FLP – první cvičení
Haskell: základy, funkce vyššího řádu

S. Židek, P. Matula, M. Kidoň, L. Škarvada

ÚIFS FIT VUT

Funkcionální a logické programování, 2019/2020

Rozvrh na semestr

- pondělí od 18:00 a čtvrtek od 17:00 a od 19:00 v N204+N205
- cvičíme ve čtrnáctidenních blocích, sudý/lichý týden
- možná náhrada s jinou skupinou, ale jen ve stejném bloku
- Haskell: 1. až 3. cvičení, RNDr. Libor Škarvada
- Prolog: 4. až 6. cvičení, Ing. Lukáš Zobal

Téma	**sudý týden**		**lichý týden**	
Haskell: základy, fce vyššího řádu	3. 2.	6. 2.	10. 2.	13. 2.
Haskell: datové typy, monády	17. 2.	20. 2.	24. 2.	27. 2.
Haskell: řešení složitějších úloh	2. 3.	5. 3.	9. 3.	12. 3.
Prolog: základy	16. 3.	19. 3.	23. 3.	26. 3.
Prolog: dynamické predikáty	30. 3.	2. 4.	6. 4.	9. 4.
Prolog: praktické příklady		16. 4.	20. 4.	23. 4.

Bodování

- Ve cvičení je možno získat bonusové body.
- Jde o několik bodů za cvičení pro nejaktivnější studenty za
  - aktivní přístup k problematice,
  - rychlé či kvalitní vyřešení některých úloh,
- Nejvýše jeden bod pro studenta za jedno cvičení.
- O udělení bodu rozhoduje cvičící.
  - pokud dostáváte bod a patříte do jiné skupiny, upozorněte na to cvičícího

Projekty

- Projekt určený pro jednoho řešitele
- Zadání ve WISu, většinou algoritmy probírané v TIN
- Také možnost vlastního zadání
- Body: FP 12 bodů, LP 8 bodů
- Formát řešení: viz pokyny ve WIS
- Odevzdat: FP do 5. 4. 2020, LP do 26. 4. 2020
- Konzultace osobně, emailem, v diskusním fóru.
  (C235, `iskarvada@fit.vut.cz`)

Motivace pro studium funkcionálního programování

- \+ Přínosy:
  - Pochopení čistě funkcionálních principů z vás udělá lepšího programátora, nezávisle na paradigmatu.
  - Přináší přesnější pohled na řešené problémy.
  - Ke znalosti principů FP se často přihlíží i u pohovorů v zaměstnání
- − Ale pozor na úskalí:
  - Strmá učicí křivka

    Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. — Edsger W. Dijkstra, 1930–2002
  - Při studiu je často třeba zbavit se zaujatosti jiným paradigmatem: o problémech a algoritmech je nutno přemýšlet jinak, globálně, nově.

Haskell

https://www.haskell.org/

Vlastnosti

- deklarativní paradigma
- čistě funkcionální
  - funkce jsou hodnoty, *first-class citizens*
  - hlavní syntaktickou kategorií je výraz
  - referenčně transparentní
  - vedlejší účinky bezpečně zabalené do monadických hodnot (`IO`)
- silně staticky typovaný
  - typy jsou pro programování v Haskellu zásadní
  - parametrický polymorfismus
  - typové třídy, druhy (typy typů)
  - automatická typová inference
- nestriktní (líné) vyhodnocování
  - nekonečné datové struktury
- velmi čistý jazyk
  - minimální *core language* (většina syntaxe je jen *cukr*)
  - ortogonální architektura

Odkud se Haskell učit

- Motivační článek o FP všeobecně, s řadou příkladů
  - John Hughes: Why functional programming matters
- Základy FP a Haskellu
  - Přednášky a cvičení
  - Miran Lipovača: Learn You a Haskell for Great Good!
  - C. Allen, J. Moronuki: Haskell Programming From First Principles
  - Richard Bird: Thinking functionally with Haskell
  - Simon Thompson: Haskell – The Craft of Functional Programming
  - O'Sullivan, Stewart, Goerzen: Real World Haskell
  - Reading Simple Haskell, Writing Simple Haskell
  - Monday morning Haskell

Kompilátor a interpret

- GHC – Glasgow Haskell Compiler (Linux, Mac, Windows)
- WinHugs – interpret pro Windows
- GHCi – interaktivní interpret

Pracovní cyklus při práci s interpretem

Dvě okna: editor se zdrojovým souborem + terminál s interpretem

1. editace
2. uložení v editoru
3. nové načtení v interpretu (`:r`)
4. výpočet v interpretu

Některé editory (atom, emacs, vim) mají textové vývojové prostředí pro automatizaci těchto kroků.

Obsluha GHCi, WinHugs

Na vstup interpretu zapisujeme haskellové výrazy, případně speciální *povely* pro interpret. Povely interpretu začínají dvojtečkou, většinou stačí psát první písmeno.

- `:?` – nápověda
- `:reload` – znovunačtení
- `:type ...` – zobrazení typu výrazu
- `:info ...` – popis objektu
- `:set +s, +t, ...` – různá nastavení
- `:quit` – ukončení

Syntax I

- Komentáře:
  - `... -- komentar do konce radku`
  - `... {- blokovy komentar -} ...`
- Definice
  - hodnot, typů, typových tříd
  - globální / lokální
  - definice hodnot podle vzoru
- Pozor na odsazení – je významné a označuje úroveň zanoření
- Jméno proměnné, hodnoty, funkce
  - začíná malým písmenem, dále písmena, číslice, apostrofy, podtržítka.
  - `foo`, `x'`, `accessRights`, `access_rights`
- Jméno typu začíná velkým písmenem.
  - **`Bool`**, **`Foo'`**, **`InputType`**, **`Input_type`**
- Velkým písmenem začínají také jména datových konstruktorů (**`False`**, **`True`**, **`Nothing`**, **`Right`**), typových tříd (**`Eq`**, **`Show`**, **`Monad`**) a modulů (**`Main`**)

Definice funkcí

- Funkce:

```
odmocnina :: Double -> Double
odmocnina = sqrt

soucet :: Integer -> Integer -> Integer
soucet x y = x + y

dropString :: Int -> String -> String
dropString 0 str = str      -- definice podle vzoru
dropString n (x:xs) = dropString (n-1) xs
```

- Aplikace funkce:

```
odmocnina pi
soucet 21 34
dropString 7 "flp je skvele"
```

Definice podle vzoru

Na levé straně definice mohou být vzory

Vzor je výraz, který může obsahovat pouze

- datové konstruktory
  - konstanty `0`, `'a'`, `()`, `[]`, `Nothing`, `True`
  - konstruktory vyšší arity `Left`, `Just`, `(:)`, `(,)`
- proměnné
  - pojmenované (jen jeden výskyt ve vzoru)
  - anonymní `_`

Vzor může být na místě parametru funkce

```
faktorial 0 = 1
```

i na místě definované hodnoty

```
(a, b) = break even [1,3,5,6,7,9]
```

Typy

- Datové typy a typy funkcí
- Typy funkcí
  - `Char -> Int`
    s argumentem typu `Char` a výsledkem `Int`
  - `Int -> String -> Char`
    s argumenty typů `Int`, `String` a výsledkem typu `Char`
- Hodnoty datových typů lze vytvářet
  - `1 :: Integer`
  - `'c' :: Char`
  - `(1, 'a') :: (Integer, Char)`
- a rozebírat (v definicích podle vzoru)
  - `soucetSlozekDvojice (a, b) = a + b`
  - `(a, b) = break isSpace "Hello world"`

Příklady typů – datové typy

Hodnoty	Typy	
`-42, 2^31, 2^40287`	`Int, Integer`	celá čísla
`2.718281828459045`	`Float, Double`	desetinná čísla
`'a', '@', 'ř'`	`Char`	znaky
`"cvika FLP"`	`String`	řetězce
`()`	`()`	triviální typ
`False, True`	`Bool`	log. hodnoty
`('a',97), ('*',42)`	`(Char, Int)`	kart. součin
`Left 6, Right '@'`	`Either Int Char`	disj. sjednocení
`[8], [2,3,5,7], [0..]`	`[Integer]`	seznamy

Příklady typů – typy funkcí

Hodnoty	Typ	
`not`	`Bool -> Bool`	unární funkce
`toLower`	`Char -> Char`	unární funkce
`isSpace`	`Char -> Bool`	predikáty
`(||), (&&)`	`Bool -> Bool -> Bool`	„binární“ funkce
`encode`	`(Char->Char) -> String -> String`	vyšší funkce

Parametrický polymorfismus

Typové proměnné v typových výrazech zastupují *libovolný typ*.
Typové proměnné jsou implicitně univerzálně kvantifikovány: pro *každý* typ platí, že ho můžeme za typovou proměnnou dosadit.

```
id :: a -> a ≡ forall a. a -> a
```

Zjistěte typy funkcí (pomocí povelu `:t` pro ghci):

`id`, `const`, `flip`, `($)`, `(.)`

Typové třídy omezují množinu všech typů na nějakou její podmnožinu.
Například `(==) :: Eq a => a -> a -> Bool`
říká, že porovnávat relačním operátorem `(==)` lze jen dvě hodnoty takových typů, které jsou instancemi (leží v) typové třídy **`Eq`**.

Zjistěte typy funkcí:

`(+)`, `(/)`, `max`, `gcd`, `(^)`, `log`

Aritmetické funkce – infixový a prefixový zápis

Zápis	Význam
`2*3+2^5`	binární operátory
`(+) ((*) 2 3) ((^) 2 5)`	binární operátory (prefixový zápis)
`div 60 5`	binární funkce
`60 `div` 5`	infixový zápis funkce

Priority operátorů

Priority a implicitní směry sdružování některých operátorů.
V interpretu zjistíme povelem `:i`

←	9	`.`	skládání funkcí
←	8	`^`	umocňování
→	7	`* / `div` `mod``	multiplikativní aritmetické operátory
→	6	`+ -`	aditivní aritmetické operátory
←	5	`: ++`	přidání prvku, spojení seznamů
	4	`== /= < <= > >=`	relační operátory
←	3	`&&`	logická konjunkce
←	2	`||`	logická disjunkce

Operátorové sekce

Způsob umožňující udělat z binárního operátoru (pomocí tzv. částečné aplikace) unární funkci.

Syntax:

- levá sekce (*operand operátor*)
- pravá sekce (*operátor operand*)

`(1+)`	inkrement o 1
`(1/)`	převrácená hodnota
`(^2)`	druhá mocnina
`(2^)`	exponenciální funkce
`(>0)`	predikát kladnosti

$$(c \oplus) \equiv \lambda x \to c \oplus x$$
$$(\oplus c) \equiv \lambda x \to x \oplus c$$

Faktoriál

Rekurzivní definice:

```
faktorial :: Integer -> Integer
faktorial 0 = 1
faktorial n = n * faktorial (n-1)
```

Kratčeji a přehledněji (je to ekvivalentní?)

```
faktorialP :: Integer -> Integer
faktorialP n = product [1 .. n]
```

Vyzkoušejte:

```
faktorialP 20
faktorial 20
faktorialP (-1)
faktorial (-1)
```

Podmíněný výraz

`if` podmínka `then` výraz$_1$ `else` výraz$_2$

Podmínka – výraz typu `Bool`
Uspěje-li podmínka, celý výraz je roven výrazu$_1$, jinak je roven výrazu$_2$.

Obě větve then/else jsou povinné.
Výrazy v obou větvích musí mít stejný typ.

Ošetření chyb

Parciální funkce **`error`**

```
odmocnina :: Double -> Double
odmocnina x =
  if x < 0
    then error "Zaporne cislo."
    else sqrt x
```

```
error :: String -> a
```

Co plyne z takovéto signatury?

Jaký je návratový typ?

Která (jediná) hodnota má tento polymorfní typ?

Ošetření chyb

Parciální funkce bez ošetření

```
arcsin :: Double -> Double
arcsin  =  asin
```

Parciální funkce doplněná chybovým hlášením

```
arcsin :: Double -> Double
arcsin x =
  if x >= -1 && x <= 1
    then asin x
    else error "arcsin: nedefinovano"
```

Dodefinovaná speciální hodnotou **Nothing** na totální funkci

```
arcsin :: Double -> Maybe Double
arcsin x =
  if x >= -1 && x <= 1
    then Just (asin x)
    else Nothing
```

Ošetření chyb

Dodefinovaná speciálními hodnotami `Left e` na totální funkci

```
arcsin :: Double -> Either String Double
arcsin x =
  if x >= -1 && x <= 1
    then Right (asin x)
    else Left "arcsin: argument mimo interval"
```

Faktoriál revisited

Dodefinujte parciální funkci `faktorial` i v záporných hodnotách na totální funkci `faktorial'`

```
faktorial' :: Integer -> Maybe Integer
faktorial' n = ...
```

Vyzkoušejte.

```
faktorial' (-1)
```

Druhy rekurze

- Zpětná
  po návratu ze zanořeného volání se ještě něco počítá
- Lineární
  v těle funkce se vyhodnocuje jen jedno rekurzivní volání
- Dopředná
  v těle funkce (tj. ve výrazu definujícím funkci) se všechna rekurzivní volání vyhodnocují jako poslední; dopředná a neparalelizovaná rekurze je koncová
- Koncová, *tail recursion* = dopředná a lineární
  Při koncové rekurzi je rekurzivní volání jen jedno a je posledním vyhodnoceným podvýrazem v těle funkce.

Druhy rekurze

Pomocí kterého druhu rekurze je definovaná funkce `ff` a co počítá ?

```
ff :: Integer -> Integer
ff n = if n <= 1 then 1 else g 1 n
  where
    g z 0 = z
    g z n = g (z*n) (n-1)
```

Fibonacciho posloupnost (parciální funkce)

Pomocí kterého druhu rekurze je definovaná funkce `fib`?

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

Definujte ji pomocí koncové rekurze.

```
fib' :: Integer -> Integer
fib' n = ...
  where
    f :: ...
    f ... = ...
    f ... = ...
```

Zkuste:

```
:set +s
fib 35
fib' 35000
```

Seznamy

Konstruktory:

- `[]` – konstruuje prázdný seznam
- `:` – konstruuje nový seznam přidáním prvku na začátek

```
1:2:3:[] ≡ [1,2,3]
```

Dva konstruktory hodnot `(:)` a `[]`. Pattern-matching aplikujeme pro oba případy `[]`, `(x:xs)`

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs
```

Transformace seznamu

Definujte spojování seznamů.

```
spoj :: [a] -> [a] -> [a]
-- spoj [1, 2] [3, 4, 5] = [1, 2, 3, 4, 5]
spoj [] ys =
spoj (x:xs) ys =
```

Částečná aplikace

```
(+)       :: Num a =>  a -> a -> a
(+) 1     :: Num a =>       a -> a
(+) 1 2   :: Num a =>            a
```

Funkce v Haskellu mají jen jeden parametr

```
a -> b -> c -> d   ≡   a -> (b -> (c -> d))

    f x y z        ≡        ((f x) y) z
```

curry, uncurry

Lze však také definovat funkce na dvojicích.

```
csoucet :: Int -> Int -> Int
csoucet x y = x + y

usoucet :: (Int, Int) -> Int
usoucet (x, y) = x + y
```

Převod mezi funkcí s více parametry a funkcí na dvojicích:

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

csoucet = curry usoucet
usoucet = uncurry csoucet
```