Funkcionální programování

Přednáška 10, LS 2013/2014

Teoretické základy

Lambda Calculus - Syntax

$e \rightarrow x$	variable
$e_1 \; e_2$	application
$\lambda x.e$	abstraction

Sémantika Lambda kalkulu

$\lambda x_i.e \Rightarrow \lambda x_j.[x_j/x_i]e$ iff x_j is not free in e (α-conversion, renaming)

$(\lambda x.e_1)e_2 \Rightarrow [e_2/x]e_1$ (β-conversion, application)

$\lambda x.(e\ x) \Rightarrow e$ iff x is not free in e (η-conversion)

Výpočetní síla Lambda kalkulu

- Lambda kalkulus má stejnou výpočetní sílu jako Turingův stroj. Pomocí Lambda kalkulu můžeme definovat funkce, které reprezentují:
    - čísla
    - booleovské hodnoty a podmínky
    - rekurzivní výpočet

Příklad

- Mějme funkci reprezentující “0” a “1”:

$$\lambda sz.z \quad \lambda sz.s(z)$$

- Mějmě funkci reprezentující následníka:

$$S \equiv \lambda wyx.y(wyx)$$

- Aplikace funkce S na funkci “0”:

$$S0 \equiv (\lambda wyx.y(wyx))(\lambda sz.z)$$

$$\lambda yx.y((\lambda sz.z)yx) = \lambda yx.y((\lambda z.z)x) = \lambda yx.y(x) \equiv 1$$

Podmínky

- Definujeme funkci True:

$$T \equiv \lambda xy.x$$

- a False:

$$F \equiv \lambda xy.y$$

Logické výrazy

- AND funkce dvou argumentů

$$\wedge \equiv \lambda xy.xy(\lambda uv.v) \equiv \lambda xy.xyF$$

- OR funkce dvou argumentů

$$\vee \equiv \lambda xy.x(\lambda uv.u)y \equiv \lambda xy.xTy$$

- Negace

$$\neg \equiv \lambda x.x(\lambda uv.v)(\lambda ab.a) \equiv \lambda x.xFT$$

- Negace aplikovaná na T

$$\neg T \equiv \lambda x.x(\lambda uv.v)(\lambda ab.a)(\lambda cd.c)$$

- se zredukuje na

$$TFT \equiv (\lambda cd.c)(\lambda uv.v)(\lambda ab.a) = (\lambda uv.v) \equiv F$$

Funkcionální jazyky

- Neobsahují příkazy, pouze funkce.
- Immutable proměnné.
- Nemají žádný implicitní stav.
- Program se specifikuje pomocí skládání funkcí.

Skládání funkcí

- Nemá žádné **vedlejší efekty** (nemáme implicitní stav ani globální proměnné).
- **Funkce vyššího řádu** (funkce, která má jako parametr funkci).
- **Currying** (funkce více proměnných může být reprezentována jako složená funkce z funkcí jedné proměnné).
- **Pattern matching** (parametry funkce lze specifikovat pomocí struktury vstupních dat).

Vyhodnocování

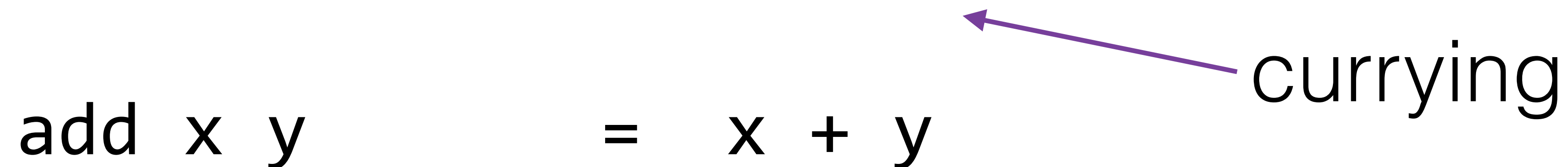
- Obvykle **není striktní** (ale líné), umožňuje specifikovat **nekonečné datové struktury** (např. celá čísla).
- Lze **snadněji paralelizovat** díky absenci globálního stavu.
- **Memoization** - pokud volám funkci se stejnými parametry vícekrát, pak si zapamatuji výsledek a již ji nemusím znovu vyhodnocovat (výpočet faktoriálu).

Ukázky v Haskellu

Ukázky v Haskellu

```
add              :: Integer -> Integer -> Integer
```

currying



```
add x y          =  x + y

inc              = add 1

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

funkce vyššího řádu

```
map f  []        =  []

map f (x:xs)             =  f x : map f xs
```

konstruktor seznamu

```
map (add 1) [1,2,3] => [2,3,4]
```

Ukázky v Haskellu

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

filter _ [] = []

filter p (x:xs)

    | p x       = x : filter p xs

    | otherwise = filter p xs
```

Ukázky v Haskellu

```
quicksort :: (Ord a) => [a] -> [a]

quicksort [] = []

quicksort (x:xs) =

 let smallerSorted = quicksort (filter (<=x) xs)

     biggerSorted = quicksort (filter (>x) xs)

        in  smallerSorted ++ [x] ++ biggerSorted
```

“Nekonečné” datové struktury

Nekonečný seznam “1”:

```
ones           = 1 : ones
```

Seznam čísel od “n”:

```
numsFrom n  = n : numsFrom (n+1)
```

Seznam čtverců čísel od 0:

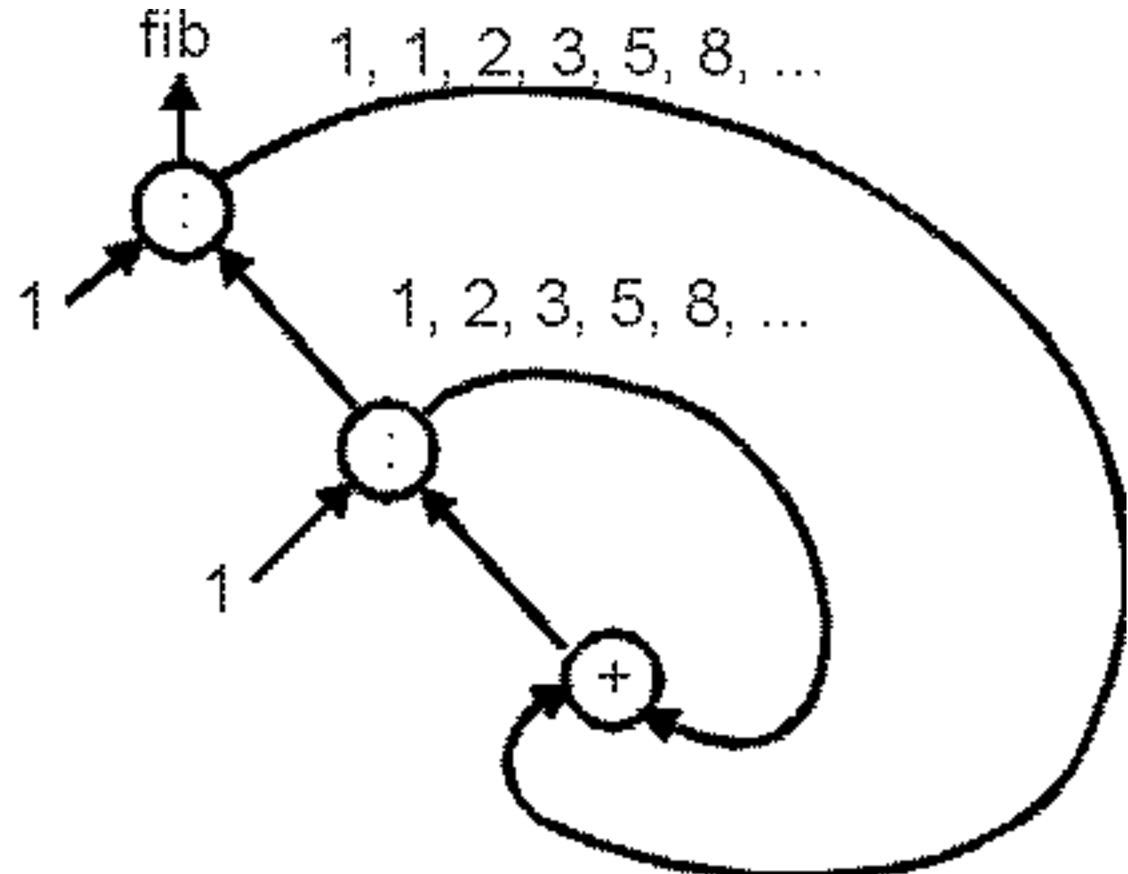
```
squares     = map (^2) (numsfrom 0)
```

- Fibonacciho posloupnost:

```
fib              = 1 : 1 : [ a+b | (a,b) <- zip fib
(tail fib) ]

zip (x:xs) (y:ys) = (x,y) : zip xs ys

zip  xs     ys    = []
```



```
data Regexp = Character Char | Concat Regexp Regexp | Altern Regexp Regexp

m :: Regexp -> Maybe String -> (Maybe String -> Maybe String) -> Maybe String

m regex Nothing cont = cont Nothing

m (Character c) (Just "") cont = cont Nothing

m (Character c) (Just (first:rest)) cont

        | c == first = cont (Just rest)

        | otherwise = cont Nothing

m (Concat r1 r2) str cont = m r1 str (\param -> m r2 param cont)

m (Altern r1 r2) str cont =  m r1 str (\param -> if cont param /= Just ""

                         then m r2 str cont

                         else cont param)

match :: Regexp -> String -> Bool

match regexp str = (m regexp (Just str) id) == Just ""

main = putStrLn (show (match (Character 'a') "a"))
```

Ukázky v Java

Functional Java

- Knihovna implementující principy funkcionálního programování v Java.
- http://functionaljava.org

Funkce filter

```
import fj.data.Array;

import static fj.data.Array.array;

import static fj.pre.Show.arrayShow;

import static fj.pre.Show.intShow;

import static fj.function.Integers.even;

public final class Array_filter {

  public static void main(final String[] args) {

    final Array<Integer> a = array(97, 44, 67, 3, 22, 90, 1, 77, 98, 1078, 6,
64, 6, 79, 42);

    final Array<Integer> b = a.filter(even);

    arrayShow(intShow).println(b); // {44,22,90,98,1078,6,64,6,42}

  }

}
```

Funkce Fold

```
import fj.data.Array;

import static fj.data.Array.array;

import static fj.function.Integers.add;


public final class Array_foldLeft {

  public static void main(final String[] args) {

    final Array<Integer> a = array(97, 44, 67, 3, 22, 90, 1, 77, 98, 1078, 6, 
64, 6, 79, 42);

    final int b = a.foldLeft(add, 0);

    System.out.println(b); // 1774

  }

}
```

Funkce Map

```
import fj.data.Array;

import static fj.data.Array.array;

import static fj.function.Integers.add;

import static fj.pre.Show.arrayShow;

import static fj.pre.Show.intShow;


public final class Array_map {

  public static void main(final String[] args) {

    final Array<Integer> a = array(1, 2, 3);

    final Array<Integer> b = a.map(add.f(42));

    arrayShow(intShow).println(b); // {43,44,45}

  }

}
```