Haskell – lazy evaluation

Z FITwiki

Funkcionální jazyk, který je case-sensitive. Ke všemu má ještě speciální pravidla pro první znaky literálů. Je nutné dodržovat při psaní programu tato pravidla:

- Jména typů, typových tříd a datové konstruktory musejí mít velké počáteční písmeno
- Ostatní literály musejí mít písmeno malé

Obsah

Datové typy

- Primitivní typy
  - **Int** - celá čísla (45, 890)
  - **Word** - celá čísla bez znaménka (45, 890)
  - **Integer** - celá čísla bez omezení rozsahu
  - **Char** - znaky ('a', 'X')
  - **Bool** - reprezentace logických hodnot (True, False) ... jedná se o datové konstruktory, proto mají hodnoty velké počáteční písmeno
  - **Float** - desetinná čísla (3.141, 234.8)
- Strukturované typy
  - ***Seznam*** - homogenní datová struktura, která nemá obecně žádné omezení rozměru. Konstruktorem je ":" a "[]". Kdy dvojtečka připojuje prvky do seznamu a závorky značí prázdný seznam. Zápis 1:2:3:[] je tedy konstrukce seznamu o třech prvcích. Alternativním a přirozenějším zápisem je [1,2,3]. Toto však není zcela čistý zápis v jazyce Haskell, ale překladač si ji interně přeloží na dříve zmíněnou sekvenci.
  - ***N-tice*** - heterogení datová struktura. Jejím konstruktorem je "," ve spojení s párem kulatých závorek. Například dvojice celých čísel (1,2)

Haskell je silně typovaný jazyk. Změna jednoho typu na druhý je možná pouze zavoláním převodní funkce a každá entita má přesně daný svůj typ. Pro popis typu vstupu a výstupu funkce se používá zápis (příklad konstruktoru seznamu):

```
(:) :: Int -> [Int] -> [Int]
```

Tento zápis říká asi toto: Funkce se jménem (:) jako jeden vstup bere celé číslo. Druhý vstupní parametr je seznam celých čísel. Poslední část potom říká, že výstupem bude zase seznam celých čísel. Jelikož se jedná o jazyk silně typovaný, tak aby pro každý vstup nemuselo existovat několik definicí stejného operátoru, tak se zavádějí **typové proměnné**. Ty nahrazují skutečné typy a při vyhodnocení jsou nahrazeny skutečnými typy. Zápis vypadá následovně:

```
(:) :: a -> [a] -> [a]
```

Za symbol **a** se potom na všechna místa potom přiřadí právě typ **Int** nebo třeba **Float**. Tento princip právě umožňuje vytváření seznamů jakéhokoliv typu.

Typová třída

Hlavní stránka: Haskell – typové třídy

- je druh rozhraní
- typ, který je součástí typové třídy implementuje definované chování
  - Typ a je instancí třídy C, je-li pro něj definována vymezená množina operací
  - Interně je pro každou typovou třídu slovník, který pro jednotlivé datové typy ukazuje na funkce, které implementují operace požadované tou typovou třídou.
- => značí typové omezení (např. (==) :: (Eq a) => a -> a -> Bool)
  - Například typ pro funkci elem je: "elem :: (Eq a) => a -> [a] -> Bool"
- např.:
  - Eq - typová třída pro testování rovnosti
  - Ord - typová třída podporující porovnávání
  - Show - instance této třídy může být převedena do řetězce (funkce show)

Instance typové třídy (přidání datového typu Int do třídy Eq - definicice operace požadované třídou (operace porovnání))

```
instance Eq Int where
    x==y = intEq x y
           // metoda (definice operace)
```

Instance typové třídy (přidání datového typu který staví nad existujícím typem (tady seznam) do třídy Eq - definicice operace požadované třídou (operace porovnání))

```
instance (Eq a) => Eq [a] where
     []==[] = True
     (x:xs)==(y:ys) = x==y && xs==ys
      _==_ = False
```

Definování vlastní typové třídy

Typová třída **Shape** musí mít povinně operaci **area** (obsah útvaru). Instancí typové třídy je **MyShape**, což může být kruh nebo obdelník:

```
class Shape s where
    area :: s -> Float

data MyShape = Circle Float | Rectangle Float Float

instance Shape MyShape where
    area (Circle r) = 3.145 * r
    area (Rectangle a b) = a * b
```

Funkce

Funkce **f** má pro jeden vstupní parametr definován typ jako **f :: a -> b**. Tento zápis lze předefinovat a říci buď přesný typ a nebo které typy musejí být stejné, a tak podobně.

Funkce součtu dvou prvků a druhé mocniny vypadá následovně:

```
add x y = x + y
square x = x * x
```

Práci se seznamem jako parametrem ukazuje funkce, která počítá délku seznamu:

```
length [] = 0
length (x:xs) = 1 + length xs
```

Vyhodnocení funkcí probíhá od shora dolů, proto funkce, které popisují nadmnožinu ostatních případů, musejí být uvedeny jako poslední. První funkce se ztotožní s prázdným seznamem a vrací nulu. Když ne vstupu není prázdný seznam, tak se přistoupí ke druhé funkci. Tam se vzor chápe jako seznam, který má na začátku prvek zastoupený parametrem **x** a zbytek seznamu označený **xs**. Funkce potom rekurzivně přičte jedničku k délce zbytku seznamu.

Pojmenování části vzoru

Při spojování dvou seznamů nás nezajímá pouze hlavička a zbytek seznamu. Je potřeba pracovat s jedním parametrem jako celkem, proto si ho můžeme pojmenovat(viz jména **l1** a **l2**).

```
merge [] l2 = l2
merge l1 [] = l1
merge l1@(x:xs) l2@(y:ys) =
  if x<y then x:merge xs l2 else y:merge l1 ys
```

Anonymní proměnné

Funkce vrací hlavičku seznamu. Zbytek nás nezajímá, proto je pouze naznačeno, že by tam mělo něco být, ale nemá to konkrétní název.

```
hd (x:_) = x
```

Lokální funkce

Jsou definované za klíčovým slovem **where**. Je důležité dodržovat odsazení tak jak je vidět. **where** je odsazeno od definice funkce a lokální definice by měly být odsazeny v něm.

```
sumsqr x y = xx + yy
  where
    sqr a b = a*b
    xx = sqr x x
    yy = sqr y y
```

Jinou možností zápisu lokálních funkcí je využití konstrukce **let in**.

```
sumsqr x y =
  let
    sqr a b = a*b
    xx = sqr x x
    yy = sqr y y
  in
    xx + yy
```

Akce

V imperativních jazycích jsou programy tvořeny akcemi. Na rozdíl od běžných funkcí u akcí záleží na pořadí vykonávání. Musejí být tedy v rámci monád - bloku **do** apod.

Akce je v haskellu funkce s výsledkem typu **IO a** (např. IO Integer) - odtud "typy v jazyce včetně akcí"

[1] (https://www.haskell.org/tutorial/io.html)

Líné vyhodnocení

Tato strategie vyhodnocení uvádí do absolutní perfekce strategii vyhodnocení označovanou, vyhodnocení v případě potřeby (call-by-need). Tento přístup vyhodnocuje určitý výraz pouze jednou a pouze tehdy, je li potřeba. Hodnota výrazu taky zůstává zachována pro pozdější použití.

Příkladem, na kterém se nejčastěji ukazuje tato vlastnost Haskellu, je definice nekonečného seznamu :

```
[1..]
```

Pokud by nebyl Haskell jazyk s líným vyhodnocením, tak kdekoliv by se tato konstrukce vyskytnula, tak by začal generovat pole, dokud by nedošla paměť a program neskončil chybou. Pokud ale máme správně napsaný program v Haskellu, tak není problém s takovýmto polem pracovat. V standardní knihovně existuje funkce **take**, která jako první parametr bere číslo **n** a druhý pole. Výsledkem je prvních **n** prvků pole. Vysledek použití je vidět za

```
take 5 [1..] => [1,2,3,4,5]
take 10 [-1,-3..] => [-1,-3,-5,-7,-9,-11,-13,-15,-17,-19]
```

Výsledek líného vyhodnocení může být použit přes výpočtovou sekvenci celé funkce.

Využití lazy evaluation pro efektivní vyhodnocení fibonnaciho řady se složitostí O(n)

- je to oblíbený příklad lazy evalution v Haskellu, dával ho i přímo Kolář na státnicích (zkuste si to vymyslet v těch nervech bez přípravy...)
- podrobnější popis viz. http://en.literateprograms.org/Fibonacci_numbers_(Haskell)

Naivní implementace:

```
fib :: Int -> Integer
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

S pomocí nekonečného listu a lazy evalution:

```
fib :: Int -> Integer
fib n = fibs !! n   -- Vrat n-ty prvek z pole fibs
  where
    fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Rekurze

- **Zpětná rekurze** - Pokud po návratu z rekurzivního volání probíhá ještě nějaký výpočet. f(x) = e(f(x))
- **Dopředná rekurze** - Pokud rekurzivní volání je poslední část výpočtu. f(x) = f(e(x))
- **Lineární rekurze** - Ve výpočtu rekurze je právě jedno rekurzivní volání funkce. Tento typ rekurze lze převést na cyklus.
- **Koncová rekurze** - Dopředně lineární rekurze. Každou takovouto funkci lze převést na efektivní cyklus! Není potřeba uchovávat stav nedokončených výpočtů.

Kategorie: Státnice 2016 | Státnice FPR | Funkcionální a logické programování

- Stránka byla naposledy editována 17. 6. 2016 v 12:09.