IB015 Neimperativní programování

Funkce vyšších řádů a λ-funkce

Jiří Barnat
Libor Škarvada

Připomenutí

Datové struktury

- Uspořádané n-tice, seznamy.

Hodnoty a typy

- Elementární, složené typy, typy funkcí.
- Polymorfní a kvalifikované typy.
- Hodnotové konstruktory.

Rekurze

- Rekurzivní funkce na seznamech.

Funkce vyšších řádů

Funkce vyšších řádů

Definice

- Funkce, je považována za funkci vyššího řádu, pokud alespoň jeden z jejich argumentů, které funkce má, nebo výsledek, který funkce vrací, je opět funkce.
- Funkce vyššího řádu se též označují jako funkcionály.

Příklady

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

N-ární funkce jako funkce vyšších řádů

Pozorování

- Každou funkci, která má alespoň 2 argumenty, lze chápat jako funkci vyššího řádu.

Příklad

- Funkci

  ```
  (*) :: Num a => a -> a -> a
  ```

  lze číst jako

  ```
  (*) :: Num a => a -> (a -> a)
  ```

- Funkci, která bere dva číselné argumenty typu `a` a vrací hodnotu typu `a`, lze také chápat jako funkci, která bere hodnotu číselného typu `a` a vrací hodnotu typu `a -> a`.

Částečná aplikace

Pozorování

- Chápeme-li n-ární ($n \geq 2$) funkce jako funkce vyššího řádu, lze tyto funkce tzv **částečně aplikovat**, tj. vyhodnotit je i pro neúplný výčet argumentů.

Příklad

- Uvažme funkci násobení

```
(*) :: Num a => a -> (a -> a)
(*) x y = x*y
```

- Výsledkem aplikace `(*)` na hodnotu `3` je funkce.

```
(*) 3 :: Num a => a -> a
```

- Tuto funkci je možné označit, a posléze použít.

```
f = (*) 3
f :: Num a => a -> a
f 4 ⇝ ((*) 3) 4 ⇝ 12
```

Implicitní závorkování typu a aplikace

Připomenutí

- Typový konstruktor `->` je binární.
- Typový konstruktor `->` se používá pouze infixově.

Implicitní závorkování

- Typový konstruktor `->` implicitně sdružuje zprava

  $\mathbf{f}$ `::` a_1 `->` `(` a_2 `->` `(` a_3 `->` `...` `->` `(` a_{n-1} `->` `(`a_n `->` a`))` `...` `))`

- Aplikace funkce na argumenty implicitně sdružuje zleva

  `(` `...` `(` `(` `(` $\mathbf{f}$ x_1 `)` x_2 `)` x_3 `)` `...` x_{n-1} `)` x_n

Pozorování

- Libovolnou n-ární funkci lze také chápat jako k-ární, kde k nabývá hodnot od 1 do n.

Odvozování typu

S každou aplikací ubude jeden výskyt `->` v typu výrazu

```
(+)       :: Num a => a -> a -> a
(+) 2     :: Num a =>      a -> a
((+) 2) 3 :: Num a =>           a
```

Odvozování typu

S každou aplikací ubude jeden výskyt `->` v typu výrazu

```
(+)       :: Num a => a -> a -> a
(+) 2     :: Num a =>      a -> a
((+) 2) 3 :: Num a =>           a
```

Specializací typové proměnné však mohou `->` přibýt.

- Vezměme například funkci identity

  ```
  id :: a -> a
  id x = x
  ```

- Při aplikaci `id` na `(+)` ubude `->` z typu `id`, tj.

  ```
  id (+) :: a
  ```

- Avšak po specializaci typové proměnné `a` na typ funkce `(+)`

  ```
  id (+) :: Num a => a -> a -> a
  ```

Pořadí parametrů a částečná aplikace

Částečná aplikace

- Má-li funkce více formálních parametrů částečná aplikace probíhá vždy od parametru nejvíce vlevo.

Problém

- Funkci nelze přímo částečně aplikovat na jiný než první parametr.

Funkce `flip`

- Při aplikaci na binární funkci tuto funkci modifikuje tak, aby své dva argumenty přijímala v obráceném pořadí.

```
flip :: ( a -> b -> c ) -> b -> a -> c
flip f x y = f y x
```

- Funkci `flip` je třeba chápat jako **modifikátor funkce**, ne jako prohazovačku parametrů!

Příklady

```
(-) 3 4 ⇝ -1
flip (-) 3 4 ⇝ 1
(-) flip 3 4 ⇝ ERROR
```

Příklad, částečná aplikace a funkce `flip`

Příklad

- Pomocí částečné aplikace funkce `(-)` definujte novou funkci `minus4` , která při aplikaci na číselnou hodnotu vrátí hodnotu o 4 menší.

Řešení

- Definice s využitím částečné aplikace, bez formálních parametrů.

```
minus4 :: Num a => a -> a
minus4 = flip (-) 4
```

- Standardní definice téhož s využitím formálních parametrů.

```
minus4 :: Num a => a -> a
minus4 x = (-) x 4
```

Jak bránit částečné aplikaci

Pozorování

- Funkce vyššího řádu a částečná aplikace souvisí s násobným použitím funkcionálního typového konstruktoru `->` .
- Chceme-li zabránit částečné aplikaci, musíme definovat funkci tak, aby v jejím typu byl pouze jeden výskyt `->` .
- Pokud chceme funkci předat více argumentů najednou, předáme je jako uspořádanou n-tici.

Příklad

- Všimněte si rozdílu v typech a definicích následujících funkcí.

```
krat :: Num a => a -> a -> a
krat x y = x * y

krat1 :: Num a => (a,a) -> a
krat1 (x,y) = x * y
```

curry, uncurry

Funkce `curry` a `uncurry`

- Předdefinované funkce pro změnu řádu binárních funkcí.

curry

- Modifikuje funkci tak, že tato funkce místo uspořádané dvojice hodnot přijímá dva samostatné parametry.

```
curry :: ((a,b) -> c) -> a -> b -> c
curry f x y = f (x,y)
```

uncurry

- Modifikuje funkci tak, že tato funkce místo dvou samostatných parametrů přijímá uspořádanou dvojici hodnot.

```
uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (x,y) = f x y
```

Příklad – curry a uncurry

Příklad

- Mějme definovány následující funkce

```
krat :: Num a => a -> a -> a
krat x y = x * y

krat1 :: Num a => (a,a) -> a
krat1 (x,y) = x * y
```

- Uveďte alternativní definici funkce `krat` pomocí funkce `krat1` a obráceně.

Příklad – curry a uncurry

Příklad

- Mějme definovány následující funkce

```
krat :: Num a => a -> a -> a
krat x y = x * y

krat1 :: Num a => (a,a) -> a
krat1 (x,y) = x * y
```

- Uveďte alternativní definici funkce `krat` pomocí funkce `krat1` a obráceně.

Řešení

```
krat = curry krat1
krat1 = uncurry krat
```

Operátorové sekce

Co je to

- Pro každý **binární operátor** je možné definovat funkci, jež odpovídá částečné aplikaci funkce na první formální parametr a funkci, jež odpovídá částečné aplikaci funkce na druhý formální parametr. Těmto funkcím se říká **operátorové sekce**.

Operátorové sekce

- Předpokládejme binární operátor $\oplus$ a hodnoty `p` a `q`

```
⊕ :: a -> b -> c
p :: a
q :: b
```

- Částečnou aplikaci na první argument zapíšeme jako `(p⊕)`

```
(p⊕) = (⊕) p
```

- Částečnou aplikaci na druhý argument zapíšeme jako `(⊕q)`

```
(⊕q) = flip (⊕) q
```

Částečná aplikace – příklady

Příklad 1

- Jednoduché použití operátorových sekcí.

```
(*2) 34 ⇝ 68
(++".") ['V','ě','t','a'] ⇝ "Věta."
```

Příklad 2

- Odvoďte typ následujících funkcí, popište jejich význam:

```
(1.0/)
(`mod` 3)
(!!0)
(>0)
```

Částečná aplikace – příklady

Příklad 1

- Jednoduché použití operátorových sekcí.

```
(*2) 34 ⇝ 68
(++".") ['V','ě','t','a'] ⇝ "Věta."
```

Příklad 2

- Odvoďte typ následujících funkcí, popište jejich význam:

```
(1.0/)      :: Fractional a => a -> a
(`mod` 3)   :: Integral a => a -> a
(!!0)       :: [a] -> a
(>0)        :: (Num a, Ord a) => a -> Bool
```

Binární operátor pro skládání funkcí

Skládání funkcí

- Základní operátor funkcionálního programování

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g x = f ( g x )
```

Pozorování

- Operátor pro skládání funkcí lze chápat také jako binární.

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

- Pro operátor `(.)` je možné definovat operátorovou sekci.
- Použití operátorové sekce pro operátor `(.)` na jiné než jednoduché funkce je však velmi matoucí a v praxi se nepoužívá.

```
(.(.)) :: (((a -> b) -> a -> c) -> d) -> (b -> c) -> d
```

Binární operátor pro skládání funkcí

Skládání funkcí

- Základní operátor funkcionálního programování

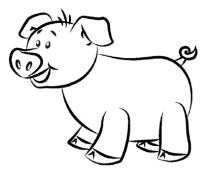
```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g x = f ( g x )
```

Pozorování

- Operátor pro skládání funkcí lze chápat také jako binární.

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

- Pro operátor `(.)` je možné definovat operátorovou sekci.
- Použití operátorové sekce pro operátor `(.)` na jiné než jednoduché funkce je však velmi matoucí a v praxi se nepoužívá.



```
(.(.)) :: (((a -> b) -> a -> c) -> d) -> (b -> c) -> d
```

Kombinátory

Pozorování

- Funkce `flip` , `curry` , `uncurry` , `(.)` a operátorové sekce používáme k vytvoření tzv. **bezparametrové** definice funkce.

Připomenutí bezparametrové definice

- Funkci `f` definovanou s použitím formálního parametru

```
f x = (not.odd) x
```

- Je možné definovat i bez použití formálního parametru

```
f = (not.odd)
```

Příklad

```
f x = (3*x)^7
f x = flip (^) 7 (3*x)
f x = flip (^) 7 ((*) 3 x)
f x = (.) (flip (^) 7) ((*) 3) x
f x = (.) (flip (^) 7) (3*) x
f = (.) (flip (^) 7) (3*)
```

Nepojmenované funkce

Jednorázové použité pojmenované funkce

Motivace

- Při standardní definici funkce musíme tuto funkci pojmenovat.
- Mnohé funkce, často jednoduché, použijeme jednorázově.
- Funkce s jednorázovým použitím je zbytečné pojmenovávat.

Příklad

- Globální definice a použití jednoduché funkce

```
f x = x*x + 1
map f [1,2,3,4,5] ⇝ [2,5,10,17,26]
```

- Lokální definice a použití jednoduché funkce

```
let f x = x*x + 1 in map f [1,2,3,4,5] ⇝ [2,5,10,17,26]
```

Nepojmenované funkce

Co to je

- Definice funkce v místě jejího použití bez uvedení jejího jména.
- Příklad:

```
map (\x -> x*x+1) [1,2,3,4,5] ⇝ [2,5,10,17,26]
```

Původ a všeobecné označení

- Myšlenka a teoretický základ pochází z Lambda kalkulu, proto se též nepojmenovaným funkcím říká **lambda funkce**.
- Principu vytváření lambda funkcí, se říká lambda abstrakce.

Pozorování

- Koncept lambda funkcí se vyskytuje v mnoha imperativních programovacích jazycích, např. C++, C#, SCALA, PHP, ...

Princip tvorby lambda funkcí

Lambda abstrakce

- Uvažme výraz M, který představuje tělo funkce.

  $$M \equiv x * x + 1$$

- Z těla funkce vytvoříme funkci použitím lambda abstrakce.

  $$\lambda x.M \equiv \lambda x.(x * x + 1)$$

- Při aplikaci lambda funkce $\lambda x.M$ na výraz N, se všechny volné výskyty formálního parametru x v M nahradí výrazem N. Výskyt proměnné x je označován jako volný, pokud není v rozsahu žádné lambda abstrakce.

  $$\lambda x.M\ N \equiv \lambda x.(x * x + 1)\ N = N * N + 1$$

Příklady

- $(\lambda x.x * x + 1)\ 3 = 3 * 3 + 1 = 10.$
- $(\lambda x.x + (\lambda x.x + x)\ x)\ 5 = 5 + (\lambda x.x + x)\ 5 = 5 + (5 + 5) = 15.$
- $(\lambda x.34)\ 3 = 34$

Lambda funkce v Haskellu

Definice a použití nepojmenované funkce

- λ se špatně píše na klávesnici.
- V programovacím jazyce *Haskell* je lambda abstrakce zapsána pomocí zpětného lomítka a šipky:

```
\ formální parametry -> tělo funkce
```

Příklady

```
(\ x -> x*x + x*x) 3 ⇝ ... ⇝ 18
(\ x -> x False) not ⇝ not False ⇝ True
```

Nepojmenovanou funkci je možné pojmenovat

```
f = (\ x -> x*x + x*x)
f 3 ⇝ ... ⇝ 18
f 3 ⇝ (\ x -> x*x + x*x) 3 ⇝ ... ⇝ 18
```

Nepojmenované funkce vyšších řádů

Pozorování

- Vnořená lambda abstrakce vytváří funkce vyšších řádů.
- $(\lambda x.(\lambda y.(x - y)))\ 3\ 4 = (\lambda y.(3 - y))\ 4 = 3 - 4 = -1$

Zápis v Haskellu

- Odvozený přímo z vícenásobné aplikace lambda abstrakce

```
\ x -> (\ y -> x - y)
( \ x -> (\ y -> x - y) ) 3 4 ⇝ ... ⇝ -1
```

- Zkrácený z důvodu čitelnosti kódu a pohodlí programátorů

```
\ x y -> x - y
( \ x y -> x - y) 3 4 ⇝ ... ⇝ -1
```

Nepojmenované funkce a typy

Nepojmenované funkce a jejich typy

- Obecně platí stejná pravidla jako pro pojmenované funkce.
- Název funkce (ani jeho neexistence) nemá vliv na typ funkce.

Efekt lambda abstrakce na typ

- Každý formální parametr v lambda abstrakci přidá do typu výrazu jeden výskyt typového konstruktoru `->` a novou typovou proměnnou, která se navíc specializuje podle kontextu použití konkrétního formálního parametru.

Pozorování

- Typ funkce si Hugs/GHCi umí odvodit z její definice.

Nepojmenované funkce a jejich typy – příklady

Příklad 1

```
'a'                    :: Char
(\ x -> 'a')           :: a -> Char
(\ x y -> 'a')         :: a -> b -> Char
(\ x -> (\ y -> 'a'))  :: a -> b -> Char
```

Příklad 2

```
\ x y -> x !! y        :: [a] -> Int -> a
\ x y -> x || y        :: Bool -> Bool -> Bool
```

Příklad 3

```
(\ x y -> x . y)       :: (a -> b) -> (c -> a) -> c -> b
(\ x y -> x y)         :: (a -> b) -> a -> b
```

Aplikační operátor $

Pozorování

- Vnořené aplikace funkcí mohou být díky závorkování nečitelné.
- Některým uzávorkováním se lze vyhnout použitím aplikačního operátoru `$` , který má při vyhodnocování nejnižší prioritu.

Aplikační operátor $

- ```
  ($) :: (a -> b) -> a -> b
  ($) f x = f x
  ```
- `f(g x)` ≡ `($) f (g x)` ≡ `f $ (g x)` ≡ `f $ g x`
- `f(g(h x))` ≡ `f $ g $ h x`
  ```

Aplikační operátor $ – příklady

Ekvivalentní zápisy pomocí operátoru $

- ```
  not ( not ( not ( not ( not True ))))
  not $ not $ not $ not $ not True
  ```

- ```
  (+1) ((+2) ((+3) ((+4) 0)))
  (+1) $ (+2) $ (+3) $ (+4) 0
  ```

- ```
  (even ((+1) 0)) || (odd ((+2) 0))
  (even $ (+1) 0) || (odd $ (+2) 0)
  ```

Příklady

- `(++) [1] $ map (+1) [1]` $\rightsquigarrow^*$ `[1,2]`
- `[1] ++ $ map (+1) [1]` $\rightsquigarrow^*$ **ERROR**
- Rozsah platnost `$` je podřízen syntaktickému pravidlu o zarovnání, tj. v `do` notaci "končí s koncem řádku".
  ```

Domácí cvičení

Mentální cvičení

- Definujte operátorové sekce pomocí lambda abstrakce.
- Identifikujte funkce vyššího řádu ve svém každodenním životě.