Částečná aplikace, skládání funkcí, typování II, další funkce na seznamech

IB015 Neimperativní programování

Tomáš Szaniszlo, Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Podzim 2017

Částečná aplikace

Příklad 3.1.1: Co vyjadřuje výraz `min 6`? Napište ekvivalentní výraz pomocí `if`.

Částečná aplikace

Částečná aplikace

Umožňuje nám aplikovat funkci na méně parametrů, než očekává, a získat tak funkci, která bude očekávat zbylé parametry.

- `(+) 4`

Částečná aplikace

Částečná aplikace

Umožňuje nám aplikovat funkci na méně parametrů, než očekává, a získat tak funkci, která bude očekávat zbylé parametry.

- `(+) 4` (funkce, která očekává 1 parametr a přičte k němu 4)
- `(-) 3`

Částečná aplikace

Částečná aplikace

Umožňuje nám aplikovat funkci na méně parametrů, než očekává, a získat tak funkci, která bude očekávat zbylé parametry.

- `(+) 4` (funkce, která očekává 1 parametr a přičte k němu 4)
- `(-) 3` (funkce očekává 1 parametr a odečte ho od trojky)

Částečná aplikace

Částečná aplikace

Umožňuje nám aplikovat funkci na méně parametrů, než očekává, a získat tak funkci, která bude očekávat zbylé parametry.

- `(+) 4` (funkce, která očekává 1 parametr a přičte k němu 4)
- `(-) 3` (funkce očekává 1 parametr a odečte ho od trojky)

Operátorové sekce

Alternativní metoda zápisu částečné aplikace pro binární funkce.

- binární operátor a jeden argument v závorkách
- `(2 /)`

Částečná aplikace

Částečná aplikace

Umožňuje nám aplikovat funkci na méně parametrů, než očekává, a získat tak funkci, která bude očekávat zbylé parametry.

- `(+) 4` (funkce, která očekává 1 parametr a přičte k němu 4)
- `(-) 3` (funkce očekává 1 parametr a odečte ho od trojky)

Operátorové sekce

Alternativní metoda zápisu částečné aplikace pro binární funkce.

- binární operátor a jeden argument v závorkách
- `(2 /)` (vydělí 2 svým argumentem – levá operátorová sekce)
- `(/ 2)`

Částečná aplikace

Částečná aplikace

Umožňuje nám aplikovat funkci na méně parametrů, než očekává, a získat tak funkci, která bude očekávat zbylé parametry.

- `(+) 4` (funkce, která očekává 1 parametr a přičte k němu 4)
- `(-) 3` (funkce očekává 1 parametr a odečte ho od trojky)

Operátorové sekce

Alternativní metoda zápisu částečné aplikace pro binární funkce.

- binární operátor a jeden argument v závorkách
- `(2 /)` (vydělí 2 svým argumentem – levá operátorová sekce)
- `(/ 2)` (vydělí svůj argument 2 – pravá operátorová sekce)
- `` (`mod` 2) ``

Částečná aplikace

Částečná aplikace

Umožňuje nám aplikovat funkci na méně parametrů, než očekává, a získat tak funkci, která bude očekávat zbylé parametry.

- `(+) 4` (funkce, která očekává 1 parametr a přičte k němu 4)
- `(-) 3` (funkce očekává 1 parametr a odečte ho od trojky)

Operátorové sekce

Alternativní metoda zápisu částečné aplikace pro binární funkce.

- binární operátor a jeden argument v závorkách
- `(2 /)` (vydělí 2 svým argumentem – levá operátorová sekce)
- `(/ 2)` (vydělí svůj argument 2 – pravá operátorová sekce)
- `(`mod` 2)` (zjistí zbytek po dělení argumentu dvěma)

Curryfikace[1] – teorie

Předávání argumentů postupně („curryfikovaná“ funkce)

- umožňuje částečnou aplikaci
- `const :: a -> b -> a`

Předávání argumentů společne jako n-tice

- nelze použít částečnou aplikaci
- `fst :: (a, b) -> a`

Převody „curryfikovaných“ a „necurryfikovaných“ funkcí

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

Jaký je tedy vzájemný vztah `const` a `fst`?

[1] Pojmenováno po matematikovi a logikovi jménem Haskell Brooks Curry

Curryfikace[1] – teorie

Předávání argumentů postupně („curryfikovaná“ funkce)

- umožňuje částečnou aplikaci
- `const :: a -> b -> a`

Předávání argumentů společne jako n-tice

- nelze použít částečnou aplikaci
- `fst :: (a, b) -> a`

Převody „curryfikovaných“ a „necurryfikovaných“ funkcí

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

Jaký je tedy vzájemný vztah `const` a `fst`?

- `const` $\equiv$ `curry fst`, `uncurry const` $\equiv$ `fst`

[1] Pojmenováno po matematikovi a logikovi jménem Haskell Brooks Curry

Skládání funkcí

Jednou ze základních operací s funkcemi je jejich skládání.

- v matematice zapisujeme $f \circ g$, v Haskellu `f . g`,
  (`f` a `g` bereme jako unární funkce)
- aplikace složené funkce `(f . g) x` je ekvivalentní `f (g x)`
  - závorky jsou nutné, implicitní závorkování je `f . (g x)`
- operátor tečka má nejvyšší prioritu a je asociativní zprava
- `(.) ::`

Skládání funkcí

Jednou ze základních operací s funkcemi je jejich skládání.

- v matematice zapisujeme $f \circ g$, v Haskellu `f . g`,
  (`f a g` bereme jako unární funkce)
- aplikace složené funkce `(f . g) x` je ekvivalentní `f (g x)`
  - závorky jsou nutné, implicitní závorkování je `f . (g x)`
- operátor tečka má nejvyšší prioritu a je asociativní zprava
- `(.) :: (b -> c) -> (a -> b) -> (a -> c)`

Skládání funkcí v praxi

Příklad 3.2.1: Vyhodnoťte následující výrazy:

a) `((== 42) . (2 +)) 40`

b) `((> 2) . (* 3) . ((-) 4)) 5`

c) `filter ((>= 2) . fst) [(1,"a"), (2,"b"), (3,"c")]`

Typování funkcí: teorie

Intuitivní typování

- výraz posoudíme z hlediska toho, co dělá, a na základě toho určíme typ
- vhodné pro jednoduché výrazy

Algoritmické typování

- exaktní určení typu „typovacím algoritmem“
- zdlouhavější, ale funguje pro všechny výrazy

Typování funkcí: teorie

Typovací algoritmus

- každý výskyt funkce, konstanty, formálního argumentu otypujeme (nezávislé typové proměnné)
- určíme rovnosti vyplývající z aplikací funkcí na argumenty (aplikace vynucuje shodu typu formálního a skutečného parametru)
- rozepíšeme a zjednodušíme rovnosti na co nejjednodušší
- vyjádříme všechny typové proměnné pomocí minimální množiny typových proměnných
- určíme hledaný typ a dosadíme do něj vyjádření
- zohledníme typové kontexty, jsou-li nějaké
- volitelně převedeme typ do kanonického tvaru

Typování funkcí I.

Příklad 3.3.1: Určete typy výrazů:

a) `(&&) True`
b) `id "foo"`
c) `(&& False)`
d) `const True`
e) `const True False`
f) `(:[])`
g) `(:[]) True`
h) `[]:[]:[]`
i) `([]:[]):[]`

Zjednodušování před typováním

Výraz lze před otypováním vyhodnotit (zjednodušit). Ale pozor! Polymorfizmus může při intuitivním vyhodnocení způsobit změnu typu:

- `head [id, not]`
- `min 2 (3.1 :: Float)`

Typování funkcí II.

Příklad 3.3.3: Určete typy funkcí:

a) `swap (x,y) = (y,x)`

b) `cadr = head . tail`

c) `caar = head . head`

d) `twice f = f . f`

e) `comp12 g h x y = g (h x y)`

Typování funkcí IIi.

Příklad 3.3.4: Určete typy následujících funkcí:

a)
```
sayLength [] = "empty"
sayLength x  = "noempty"
```

b)
```
mswap True  (x, y) = (y, x)
mswap False (x, y) = (x, y)
```

c)
```
gfst (x, _) = x
gfst (x, _, _) = x
gfst (x, _, _, _) = x
```

d)
```
foo True []    = True
foo True (_:_) = False
foo False      = False
```

První pohled na typové třídy

Typové třídy umožňují omezit polymorfizmus funkcí:

- `(+) :: Num a => a -> a -> a`
  `Num a` představuje numerické typy
- `(==) :: Eq a => a -> a -> Bool`
  `Eq a` představuje typy porovnatelné na shodu
- `(<) :: Ord a => a -> a -> Bool`
  `Ord a` představuje uspořádatelné typy
- `show :: Show a => a -> String`
  `Show a` představuje typy, kterých hodnoty mají smysluplnou textovou reprezentaci

První pohled na typové třídy

Část typu před `=>` označujeme jako **typový kontext**.
Omezení může být i více:
`(^) :: (Num a, Integral b) => a -> b -> a`.

Při typování výrazů, které obsahují takto kvantifikovaně polymorfní typy, je nutné uvádět i přesný typový kontext.

Příklad 3.3.5: Určete typy následujících výrazů:
a) `(+ 3)`
b) `(+ 3.0)`
c) `filter (>= 2)`
d) `(> 2) . (`div` 3)`

Další funkce na seznamech

Funkcí na seznamech je velké množství, mnohé užitečné lze nalézt v modulu `Data.List`[2], základní i přímo z `Prelude`[3].

Funkce `zip` a `unzip` umožňují převod mezi dvojicí seznamů a seznamem dvojic:

```
zip [1,2,3] ['a','b','c'] ⇝*
    [(1,'a'),(2,'b'),(3,'c')]
unzip [(1,'a'),(2,'b'),(3,'c')] ⇝*
    ([1,2,3],['a','b','c'])
```

Délka výsledku funkce `zip` je určena délkou kratšího seznamu:

```
zip [1,2,3,4] ['a','b'] ⇝* [(1,'a'),(2,'b')]
```

[2] `http://hackage.haskell.org/package/base/docs/Data-List.html`
[3] `http://hackage.haskell.org/package/base/docs/Prelude.html#g:11`

Funkce `zipWith`: intuice

Funkce `zipWith` je zobecněním funkce `zip`:

- funkce `zip` spojuje prvky seznamů pomocí datového konstruktoru uspořádané dvojice `(,)`
- funkce `zipWith` má navíc jako argument funkci, kterou určíme způsob spojení prvků ze seznamů

```
zipWith f [x,y] [z,q] ⇝* [f x z, f y q]
zipWith (,) [1,2] ['a','b'] ⇝* [(1,'a'),(2,'b')]
zipWith (*) [5,10,11] [3,4] ⇝* [15,40]
```

Funkce `zipWith` I.

Příklad 3.4.6: Jaká je hodnota následujících výrazů?

a) `zipWith (^) [1..5] [1..5]`

b) `zipWith (:) "MF" ["axipes", "ík"]`

c)
```
let fibs = [0,1,1,2,3,5,8,13] in zipWith (+) fibs
  (tail fibs)
```

d)
```
let fibs = [0,1,1,2,3,5] in zipWith (/) (tail
  (tail fibs)) (tail fibs)
```

Funkce `zipWith` II.

Příklad 3.4.8: Napište funkci, která zjistí, jestli jsou v seznamu typu `(Eq a) => [a]` některé dva sousední prvky stejné. Úlohu zkuste vyřešit pomocí funkce `zipWith`.