Hello world!"}
```

- Kompilujemy `ghc Hello_Haskell.hs`
-
```
$ ./Hello_Haskell.exe
Hello world!
```

Outline

Listy

- Listy są ważną częścią Haskella.
- Jedna z ich funkcj1 i jest taka, jak w językach imperatywnych.
- Pozwalają definiować struktury nieskończone.
- Są użyteczne przy sterowaniu rekursją (w Haskellu nie ma pętli).

- Lista elementów typu a ma typ [a]
- Lista pusta to []
- Drugi konstruktor listy to operator (:): `x:xs`
  - Lista [1,2,3] to 1:(2:(3:[])).
  - Lista `"ala"` to `'a':('l':('a':[]))`
- (:) możemy wykorzystać przy dopasowywaniu wzorców:
  - `let head' (x:_) = x`
  - `let tail' (_:xs)=xs`

Definiowanie list

- `[2..6] = [2,3,4,5,6]`
- `[2..1]=[]`
- `[2,4..8]=[2,4,6,8]`
- `[1.2..5.1]=[1.2, 2.2, 3.2,4.2, 5.2]`
- `ghci> [1.2, 1.8..5.1]`
  `[1.2,1.8,2.4000000000000004,3.0,`
  `3.6000000000000005,4.2,4.800000000000001]`
- `enumFromTo, enumFromThenTo`

- take n xs, drop n xs,
- takeWhile warunek xs
  - `takeWhile (<9) [1,2..]`
- dropWhile

Listy – wyróżnianie

- Możemy obliczyć obraz listy względem funkcji:
  - `[f(x) | x <-xs]`
  - `[2*x | x<-[0..4]]`
- Wynik tej operacji możemy filtrować przez predykaty:
  - `[x^2 | x<-[1..10], rem x 3 =0]`
- Możemy użyć wielu list:

```
ghci> [ (x,y) | x<-[1..2], y<-[1..2]]
[(1,1),(1,2),(2,1),(2,2)]
```

- ```
  Prelude> :type map
  map :: (a -> b) -> [a] -> [b]

  map _ [] = []
  map f (x:xs) = (f x): map f xs
  ```

- ```
  Prelude> :type filter
  filter :: (a -> Bool) -> [a] -> [a]

  filter _ [] = []
  filter f (x:xs) = if f x then (x:tail) else (tail)
                        where tail = filter f xs
  ```

- `filter f xs` można wyrazić też przez
  `[ x | x <-xs, f x]`
- `zip, unzip`

Funkcje rekurencyjne i listy

- Konstruktor listy (:) jest naprostszym sposobem definiowania funkcji rekurencyjnych na listach.
- ```
  sum::[Int]->Int
  sum [] = 0
  sum (x:xs) = x + sum xs
  ```
- Podobnie możemy zdefiniować
  `product, length, concat, and, or, ...`
  ```

Przykład – liczby pierwsze

```
Prelude> :{
Prelude> let mySieve []=[]
Prelude|     mySieve (x:xs) =
                   filter (\y -> rem y x /= 0) xs
Prelude| :}
Prelude> :{
Prelude| let myPrimes [] = []
Prelude|     myPrimes (x:xs)= x:(myPrimes $ mySieve xs)
Prelude| :}

Prelude> take 21 (myPrimes [2..])
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71
```

Albo:

```
let myPrimes (x:xs) =
          x:(myPrimes [z| z<-xs, rem z x /= 0] )
```

Zwijanie list – folding

- ```
  Prelude> :type foldr
  foldr :: Foldable t =>
             (a -> b -> b) -> b -> t a -> b
  ```
- Teraz interesuje nas operacja zwijania w odniesieniu do list, czyli t będzie konstruktorem typu [a].
- ```
  listFoldr::(a->b->b) -> b -> [a] -> b
  listFoldr = foldr
  ```

foldr

- `foldr` możemy zdefiniować następująco:

```
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

- Obliczeniu `foldr f b xs` odpowiada następujący term:
  - `f(x1, f(x2, f(x3, ..., f(xN, f(z,[]))...),`
  - gdzie `xs =[x1,...,xN],`
  - `f(z,[])=z.`

foldr – przykłady

- `sum xs = foldr (+) 0 xs`
- `length xs = foldr (\x y -> y+1 ) 0 xs`
- `concat xs ys = foldr (\x y -> x:y) ys xs`
- ...

foldr

Rysunek porównujący foldr z listą.
Zwrócić uwagę, że główny operator pojawia się już przy x1 i czasem nie trzeba ewaluaować reszty termu.
Ma to znaczenie dla efektywności oraz w sktrukturach nieskończonych.

foldr – ewaluacja

- `foldr f b xs` może zredukować tylko część termu.
- Koniecznie jednak musi rozstrzygnąć czy xs jest listą pustą czy nie.

foldl

- ```
  foldl :: (b -> a -> b) -> b -> [a] -> b
  foldl f z [] = z
  foldl f z (x:xs) = foldl f (f z x) xs
  ```

-
  ```

foldr vs. foldl

- Przykład za Cale Gibbard (https://wiki.haskell.org/Fold#Examples)
-
```
Prelude> let f x y = concat ["f(", x, ", ", y, ")"]
Prelude> let xs= map show [1,2,3,4]
Prelude> xs
["1","2","3","4"]
Prelude> foldr f "0" xs
"f(1, f(2, f(3, f(4, 0))))"
Prelude> foldl f "0" xs
"f(f(f(f(0, 1), 2), 3), 4)"
```

foldr vs. foldl

```
Prelude> foldr (:) [] [1,2,3]
[1,2,3]
Prelude> foldl (flip (:)) [] [1,2,3]
[3,2,1]
```

Dopisac o folrd vs foldl na nieskocznej liscie (brak glownego operatora w termie z foldl).

Outline

Rachunek lambda

- Rachunek lambda to matematyczna podstawa funkcyjnych języków programowania.
- Stworzony przez Churcha w latach 30tych XIX wieku był jedną z pierwszych formalizacji pojęcia obliczalności.
- Posłużył do dowodu nierozstrzygalności logiki pierwszego rzędu.
- Podstawową zasadą semantyki rachunku lambda jest: wszystko jest funkcją.

Wszystko jest funkcją

- W rachunku lambda (RL) wartość każdego wyrażenia to funkcja.
- Język RL to:
  - operator lambda: λ,
  - operator aplikacji funkcji do argumentów (nie zapisywany),
  - zmienne,
  - symbole pomocnicze: nawiasy, kropka.
- Przykład funkcji RL: $\lambda x.x$

Lambda termy

- Lambda wyrażenia to:
  - zmienne,
  - aplikacja jednego wyrażenia do drugiego: $t_1\,t_2$ (t_2 jest wtedy argumentem funkcji t_1),
  - lambda abstrakcje: $\lambda x.t$, gdzie t jest lambda wyrażeniem (abstrakcja tworzy funkcję, której argumentem jest x a wartością t).
- Operator $\lambda x.t$ wiąże zmienną x w t.
- Lambda termy to klasy abstrakcji lambda wyrażeń relacji alpha równoważności.

Przykłady lambda termów

- Klasyczne lambda termy:
  - $\lambda x.x$ – funkcja identycznościowa (kombinator I),
  - $\lambda x \lambda y.x$ – funkcja zwracająca pierwszy argument (kombinator K),
  - $\lambda x \lambda y \lambda z.(xz)(yz)$ – kombinator S,
- W kombinatorze S, jeśli z ma typ a, a y ma typ $a \to b$, to x musi mieć typ $a \to b \to c$.

Redukcje lambda termów

- Załóżmy, że mamy dobrze zdefiniowaną operację podstawienia.
- α równoważność – utożsamiamy termu różniące się tylko nazwami zmiennych związanych (możemy zmieniać nazwy zmiennych związanych bez zmiany znaczenia termu),
- β redukcja: term postaci $(\lambda x.t(x))s$ możemy przepisac jako $t(s/x)$.
- β redukcję możemy interpretować jako podstawienie argumentu do funkcji.
- Aplikacja funkcji po lewej wiąże silniej.

Redukcje lambda termów

- Redukcje lambda termów to pewien sposób obliczeń.
- Ich kolejność nie jest określona (możemy mieć rózne możliwości β redukcji w termie).
- O obliczeniu Haskella też możemy myśleć jako o ciągu redukcji termów.

Postać normalna

- Postać normalna (β normalna) termu t to postać, w której nie możemy w termie t wykonać żadnej β redukcji.
- Piszemy $M \rightarrow_\beta N$ jeśli N można otrzymać z M przez wykonanie β redukcji.
- Piszemy $M \twoheadrightarrow_\beta N$ jeśli N można otrzymać z M przez wykonanie ciągu β redukcji.
- $=_\beta$ to najmniejsza relacja równoważności pomiędzy lambda termami zawierająca $\rightarrow_\beta$.

Postać normalna

Twierdzenie 1 (Churcha-Rossera)

Jeśli $M \twoheadrightarrow_\beta P$ i $M \twoheadrightarrow_\beta Q$, to istnieje term N, taki, że $P \twoheadrightarrow_\beta N$ i $Q \twoheadrightarrow_\beta N$.

- Z twierdzenia tego, nie wynika, że każdy term ma postać normalną.
- Można podać efektywny sposób wyboru beta redukcji taki, że dla dowolnego termu M, jeśli M ma postać normalną, to ten sposób wyboru do niej doprowadzi.

Przykłady

- Te termy są w postaci normalnej:
  - $I = \lambda x.x$
  - $K = \lambda xy.x$
  - $S = \lambda xyz.(xz)(yz)$
  - $\omega = \lambda x.xx$
- Term $\Omega = \omega\omega$ nie jest w postaci normalnej bo $\Omega \rightarrow \beta\Omega$.
- Term $Kz\Omega$ ma nieskończony ciąg redukcji $Kz\Omega \rightarrow_\beta Kz\Omega$, jeśli wykonujemy redukcje dla Ω.
- Term $Kz\Omega$ ma postać normalną bo $Kz\Omega \rightarrow_\beta z$

Kombinator punktu stałego Y

Definicja 2

Kombinator punktu stałego Y to $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$

Twierdzenie 3

Dla dowolnego termu F, $F(Y(F)) =_\beta Y(F)$.

Dowód.

$$\begin{aligned} Y(F) &=_\beta (\lambda x.F(xx))(\lambda x.F(xx)) \\ &=_\beta F((\lambda x.F(xx))(\lambda x.F(xx))) \\ &=_\beta F((\lambda f.((\lambda x.f(xx))(\lambda x.f(xx))))F) \\ &=_\beta F(Y(F)). \end{aligned}$$

□

$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$
$F(Y(F)) =_\beta Y(F)$

Wniosek 4

Dla dowolnego termu M istnieje taki term N, że $N =_\beta M(N/x)$.

Dowód.

Niech $N = Y(\lambda x.M)$.
Wtedy
$N = Y(\lambda x.M) =_\beta (\lambda x.M)(Y(\lambda x.M)) =_\beta (\lambda x.M)(N) =_\beta M(N/x)$. □

Podstawowe konstrukcje w rachunku lambda

- Wartości logiczne:
  - $true = \lambda xy.x$,
  - $false = \lambda xy.y$
- Konstrukcja warunkowa:

$$if\ P\ then\ Q\ else\ R = PQR.$$

- $if\ true\ then\ Q\ else\ R =_\beta Q$,
- $if\ false\ then\ Q\ else\ R =_\beta R$,
- Implikacja PQR posiada wartość termu Q lub R (gdy P jest wartością true lub false).
- Implikacja w Haskellu także jest termem, który posiada wartość.

Para uporządkowana

- $\langle M, N \rangle = \lambda x.xMN$,
- $\pi_i = \lambda x_1 \lambda x_2.x_i$, dla $i = 1, 2$.
- $\langle M_1, M_2 \rangle \pi_i =_\beta M_i$, dla $i = 1, 2$.

Liczby naturalne

- Liczby naturalne definiujemy przez tzw. liczebniki Churcha.
- $c_n = \lambda f \lambda x.f^n(x)$
- $c_0 = \lambda f \lambda x.x$, $c_1 = \lambda f \lambda x.fx$, $c_2 = \lambda f \lambda x.f(f(x))$, ...
- Na tych termach możemy teraz definiować funkcje arytmetyczne za pomocą lambda termów:
  - $succ = \lambda n \lambda fx.f(nfx)$,
  - $add = \lambda mn.\lambda fx.mf(nfx)$,
  - $mult = \ldots$,
  - ...
- $succc_1 =_\beta c_2$

Przykład

$$
\begin{aligned}
succ\ c_1 &= (\lambda n \lambda f x . f(nfx))c_1 \\
&=_\beta \lambda f x . f((c_1)fx) \\
&= \lambda f x . f((\lambda f \lambda x . fx)fx) \\
&= \lambda f x . f((\lambda x . fx)x) \\
&= \lambda f x . f(fx)) \\
&= c_2 .
\end{aligned}
$$

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`

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

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