IB015 Neimperativní programování

Seznamy, Typy a Rekurze

Jiří Barnat
Libor Škarvada

Uspořádané n-tice a seznamy

Programování a data

Pozorování

- Programy pro své fungování potřebují různé informace – **data**.
- Data jsou vstupní hodnoty, výstupní hodnoty, mezivýsledky výpočtů, parametry funkcí, atd.

Programování a data

- Data je třeba uchovávat tak, aby je bylo možné zpracovat mechanicky/strojově.
- Tvorba jednoznačného popisu struktury a způsobu uložení dat je nedílná součást procesu programování.
- Pro uchování informací používáme různé **datové struktury**.

Strukturovaná data

Dekompozice dat

- Veškerá data použitá v programu je třeba vystavět ze základních datových elementů podle definovaných pravidel.
- Existují striktní pravidla pro dekompozici dat, my si však v rámci IB015 vystačíme s intuicí.

Základní datové elementy

- Čísla, Znaky, Pravdivostní hodnoty

Základní způsoby kompozice dat

- Uspořádané n-tice
- **Seznamy**

Uspořádané n-tice

Co to je

- Pevně daný počet nějakých hodnot v pevně daném pořadí.
- Prvek kartézského součinu nosných množin.

Příklady

- Datum: `(11, "březen", 1977)` .
- Přihlašovací údaje: `("xbarnat", "majen10cm")`
- Pozice pixelu v rastrovém obrázku: `(x, y)` ,
  všimněme si, že `(12,43)` $\neq$ `(43,12)` .

Kdy se má použít

- **Počet prvků v n-tici je znám předem**,
  tj. v okamžiku psaní zdrojového kódu.
- Počet prvků v n-tici je malý (hodnota n je malá).

Seznamy

Co to je

- Posloupnost hodnot stejného charakteru (stejného typu).
- Posloupnost může být prázdná, konečná i nekonečná.
- Každý prvek v seznamu je na nějaké (unikátní) pozici.

Příklady

- Seznam čísel: `[12,43,-3,15,29]`
- Nekonečný seznam: `[1,2,3,4,5,6,...]`
- Seznam uspořádaných dvojic:
  `[("Fero",12), ("Nero",7), ("Pero",5)]`
- Prázdný seznam: `[]`

Kdy se má použít

- **Data vznikají nebo se zpracovávají postupně.**
- Počet prvků použitých programem není předem znám.

Příklad dekompozice dat

Aplikace – Diář squashových partnerů.

- Program pro správu kontaktů na různé squashové hráče.
- Hlavní datová struktura je seznam kontaktů.

Datová dekompozice

- Seznam kontaktů

  `[kontakt1, kontakt2, kontakt3, ..., kontakt315]`

- Kontakt je uspořádaná trojice

  `(Prezdivka, Telefon, Adresa)`

- Adresa je uspořádaná pětice

  `(Jmeno, Prijmeni, Ulice, Cislo Popisne, Mesto)`

- `Prezdivka, Jmeno, Prijmeno, Ulice, Mesto` jsou seznamy znaků
- `Telefon, Cislo Popisne` jsou čísla

Hodnoty a Typy

Co není typ

Typ není váš nepřítel

Co není typ

Typ není váš nepřítel

Co je to typ a k čemu je

Co je to typ

- Označení množiny všech hodnot dané kvality.
- Komunikační prostředek napomáhající správnému skládání programů z jednotlivých funkcí.

K čemu se používají typy

- Každá hodnota, nebo výraz má svůj typ.
- Definice typové signatury funkcí.
- Kontrola logické konzistence programu v době překladu.
- Popis způsobu kompozice složených datových struktur. (Typy se komponují stejně jako data.)

Konstrukce typů

Základní datové typy

- `Int, Integer, Float, Char, Bool`

Složené typy

- Uspořádané n-tice:

  `(Bool,Int)`

- Seznamy:

  `[Int], [Char], [[Char]]`

  `[Char]` ≡ `String`

Funkcionální typy

- `Integer -> Bool, Float -> Float -> Float`

Funkcionální typy

Konstrukce

- Jsou-li σ a τ nějaké typy, tak σ `->` τ je typ všech funkcí s parametrem typu σ a funkční hodnotou typu τ.

Typ n-árních funkcí

- Jsou-li σ_1, σ_2, σ_3 $\ldots$ σ_n a τ nějaké typy, tak

  $$\sigma_1 \texttt{ -> } \sigma_2 \texttt{ -> } \sigma_3 \texttt{ -> } \ldots \texttt{ -> } \sigma_n \texttt{ -> } \tau$$

  je typ všech funkcí s prvním parametrem typu σ_1, druhým parametrem typu σ_2, ... a funkční hodnotou typu τ.

Terminologie

- **Arita funkce** označuje počet parametrů funkce.
- Konstanty, unární, binární, ternární funkce.
- Nulární funkce (n=0) jsou konstanty daného typu.

Funkcionální typ a aplikace

Pozorování

- Typ výrazu, který je úplná aplikace funkce na parametry, lze odvodit z typu použité funkce **bez nutnosti výpočtu výsledné hodnoty**.

Příklad

```
odd :: Integer -> Bool
27 :: Integer
odd 27 :: Bool
```

Polymorfní typy

Pozorování

- Některé funkce nepotřebují znát konkrétní typy formálních parametrů, pouze jejich strukturu.
- Místo konkrétního typu se použije **typová proměnná**.
- Při aplikaci funkce na konkrétní parametry, se za typovou proměnnou dosadí typ, který odpovídá použitému parametru. (Typová proměnná se specializuje.)
- **POZOR! Typová proměnná zastupuje i složené typy.**

Příklad

```
fst :: (a,b) -> a
(not,"Coze?") :: (Bool -> Bool,[Char])
fst (not ,"Coze?") :: Bool -> Bool
```

Typové třídy

Pozorování

- Některé funkce nevyžadují konkrétní typ, ale zároveň nedovolují použití libovolného typu, proto je třeba specializaci typové proměnné omezit na vybranou podtřídu typů.

Základní typové třídy

- `Integral` – celočíselné
- `Num` – numerické
- `Ord` – uspořadatelné
- `Eq` – porovnatelné na rovnost

Příklady typů s omezením specializace typové proměnné

```
odd :: Integral a => a -> Bool
(+) :: Num a => a -> a -> a
```

Uspořádané n-tice a seznamy v Haskellu

Uspořádané n-tice v Haskellu

Zápis uspořádaných n-tic

- Přirozený, pomocí závorek a čárek.
- Příklady zápisu uspořádaných n-tic v Haskellu:

```
(12,15)
(2,3,'a',5,6)
("Fiii","jo", 350, "tisíc", '!')
((1,1),(2,2),(3,3))
```

Krajní případy

- Jednotice se nepoužívají.
- Nultice: `()`

Předdefinované funkce pro uspořádané dvojice

Datové konstruktory

- `(,, ... ,)` – datový konstruktor uspořádané n-tice
- `(,)` – datový konstruktor uspořádané dvojice

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

Projekce

- `fst` , `snd` – projekce na první a druhou složku

```
fst :: (a,b) -> a
fst (x,y) = x

snd :: (a,b) -> b
snd (x,y) = y
```

Seznamy

Zápis seznamů

- V hranatých závorkách uzavřená posloupnost prvků oddělených čárkou.
- Seznam znaků též jako řetězec (text v uvozovkách).

Příklady

```
[3,3,3,3]
[ [1], [1,2], [1,2,3] ]
[]
"ahoj" = ['a','h','o','j']
"toto je také seznam"
[ or, or, or, and ]
```

Konstrukce seznamů

Datové konstruktory

- Prázdný seznam: `[]`

  ```
  [] :: [a]
  ```

- Operátor připojení prvku na začátek seznamu: `(:)`

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

Příklady

- Správné použití

  ```
  (:) 3 [3,3,3]    ⇝   [3,3,3,3]
  1:2:3:[]         ⇝   [1,2,3]
  4:[4,4,4,4]      ⇝   [4,4,4,4,4]
  'A':"hoj"        ⇝   "Ahoj"
  ```

- Nesprávné použití

  ```
  [2] : [3,4,5]    ⇝ ERROR
  [2,3,4] : 5      ⇝ ERROR
  'A' : [1,2,3]    ⇝ ERROR
  ```

Konstrukce seznamů – pokračování

Funkce pro spojení seznamů

- Seznamy **stejného typu** lze spojit pomocí funkce `(++)`

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

Příklady

- Správné použití

```
(++) "Ahoj " "světe!"  ⇝  "Ahoj světe!"
"Ahoj" ++ " " ++ "světe!"  ⇝  "Ahoj světe!"
[1,2,3] ++ [4,5,6]  ⇝  [1,2,3,4,5,6]
```

- Nesprávné použití

```
2 ++ [3,4,5]  ⇝ ERROR
[2,3,4] ++ 5  ⇝ ERROR
[2,3] ++ "text"  ⇝ ERROR
```

Význam datových konstruktorů

Datové konstruktory ve víceřádkových definicích

- Fungují jako vzory na levých stranách definice.
- Mapují se vždy na nejvnějšnější výskyt.

Příklady

- Funkce `null` aplikovaná na nějaký seznam, vrací `True` pokud je seznam prázdný a `False` pokud je neprázdný.

```
null :: [a] -> Bool
null (_:_) = False
null [] = True
```

- Funkce `snd` aplikovaná na uspořádanou dvojici, vrací druhý prvek dvojice.

```
snd :: (a,b) -> b
snd (_,y) = y
```

Pojmenovaný vzor – symbol @

Použití symbolu @

- Pokud `vzor` je korektně vytvořený datový vzor, pak zápisem `jmeno@vzor` získáme proměnnou `jmeno`, která bude po úspěšném použití vzoru odkazovat na celý mapovaný obsah.
- Nejčastěji používané ve spojení se seznamy, ale funguje všeobecně pro jakékoliv hodnoty konstruované s využitím datových konstruktorů.

Příklady

- Při mapování seznamu `[1,2,3]` na vzor `a@(x:t)`, bude

  `a = [1,2,3],` `x = 1,` `t = [2,3]` .

- `f (a@(x:y)) = x:a++y`

  `f [1,2,3]` $\rightsquigarrow^*$ `[1,1,2,3,2,3]`

  `f []` $\rightsquigarrow^*$ **ERROR**

Rekurze

Rekurze

Co je to rekurze

- Definice funkce, nebo datové struktury, s využitím sebe sama.

Význam v programování

- Umožňuje konečně dlouhý zápis definice funkce, která je definována pro nekonečně mnoho parametrů.
- V čistě funkcionálním jazyce nahrazuje cykly známé z imperativních programovacích jazyků.

Příklad

- Funkci `length` , která při aplikaci na seznam vrací jeho délku, je nutné definovat rekurzivně.

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

Příklad výpočtu rekurzivní definice

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

```
length [6,7,8,9]  ⇝ 1 + length [7,8,9]
                  ⇝ 1 + ( 1 + length [8,9])
                  ⇝ 1 + ( 1 + ( 1 + length [9]))
                  ⇝ 1 + ( 1 + ( 1 + ( 1 + length [])))

                  ⇝ 1 + ( 1 + ( 1 + ( 1 + 0 )))
                  ⇝ 1 + ( 1 + ( 1 + 1 ))
                  ⇝ 1 + ( 1 + 2 )
                  ⇝ 1 + 3
                  ⇝ 4
```

Další příklady použití rekurze

Číselné funkce

```
factorial :: Integer -> Integer

factorial 0 = 1
factorial x = x * factorial (x-1)
```

Nekonečné opakování (teoreticky)

```
main :: IO()

main = do  putStrLn "Vloz text:"
           x <- getLine
           putStrLn ( "Zadal jsi:" ++ x )
           main
```

Rekurze je mentálně náročná

Pozorování

- Použití rekurze je možné pouze tehdy, je-li podproblém, na nějž se rekurzivně aplikuje dané řešení, **stejného typu**.

Slovní úloha

- Na Jiříkovu narozeninovou párty přišlo 7 kamarádů a přineslo mimo jiné jeden narozeninový dort. Jiřík nebyl lakomý, rozdělil dort rovnoměrně mezi všechny přítomné. Jakou k tomu použil rekurzivní funkci? Jaký je typ této rekurzivní funkce?

Rekurze je mentálně náročná

Pozorování

- Použití rekurze je možné pouze tehdy, je-li podproblém, na nějž se rekurzivně aplikuje dané řešení, **stejného typu**.

Slovní úloha

- Na Jiříkovu narozeninovou párty přišlo 7 kamarádů a přineslo mimo jiné jeden narozeninový dort. Jiřík nebyl lakomý, rozdělil dort rovnoměrně mezi všechny přítomné. Jakou k tomu použil rekurzivní funkci? Jaký je typ této rekurzivní funkce?

Možné řešení

```
dort :: [KouskyDortu] -> [KouskyDortu]
dort s = if (length s >= 8)
           then s
           else dort (map (/2) s ++ map (/2) s)
```

Práce se seznamy v Haskellu

head, tail, init, last

První prvek seznamu

```
head :: [a] -> a
head (x:_) = x
```

Seznam bez prvního prvku

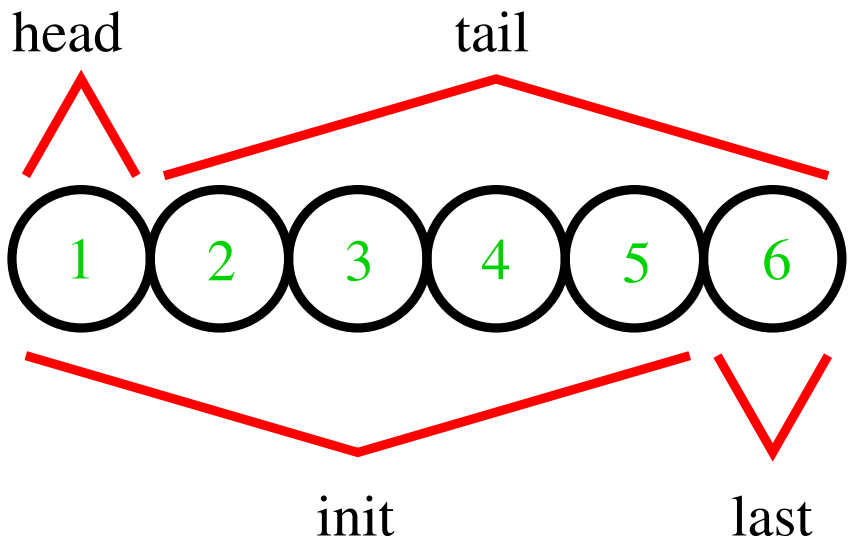
```
tail :: [a] -> [a]
tail (_:s) = s
```

Poslední prvek seznamu

```
last :: [a] -> a
last (x:[]) = x
last (_:s) = last s
```

Seznam bez posledního prvku

```
init :: [a] -> [a]
init (_:[]) = []
init (x:_:[]) = [x]
init (x:s) = x:init s
```



null, length, (!!)

Test na prázdný seznam

```
null :: [a] -> Bool
null (_:_) = False
null [] = True
```

Délka seznamu

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

N-tý prvek seznamu

```
(!!) :: [a] -> Int -> a
(x:_) !! 0 = x
(_:s) !! k = s !! (k-1)
```

take, drop

Prvních n prvků seznamu

```
take :: Int -> [a] -> [a]
take _ [] = []
take n (x:s) = if (n>0) then x : take (n-1) s
                        else []
```

Seznam bez prvních n prvků;

```
drop :: Int -> [a] -> [a]
drop _ [] = []
drop n (x:s) = if (n>0) then drop (n-1) s
                        else (x:s)
```

Poznámka

- Při infixovém použití binární funkce klesá její priorita!

  x : take (n-1) s = x : (take (n-1) s)

concat, filtr, replicate

Spojení seznamů v seznamu

```
concat :: [[a]] -> [a]
concat [] = []
concat (x:s) = x ++ concat s
```

Vynechání prvků nesplňujících danou podmínku

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter f (x:s) = if (f x) then x : filter f s
                          else filter f s
```

Vytvoření seznamu n-násobným kopírováním daného prvku

```
replicate :: Int -> a -> [a]
replicate 0 _ = []
replicate k x = x : replicate (k-1) x
```

takeWhile, dropWhile, map

Vynechání prvků seznamu od prvního, který nesplňuje podmínku

```
takeWhile :: (a->Bool) -> [a] -> [a]
takeWhile _ [] = []
takeWhile p (x:s) = if (p x) then x : takeWhile p s
                             else []
```

Vynechání prvků seznamu po první, který nesplňuje podmínku

```
dropWhile :: (a->Bool) -> [a] -> [a]
dropWhile _ [] = []
dropWhile p (x:s) = if (p x) then dropWhile p s
                             else x:s
```

Aplikace funkce na všechny prvky seznamu

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:s) = f x : map f s
```

zip,unzip

Spojení dvou seznamů do seznamu uspořádaných dvojic

```
zip :: [a] -> [b] -> [(a,b)]
zip [] _ = []
zip _ [] = []
zip (x:s) (y:t) = (x,y) : zip s t
```

Rozdělení seznamu dvojic na dvojici seznamů

```
unzip :: [(a,b)] -> ([a],[b])
unzip [] = ([],[])
unzip ((x,y):s) = (x : fst t, y : snd t) where t = unzip s
```

zipWith

Výpočet aplikace binární funkce na seznamy argumentů

```
zipWith :: (a->b->c)->[a]->[b]->[c]
zipWith _ _ [] = []
zipWith _ [] _ = []
zipWith f (x:s) (y:t) = f x y : zipWith f s t
```

Pozorování

```
zip = zipWith (,)
```

Práce na doma

Technická cvičení

- Vyzkoušejte si všechny odpřednášené funkce na modelových seznamech v prostředí interpretru jazyka Haskell.

Mentální cvičení

- Napište program, který pomocí principu rekurze a s využitím odpřednášených operací na seznamech vypočítá seznam obsahující čísla od 1 do 1024. Snažte se o to, aby hloubka rekurze byla co nejmenší.
- Jsou-li vám známy cykly s pevným počtem opakování z nějakého imperativního programovacího jazyka, popřemýšlejte o obecném postupu, jak nahradit tyto cykly voláním rekurzivní funkce.