44. Haskell - lazy evaluation (typy v jazyce včetně akcí, uživatelské typy, význam typových tříd, demonstrace lazy evaluation).

Úvod k Haskellu

- vyjadřujeme, co se má spočítat
- program nemá implicitní stav

Haskell je programovací jazyk, který je

- **deklarativní**
  - nemá žádný implicitní stav, který by se měnil posloupností příkazů
  - kód tvořen výrazy, nikoli příkazy
  - opakované provádění není vyjádřeno cyklem, ale rekurzí
  - nevyjadřujeme explicitní posloupností příkazů, jak se má program vykonat, ale vyjádříme, co se má spočítat
- **čistě funkcionální**
  - jeho formální bází je λ-kalkul
  - použití matematického modelu aplikace funkcí na argumenty
    - vede deterministicky na jediný výsledek (žádné zpětné prohledávání)
- **kompilovaný**
  - překladač **ghc** (Glasgow Haskell Compiler)

Základní typy v jazyce Haskell

V jazyce Haskell je možné použít následující bázové/odvozené typy:

- **Int**
  - celá čísla
  - rozsah závislý na architektuře stroje, který typ používá
  - **příklad**
    - 10
    - (-801::Int)
  - napíšeme-li pouze -801, překladač neví jistě, že jde o typ Int
    - chceme-li explicitně vyjádřit, že jde o typ Int, je třeba napsat **(-801::Int)**
    - jinak může jít o libovolný číselný typ podporující zápis -801
- **Integer**
  - celá čísla
  - libovolně velká dle potřeby a dle paměti stroje
- **Float**
  - desetinná čísla

- **příklad**
  - 2.72
  - (16.1::Float)

- **Char**
  - jednoduché znaky
  - **příklad**
    - 'O'
    - 'u'

type String = [Char]
String je v Haskellu typové synonymum

- **Bool**
  - pravdivostní hodnoty
  - **příklad**
    - True
    - False
  - je třeba psát s velkým počátečním písmenem

- **seznam**
  - Vestavěný homogenní (stejnorodý) strukturovaný datový typ
  - **konstruktory**
    - [ ]
      - konstruktor pro prázdný seznam
      - typ [ ] :: [a]
        - *a* je typová proměnná
        - prázdný seznam může existovat nad libovolným typem
        - celý seznam má pouze jediný typ položek *a*, ovšem ten je libovolný
    - : *(dvojtečka, cons)*
      - konstruktor pro neprázdný seznam
      - v kontextu seznamu s typem *a* píšeme
        - hodnotu typu *a* vlevo od :
        - seznam typu [a] vpravo od :
      - **příklad**
        - 1:[ ]
          - seznam [1]
          - seznam typu [Int]
        - 'F' : 'L' : 'P' : [ ]
          - seznam ['F', 'L', 'P']
          - seznam typu [Char]
        - (True:[]):(False:False:[]):[]
          - seznam [[True],[False,False]]
          - seznam typu [[Bool]]
        - pro zjednodušení lze rovnou psát
          - [1]
          - ['F', 'L', 'P']
          - [[True],[False,False]]

- obecně se používá zápis **(x:xs)**
  - **x** je první prvek seznamu
  - **xs** je zbytek seznamu
  - závorky se používají kvůli prioritě

- **ntice**
  - vestavěný různorodý strukturovaný datový typ
  - **konstruktor**
    - , *(čárka)*
      - konstruktor pro ntici
      - může obsahovat různorodé typy jako elementy
      - **příklad**
        - (1, 2, 'Q')
          - typ (Int, Int, Char)
        - ([ ], (True, False), 9)
          - typ ([a], (Bool, Bool), Int)
      - kvůli nízké prioritě čárky je nutné doplnit o závorky

- **funkce**
  - každá funkce je typu $a \rightarrow b$
    - a, b jsou typové proměnné
  - přijímá jeden argument, vrací jeden argument
  - funkci více proměnných lze realizovat tak, že typová proměnná b značí opět funkci
  - **příklad**
    - $(+) :: Num\ a \Rightarrow a \rightarrow a \rightarrow a$ (totéž jako $(+) :: a \rightarrow (a \rightarrow a)$)
      - funkce $(+)$ přijímá jeden argument typu a a vrací funkci typu $a \rightarrow a$
      - vrácená funkce typu $a \rightarrow a$ přijímá jeden argument typu a a vrací prvek typu a
      - výsledkem standardně implementované funkce sčítání je součet argumentu funkce $(+)$ a argumentu funkce, kterou funkce $(+)$ vrací
      - $(+)1\ 2$
        - vrátí výsledek 3()
      - *pozn.: V Pythonu by se něco takového implementovalo následovně:*
        - *def plus(arg1):*
          - *def inner(arg2):*
            - *return arg1 + arg2*
          - *return inner*
      - *následně lze provést:*
        - *plus(1)(2)*
          - *výsledek je 3*
        - *plus(1)*
          - *výsledek je <function plus.<locals>.inner at 0x7f984030e2a0>*
          - *tedy funkce přičítající 1 ke svému argumentu*

- $head :: [a] \rightarrow a$
  - funkce $head$ přijímá jeden argument typu $[a]$ (seznam libovolného typu) a vrací prvek typu a (první element seznamu)
  - $head\ [16, 32, 118]$
    - vrátí výsledek 16

Uživatelské typy v jazyce Haskell

Vytváření uživatelských typů v jazyce Haskell je možné následujícími způsoby:

- **typová synonyma**
  - jednoduché přejmenování (alias) existujícího typu
  - můžeme v kódu zaměňovat typová synonyma za jejich původní typ a naopak
  - pomocí klíčového slova **type** a uvedení nového jména
  - **syntaxe**
    - **type** *nové_jméno nepovinné_typové_proměnné = původní_typ*
  - **příklad**
    - **type** Digram = (Char, Char)
      - dva symboly
    - **type** Nibble = (Bool, Bool, Bool, Bool)
      - čtyřbitová hodnota
    - type Matrix a = [[a]]
      - matice jako seznam seznamů libovolného typu
      - *a* je zde typová proměnná, která může zastupovat libovolný typ
- **jednoduché typy**
  - vytvoření nového typu s právě jedním typovým konstruktorem
  - typový konstruktor musí být vždy uveden při vytváření instance daného typu, díky němu je zřejmé, jaký typ vytváříme
  - pomocí klíčového slova **newtype**, uvedení konstruktoru a nového jména
  - **syntaxe**
    - **newtype** *nové_jméno nepovinné_typové_proměnné = konstruktor původní_typ*
  - **příklad**
    - **newtype** SystemOfLinearEquations = SLE (Matrix Float, [Char], [Float])
      - matice soustavy **A**, vektor neznámých **x**, vektor absolutních členů **b**
        - **Ax = b**
      - pro vytvoření tohoto typu je třeba využít konstruktor SLE
        - SLE ([[1, 0], [0, 1]], ['x', 'y'], [5, 6])
      - *jen ilustrační příklad, kontrolovalo by se zvlášť, zda jde o validní soustavu*
    - **newtype** ContextFreeRule = CFRule (Char, [Char])
      - pravidlo bezkontextové gramatiky
      - pro vytvoření tohoto typu je třeba využít konstruktor CFRule
        - CFRule ('S', "aSb")

- **newtype** BinaryRelation a b = BinRel [(a, b)]
  - binární relace mezi množinami typu *a* a typu *b*
  - pro vytvoření tohoto typu je třeba využít konstruktor BinRel
    - BinRel [(2, 'a'), (131, 'd')]

- **komplexní datové typy**
  - složitější datové typy pracující s vyšším množstvím konstruktorů
  - výčtový typ, rekurzivní datové typy, …
  - při vytváření instance tohoto typu musí být vždy uveden některý z konstruktorů
  - konstruktory v definici typu musí být vzájemně různé
  - pomocí klíčového typu **data**, uvedení konstruktorů, nového jména, nepovinných typových proměnných a nepovinných argumentů
  - **syntaxe**
    - **data** *jméno nepovinné_typové_proměnné = konstruktor_1 argumenty_1 | … | konstruktor_n argumenty_n*
  - **příklad**
    - **data** Pseudobool = Truth | Lie | Uncertainty
      - obyčejný **výčtový typ**
      - tři různé konstruktory (Truth, Lie, Uncertainty) bez argumentů
        - *Truth* je element typu *Pseudobool*
    - **data** TestResult = Pass | Error String
      - **rozšířený typ**
      - dva různé konstruktory, jeden z nich má argument typu String
      - například pro předání chybové zprávy typu String v případě použití *Error*
        - *Pass* je element typu *TestResult*
        - *Error "Unknown symbol"* je element typu *TestResult*
    - **data** BinaryTree k v = EmptyTree | Node k v (BinaryTree k v) (BinaryTree k v)
      - **rekurzivní typ**
      - binární vyhledávací strom s klíčem a hodnotou
      - dva různé konstruktory, jeden nemá žádné argumenty, druhý má jako argument klíč typu *k*, hodnotu typu *v* a další dva binární stromy jako levého a pravého potomka
        - *EmptyTree* je element typu *BinaryTree k v*
          - prázdný strom
        - *Node 10 20 EmptyTree EmptyTree*
          - strom, který má pouze kořen s klíčem 10 a hodnotou 20
        - *Node 10 20 (Node 2 0 EmptyTree EmptyTree) EmptyTree*
          - strom, který má kořen s klíčem 10 a hodnotou 20 a levého následníka s klíčem 2 a hodnotou 0

data Color = Red | Blue | Green

data Person = Person Int String String

getName (Person _ name _) = name

String není vestavěným typem, ale synonymem k [Char]

data Color = Red | Blue | Green deriving Eq
deriving (Eq, Ord)

instance Eq Color where
Red == Red = True
Blue == Blue = True
Green == Green = True
_ == _ = False

Typové třídy

Typová třída je rozhraní definující chování. Do typových tříd spadají jednotlivé datové typy. Pokud je typ součástí typové třídy, pak zahrnuje chování a implementaci určitého chování, které typová třída definuje. Pokud je **T** typovou třídou a **a** je instancí této typové třídy, pak je pro typ **a** definovaná a implementovaná určitá množina operací daná právě třídou **T**.

Pokud vytváříme vlastní typ spadající pod určitou typovou třídu, je třeba implementovat určité operace, ovšem výsledkem je kompatibilita s řadou knihovních funkcí, které s danou typovou třídou pracují.

Typovou třídou jsou například třídy:

- **Eq**
  - typová třída definující rovnost a nerovnost
  - **Eq a** vyžaduje implementaci operací
    - $(==) :: a \to a \to Bool$
    - $(/=) :: a \to a \to Bool$
  - minimálně je třeba implementovat operaci (==), operace (/=) z ní může být odvozená
  - **příklad**
    - definujeme funkci $eqheads$, která pro dva seznamy rozhodne, zda je jejich první element stejný
    - $eqheads :: Eq\ a \Rightarrow [a] \to [a] \to Bool$
    - $eqheads\ (x{:}xs)\ (y{:}ys) = x == y$
    - $eqheads\ \_\ \_ = False$
    - v definici $eqheads$ můžeme využít operátor ==, neboť máme jistotu, že pro typ a bude tato operace implementovaná
      - u signatury funkce jsme pomocí zápisu $Eq\ a$ => stanovili, že typ a může být jakýkoliv (ale shodný pro oba seznamy), ale musí být instancí typové třídy Eq
      - musí mít tedy naimplementovanou operaci ==
- **Ord**
  - typová třída definující relační operátory
  - **Ord a** vyžaduje kupříkladu implementaci operací
    - $(<) :: a \to a \to Bool$
    - $(>) :: a \to a \to Bool$
    - $(<=) :: a \to a \to Bool$
    - $(>=) :: a \to a \to Bool$
    - $compare :: a \to a \to Ordering$
      - $Ordering$ je výčtový typ s konstruktory LT, GT, EQ
      - $compare\ 10\ 20$ vrací LT
      - $compare\ 'b'\ 'a'$ vrací GT
      - $compare\ True\ True$ vrací EQ
    - a jiné

- minimálně je třeba implementovat operaci $compare$, ostatní z ní mohou být odvozené
- **příklad**
  - definujeme funkci $ascending$, která pro zadaný seznam zjistí, zda je seřazený vzestupně
  - $ascending :: Ord\ a \Rightarrow [a] \rightarrow Bool$
  - $ascending\ [] = True$
  - $ascending\ [x] = True$
  - $ascending\ (x : y : ys) = x <= y\ \&\&\ ascending\ (y : ys)$
  - v definici $ascending$ můžeme využít operátor <=, neboť máme jistotu, že pro typ a bude tato operace implementovaná
    - u signatury funkce jsme pomocí zápisu $Ord\ a$ stanovili, že typ a může být jakýkoliv, ale musí být instancí typové třídy Ord
    - musí mít tedy naimplementovanou operaci <=

- **Num**
  - typová třída definující čísla
  - **Num a** vyžaduje implementaci operací
    - $(+) :: a \rightarrow a \rightarrow a$
    - $(-) :: a \rightarrow a \rightarrow a$
    - $(*) :: a \rightarrow a \rightarrow a$
    - $negate :: a \rightarrow a$
    - $abs :: a \rightarrow a$
    - $signum :: a \rightarrow a$
    - $fromInteger :: a \rightarrow a$
  - minimálně je třeba implementovat všechny tyto operace, výjimkou jsou operace $(-)$ a $negate$, z nichž stačí implementovat pouze jednu

Při vytváření nového datového typu je tedy možné doimplementovat operace daných typových tříd, aby do dané typové třídy nový typ spadal.

- **syntaxe**
  - **instance** *typová_třída typ = definice*
- **příklad**
  - datový typ pro číselná znaménka
  - $data\ Sign\ =\ Pos \mid Zero \mid Neg$
  - v tuto chvíli není možné provádět porovnání na rovnost
    - $Pos\ ==\ Pos$ je nedefinované
  - definujeme nutné operace pro typovou třídu Eq
    - $instance\ Eq\ Sign\ where$

$$Pos\ ==\ Pos\ =\ True$$
$$Zero\ ==\ Zero\ =\ True$$
$$Neg\ ==\ Neg\ =\ True$$
$$\_\ ==\ \_\ =\ False$$

  - nyní je možné provádět porovnání na rovnost
    - $Pos\ ==\ Pos$ vrací $True$

        - typ $Sign$ nyní spadá pod typovou třídu Eq
    - není však možné provádět relační porovnávání typu větší/menší
        - $Zero > Neg$ je nedefinované
    - definujeme nutné operace pro typovou třídu Ord
        - $instance\ Ord\ Sign\ where$

$$compare\ Pos\ Pos = EQ$$
$$compare\ Zero\ Zero = EQ$$
$$compare\ Neg\ Neg = EQ$$
$$compare\ Pos\ Zero = GT$$
$$compare\ Pos\ Neg = GT$$
$$compare\ Zero\ Neg = GT$$
$$compare\ \_\ \_ = LT$$

    - nyní je možné používat relační operátory
        - $Zero > Neg$ vrací $True$
        - typ $Sign$ nyní spadá pod typovou třídu Ord
    - není však možné provádět aritmetické operace
        - $Pos * Neg$ je nedefinované

Pro některé datové typy je možné použít klíčové slovo **deriving** s názvem dané typové třídy a odvodit generické funkce potřebné pro danou typovou třídu bez nutnosti jejich implementace.

- **syntaxe**
    - *definice_typu deriving (seznam_typových_tříd)*
- **příklad**
    - datový typ pro číselná znaménka
    - $data\ Sign = Pos \mid Zero \mid Neg\ \ deriving\ Eq$
    - nyní je možné provádět porovnání na rovnost
        - $Pos == Pos$ vrací $True$
        - typ $Sign$ nyní spadá pod typovou třídu Eq
        - nebylo nutné operaci == přímo implementovat

Akce v Haskellu

V imperativních programovacích jazycích je program tvořen posloupností **akcí** (práce s proměnnými, se soubory apod.). Jádro Haskellu je však čistě funkcionální a akce (např. Vstupně-výstupní operace) jsou od funkcionálního jádra odděleny pomocí **monadických operátorů**. Pojmem **akce** v Haskellu budeme označovat takové operace, jež mají výsledek typu **IO a**.

- **příklad**
    - $getChar :: IO\ Char$
        - akce, jež vrací jeden znak
    - $putChar :: Char \rightarrow IO\ ()$
        - akce, jež nevrací nic
        - tiskne jediný symbol
        - () je takzvaná datová jednotka (prázdná ntice, nulatice)
    - $putStrLn :: String \rightarrow IO\ ()$

        - akce, jež nevrací nic
        - tiskne řetězec s odřádkováním
    - $getLine :: IO\ String$
        - akce, jež vrací řetězec
    - $return :: a \rightarrow IO\ a$
        - zabalí element libovolného typu do monadického typu IO
    - $main :: IO\ ()$
        - akce, jež nevrací nic
    - mnohé jiné
- **ukázka použití**
    - ```
      import Data.Char
      main :: IO ()
      main = do
          putStrLn "Zadejte znak:"
          s <- getChar
          putChar (toUpper s)
      ```
        - tato akce
            - vytiskne řetězec *Zadejte znak:* s odřádkováním
            - načte symbol
            - převede malý symbol na velký a vytiskne jej
        - využívá *do* notaci, která není součástí čistého jazyka Haskell, je nejprve přeložena na ekvivalentní haskellovský zápis
    - ```
      import Data.Char
      main :: IO ()
      main = putStrLn "Zadejte znak:" >> getChar >>= (\x -> putChar $ toUpper x)
      ```
        - tato akce
            - je ekvivalentní s výše uvedenou akcí
            - nepoužívá *do* notaci
        - operátor >> zde vynucuje pořadí provedení akcí
        - operátor >>= navíc vrací výsledek akce vlevo jako argument funkce vpravo

Líné vyhodnocování

Napříč programovacími jazyky se používají různé vyhodnocovací strategie pro vyhodnocování výrazů. Klasickými představiteli těchto strategií je líné (odložené) a striktní vyhodnocování:

- **striktní vyhodnocování**
    - argumenty funkce jsou vždy zcela vyhodnoceny
    - k jejich vyhodnocení dochází před zavoláním funkce
    - funkce pracuje s vyhodnocenými argumenty
- **líné vyhodnocování**
    - argumenty funkce nemusí být vyhodnoceny vůbec

- k vyhodnocení argumentů dochází teprve tehdy, jakmile odpovídající hodnotu vyžaduje tělo funkce
- možnost práce s nekonečnými datovými strukturami bez jejich explicitního vyhodnocení
- možný efektivnější výpočet díky neprovádění zbytečných vyhodnocení

Demonstrujme nyní různé úseky kódu, v nichž se projeví líné vyhodnocování:

- **práce s nekonečnými datovými strukturami**
  - *head [1..]*
    - výsledek: *1*
    - nalezení prvního elementu v nekonečném, vzestupně seřazeném seznamu kladných přirozených čísel
    - zápis *[1..]* značí celý nekonečný seznam, který ovšem není v paměti celý rozbalen a vyhodnocen, dochází pouze k nalezení prvního elementu
  - *(\(x:y:ys) -> y) [1..]*
    - výsledek: *2*
    - podobný případ, nalezení druhého elementu v nekonečném seznamu
  - *head $ map odd [1..]*
    - výsledek: *True*
    - aplikace funkce určující lichost čísla na nekonečný seznam, nalezení prvního elementu v seznamu takových příznaků lichosti
    - funkce *odd* není aplikována na veškeré elementy seznamu, neboť všechny takové výsledky nejsou potřeba
    - funkce *odd* je aplikována pouze na první element seznamu *[1..]*
  - *filter (\x -> x > 200000) [1..] !! 2*
    - výsledek: *200003*
    - vybere z nekonečného seznamu *[1..]* čísla vyšší než *200000* a vrátí element na indexu 2 takového seznamu
    - celý seznam není vyhodnocen, stačí jen najít element na indexu 2 a nevytvářet zbytek seznamu v paměti
  - *last [1..]*
    - výpočet **cyklí**
    - zde líné vyhodnocování cyklení nezabrání

- **zefektivnění výpočtu**
  - *(\x y -> x) 1 (10^10^100)*
    - *výsledek: 1*
    - není třeba v paměti vyhodnocovat hodnotu googolplex, která bude ihned zahozena a nepoužita
    - výpočet je velmi rychlý
  - *(\x y -> y) 1 (10^10^100)*
    - *výsledek: googolplex*
    - zde líné vyhodnocování nezabrání výpočtu hodnoty googolplex, neboť funkce vyžaduje její vyhodnocení

- **práce se souborem**
  - *fileLength f = do*
    h <- openFile f ReadMode
    c <- hGetContents h
    let l = length $ lines c
    putStrLn $ show l
    hClose h
    - tato akce provede
      - otevření souboru
      - načtení obsahu souboru
      - výpočet počtu řádků souboru, tisk tohoto čísla
      - zavření souboru
    - hodnota *l* (počet řádků souboru) je vyhodnocena teprve tehdy, jakmile její hodnotu vyžaduje řádek *putStrLn $ show l* pro výpis
  - *fileLength f = do*
    h <- openFile f ReadMode
    c <- hGetContents h
    let l = length $ lines c
    hClose h
    putStrLn $ show l
    - tato akce vznikla záměnou posledních dvou řádků u předchozí akce
    - **tato akce vyvolá výjimku i pro validní soubor na vstupu**
    - při provádění *h <- openFile f ReadMode* je poznamenáno do paměti, že pracujeme s položkou *h*, která může být potenciálně v budoucnu vyhodnocena, bude-li potřeba
    - při provádění akce *hClose h* je položka *h* zcela zahozena z paměti
    - hodnota *l* je vyhodnocována líně, je vyhodnocována teprve v rámci řádku *putStrLn $ show l*, v ten okamžik ale již *h* neexistuje a hodnota *l* vyhodnocena být nemůže, není možné nijak přečíst obsah souboru
      - vyvolání výjimky
    - *Pozn.: Je třeba si uvědomit, že* do *notace je jen syntaktickým cukrem pro operátory >>, >>=*

Pokud v textu najdete chybu, nebudete něčemu rozumět nebo budete mít dojem, že by bylo vhodné něco doplnit, kontaktujte na discordu uživatele kocotom.

FLP - Haskell - lazy evaluation

Haskel je deklarativní funkcionální jazyk

- vyjadřujeme, co se má spočítat
- program nemá implicitní stav