Cvičení 3: Manipulace s funkcemi, typy

Před třetím cvičením je zapotřebí znát:

- typy základních entit v Haskellu (čísel, řetězců, seznamu, n-tic);
- základní typové třídy (pro čísla, desetinná čísla, porovnatelné a seřaditelné typy, zobrazitelné typy);
- použití operátoru `(.) :: (b -> c) -> (a -> b) -> (a -> c)` pro skládání funkcí;
- co je částečná aplikace;
- základní funkce pro manipulace s čísly a seznamy.

Na cvičení si prosím přineste papír a tužku, budou se hodit.

Pan Fešák doporučuje: Na tomto cvičení se vám bude hodit tužka a papír ještě více než obvykle. Cílem cvičení na papír je procvičit si přemýšlení nad jednotlivými koncepty, bez spoléhání se na interpret.

3.1 Typy a typové třídy

Př. 3.1.1 Určete typy výrazů:

a) `not True`
b) `(&&)`
c) `[]`
d) `"Don't Panic!"`
e) `[(True, ""), (False, [])]`
f) `[True, []]`

Pan Fešák doporučuje: Informace o typových třídách a v nich definovaných funkcích můžeme zjistit pomocí příkazu `:i` v GHCi. Např.

```
> :i Fractional
class Num a => Fractional a where
  (/) :: a -> a -> a
  recip :: a -> a
  fromRational :: Rational -> a
  {-# MINIMAL fromRational, (recip | (/)) #-}
        -- Defined in ‘GHC.Real’
instance Fractional Float -- Defined in ‘GHC.Float’
instance Fractional Double -- Defined in ‘GHC.Float’
```

To nám říká, že typová třída `Fractional` obsahuje funkce `(/)`, `recip` a `fromRational`, že do ní patří typy `Float` a `Double` a že je-li něco ve třídě `Fractional`, pak je to nutně také ve třídě `Num`. Krom toho nám tento výpis také

dává informace o tom, které funkce bychom minimálně museli implementovat, pokud bychom chtěli do této třídy nějaký nový typ přidat.

Chceme-li zjistit, zda jde nějaký kontext, například `(Num a, Floating a)` zjednodušit, potřebujeme zjistit, zda některá z těchto typových tříd vyžaduje některou další. V tomto případě bude potřeba navíc jít přes mezikrok:

```
class Fractional a => Floating a where {- ... -}
class Num a => Fractional a where {- ... -}
```

Tedy pokud je nějaký typ ve `Floating`, musí být nutně ve `Fractional` a tedy i v `Num`. Kontext tedy zjednodušíme na `Floating a`.

Př. 3.1.2 Určete typy výrazů. Snažte se neuvádět typové třídy, které jsou již implikovány jinými typovými třídami. Můžete použít interpret k zjištění typů knihovních funkcí a závislostí mezi typovými třídami.

a) `[1, 2, 3.14]`
b) `head [2 ^ 2]`
c) `[1, 2, True]`
d) `filter (\x -> x > 5) [1, 2, 4, 8]`
e) `\x -> show (x ^ x)`
f) `\x -> read x + 2`
g) `\x -> (fromIntegral x :: Double)`

Př. 3.1.3 Určete typy následujících funkcí:

a)
```
swap (x, y) = (y, x)
```
b)
```
maybeSwap True (x, y) = (y, x)
maybeSwap _    (x, y) = (x, y)
```
c)
```
sayLength [] = "empty"
sayLength x  = "nonempty"
```
d)
```
aOrX 'a' _ = True
aOrX _   x = x
```

Pan Fešák připomíná: V případě, že chci otypovat funkci definovanou víceřádkovou definicí, otypuji každý řádek zvlášť a pak unifikuji typy argumentů na odpovídajících pozicích a typy návratových hodnot.

Př. 3.1.4 Určete typ funkce `f`. Jak se funkce chová na různých vstupech?

```
f x y       True  = if x > 42 then y else []
f _ (_ : s) False = s
f _ _       _     = "IB015"
```

Př. 3.1.5 Určete typy následujících funkcí a popište slovně, co funkce dělají.

a)
```
cm _ []       = []
cm f (x : xs) = f x ++ cm f xs
```
b)
```
mm [] = error "empty list"
mm (x0 : xs0) = mm' x0 x0 xs0
    where
        mm' a b []       = (a, b)
        mm' a b (x : xs) = mm' (min a x) (max b x) xs
```

Př. 3.1.6 Vysvětlete význam a najděte příklady použití následujících funkcí:

a) `show :: Show a => a -> String`
b) `read :: Read a => String -> a`
c) `fromIntegral :: (Integral a, Num b) => a -> b`
d) `round :: (RealFrac a, Integral b) => a -> b`

3.2 Částečná aplikace a operátorové sekce

Př. 3.2.1 Vysvětlete, co dělají následující funkce, a najděte argumenty, na něž je lze aplikovat. Následně si chování ověřte v GHCi.

a) `(+) 2`
b) `(* 2)`
c) `(- 2)`
d) `(2 -)`
e) `/ 2`

Př. 3.2.2 Přepište v následujících definicích seznamových funkcí lambda funkce na částečnou aplikaci tak, aby funkčnost zůstala stejná.

a)
```
import Data.Char
upper :: String -> String
upper xs = map (\x -> toUpper x) xs
```
b)
```
embrace :: [String] -> [String]
embrace xs = map (\x -> '[' : x) (map (\x -> x ++ "]") xs)
```
c)
```
sql :: (Ord a, Num a) => [a] -> a -> [a]
sql xs lt = map (\x -> x ^ 2) (filter (\x -> x < lt) xs)
```

Př. 3.2.3 Které z následujících výrazů jsou ekvivalentní?

a) `f 1 g 2` $\overset{?}{\equiv}$ `f 1 (g 2)`
b) `(f 1 g) 2` $\overset{?}{\equiv}$ `(f 1) g 2`
c) `(* 2) 3` $\overset{?}{\equiv}$ `2 * 3`
d) `(*) 2 3` $\overset{?}{\equiv}$ `(* 3) 2`

3.3 Skládání funkcí, η-redukce, odstraňování argumentů

Pan Fešák vysvětluje: Řecké písmeno η je éta (a jeho ocásek se píše pod linku, stejně jako například u našeho j). Pojem η-redukce pochází z Lambda kalkulu a představuje odstraňování formálních argumentů. Někdy se můžete setkat také s pojmem η-konverze, který představuje jak odebírání argumentů, tak i jejich přidávání (tam, kde to typ dovoluje). Písmeno η bylo vybráno kvůli souvislosti s *extensionalitou*, tedy s tvrzením $f = g \iff \forall x.(f(x) = g(x))$.

Pan Fešák vysvětluje: Odstraněním argumentů z funkce ji převedeme na *pointfree* tvar, naopak pokud funkce má v definici všechny argumenty, o nichž hovoří její typ, je v *pointwise* tvaru. Onen *point* v těchto názvech představuje argument funkce (bod), nikoli tečku (skládání funkcí), více naleznete na Haskell Wiki. Mezi těmito tvary lze vždy převádět, někdy to však není vhodné či snadné, jednak kvůli čitelnosti, jednak je obtížné odstranit argument, který je v definici funkce použit vícekrát.

> **Jindřiška varuje:** Nic se nesmí přehánět, funkce jako `(.(,)) . (.) . (,)` nebo `(.) . (.)` nejsou ani hezké ani čitelné.

Př. 3.3.1 Otypujte následující výrazy:

a) `map even`
b) `map head . snd`
c) `filter ((4 >) . maximum)`
d) `const const`

Př. 3.3.2 Přepište v následujících definicích seznamových funkcí lambda funkce pomocí skládání funkcí, částečné aplikace nebo operátorové sekce tak, aby funkčnost zůstala stejná. Odstraňte také formální argumenty funkcí, pokud to je smysluplné.

a)
```
failing :: [(Int, Char)] -> [Int]
failing sts = map fst (filter (\t -> snd t == 'F') sts)
```

b)
```
embraceWith :: Char -> Char -> [String] -> [String]
embraceWith l r xs = map (\x -> l : x ++ [r]) xs
```

(`l` a `r` neodstraňujte)

c)
```
divisibleBy7 :: [Integer] -> [Integer]
divisibleBy7 xs = filter (\x -> x `mod` 7 == 0) xs
```

d)
```
import Data.Char
letterCaesar :: String -> String
letterCaesar xs = map (\x -> chr (3 + ord x)) (filter isLetter xs)
```

e)
```
zp :: (Integral a, Num b) => [a] -> [b] -> [b]
zp xs ys = zipWith (\x y -> y ^ x) xs ys
```

Př. 3.3.3 Mezi následujícími výrazy najděte všechny korektní a mezi nimi rozhodněte, které jsou vzájemně ekvivalentní (vzhledem k chování na libovolných vstupech povolených typem výrazu). Zdůvodněte neekvivalenci.

```
flip (>) 42 . flip (*) 2
flip > 42 . flip * 2
flip (> 42) . flip (* 2)
(\x -> x > 42) . (* 2)
(>) 42 . (*) 2
(<) 42 . (*) 2
\x -> (x * 2) > 42
(> 42) . (* 2)
(* 2) . (> 42)
* 2 . > 42
(> 42) (* 2)
\x -> ((> 42) . (* 2)) x
```

Př. 3.3.4 Uvažme funkci `negp :: (a -> Bool) -> a -> Bool`, která neguje výsledek unárních predikátů (funkcí typu `a -> Bool`). Tj. funkce `negp pred` vrátí opačnou logickou hodnotu, než by vrátil predikát `pred` na zadané hodnotě. Tedy například `negp even` by mělo být ekvivalentní s `odd`.

a) Definujte funkci `negp` (můžete využít třeba funkci `not`).
b) Definujte funkci `negp` jako unární funkci (s použitím pouze jednoho formálního parametru).
c) Definujte funkci `negp` bez použití formálních parametrů.

Př. 3.3.5 Pokud to je možné, přepište lambda funkce v následujících definicích pomocí skládání funkcí, částečné aplikace nebo operátorové sekce tak, aby funkčnost zůstala stejná. Odstraňte také formální argumenty funkcí.

```
import Data.Char

-- | Convert lowercase letters to numbers 0..25
l2c :: Char -> Int
l2c c = ord c - ord 'a'

-- | Convert 0..25 character codes to uppercase letters
c2l :: Int -> Char
c2l c = chr (c + ord 'A')

-- | Keep only lowercase English letters
lowAlphaOnly :: String -> String
lowAlphaOnly xs = filter (\x -> isLower x && isAscii x) xs

-- | Encrypt messages using Vigenere (one-time-pad) cipher
letterVigenere :: String -> String -> String
letterVigenere xs ks = zipWith
    (\x y -> c2l ((l2c x + l2c y) `mod` 26))
    (lowAlphaOnly xs)
    (lowAlphaOnly ks)
```

Nápověda: formální argument nelze odstranit (s tím, co jsme se učili), pokud je v definici použit vícekrát.

K odstranění formálního argumentu v použitého vícekrát v těle funkce lze použít funkci `(<*>)`. Její typ je sice velice obecný (a komplikovaný), ale pro tyto účely ji můžeme otypovat jako `(<*>) :: (a -> b -> c) -> (a -> b) -> a -> c`. Rovněž ji pro tyto účely můžeme nahradit následující funkcí:

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

Př. 3.3.6 Převeďte následující funkce do pointfree tvaru, neboli odstraňte formální argumenty lambda abstrakcí:

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

Př. 3.3.7 Převeďte následující výrazy do pointwise tvaru, neboli přidejte všechny argumenty, které plynou z typu výrazu:

a) `(^ 2) . mod 4 . (+ 1)`

b) `(+) . sum . take 10`
c) `map f . flip zip [1, 2, 3]` (funkce `f` je definována externě)
d) `(.)`

Př. 3.3.8 Určete typ následujících funkcí. Přepište tyto definice funkcí tak, abyste v jejich definici nepoužili λ-abstrakci a formální parametry (tj. chce se pointfree definice).

> **Pan Fešák vysvětluje:** Pokud potřebuji odstranit formální parametr, jenž se nevyskytuje v těle výrazu, pomůžu si funkcí `const`: pro libovolný výraz `w`, který nepřidává žádná nová typová omezení, je výraz `v` ekvivalentní s výrazem `const v w`. Tento výraz už obsahuje v těle navíc parametr `w`, který se dá použít pro η-redukci.

a) `f x y = y`
b) `f x y = 3 + x`

Př. 3.3.9 Převeďte následující funkce do pointfree tvaru:

a) `_ -> x`
b) `\x -> f x 1`
c) `\x -> f 1 x True`
d) `const x`
e) `\x -> 0`
f) `\x -> if x == 1 then 2 else 0`
g) `\f -> flip f x`

Př. 3.3.10 Převeďte všechny níže uvedené funkce do pointfree tvaru. Při převodu třetí si pomozte převodem druhé.

a) `f1 x y z = x`
b) `f2 x y z = y`
c) `f3 x y z = z`

Př. 3.3.11 Zapište v pointfree tvaru funkci `g x = f x c1 c2 c3 ... cn` (`f` je nějaká pevně daná funkce a `c1`, `c2`, ..., `cn` jsou konstanty).

* * *

> **Na konci cvičení byste měli zvládnout:**
>
> - otypovat výrazy a funkce, a to včetně polymorfních funkcí využívajících typové třídy a včetně funkcí definovaných podle vzoru;
> - Na intuitivní úrovni znát význam typových tříd `Num`, `Integral`, `Fractional`, `Show`, `Read`;
> - používat operátorové sekce a částečnou aplikaci;
> - skládat funkce a pochopit kód obsahující skládání funkcí;
> - odstranit formální parametry funkce (pokud je každý použit nejvýše jednou);
> - z typu poznat, kolik argumentů funkce má, a umět je přidat do její definice a odstranit při tom skládání funkcí a nyní již zbytečné částečné aplikace.

Řešení

Řeš. 3.1.1
a) `Bool`
b) `Bool -> Bool -> Bool`
c) `[a]`
d) `String` neboli `[Char]`
e) `[(Bool, String)]`
f) Nelze otypovat, seznamy musí být homogenní (všechny hodnoty musí mít stejný typ).

Řeš. 3.1.2
a) `Fractional a => [a]`; tedy `a` může být libovolný typ schopný reprezentovat zlomky (popřípadě také `(Num a, Fractional a) => [a]`, ale `Fractional` implikuje `Num`) Pozor, stále se jedná o seznam, jehož všechny prvky mají stejný typ. Jen dosud není řečeno, jaký to bude, jen že to musí být nějaký typ z třídy `Fractional`. Celá čísla lze samozřejmě reprezentovat i pomocí typů z třídy `Fractional`, např. `Double`.
b) `Num a => a`
c) Nelze otypovat. Chybová hláška `No instance for (Num Bool) arising from the literal '1'` říká, že `Bool` není číslo (instance `Num`).
d) `(Ord a, Num a) => [a]` (obě části kontextu jsou nutné, mezi `Ord` a `Num` není žádný vztah implikace)
e) `(Show a, Integral a) => a -> String`; `Show` je typová třída typů, které lze převést do textové reprezentace.
Popřípadě také `(Show a, Num a, Integral a) => a -> String`, ale `Integral` implikuje `Num`.
f) `(Num a, Read a) => String -> a`; `Read` je typová třída typů, jejichž hodnoty lze parsovat z textové reprezentace (typ výsledku funkce `read` se odvodí z kontextu použití.
g) `Integral a => a -> Double`; explicitní otypování je zde použito k vynucení konverze na konkrétní typ, `fromIntegral :: (Integral a, Num b) => a -> b` totiž umožňuje konverzi celého čísla na libovolný číselný typ.

Řeš. 3.1.3
a) Ze vzoru vidíme, že vstupem je dvojice, a z výrazu na pravé straně vidíme, že i návratový typ je dvojice. Zároveň typ první složky v argumentu musí být stejný jako typ druhé složky v návratovém typu a naopak. Celkově tedy `swap :: (a, b) -> (b, a)`.

b) První argument musí být typu `Bool` podle prvního řádku definice. Podle prvního řádku tedy dostaneme typ `Bool -> (a, b) -> (b, a)`, zatímco podle druhého `Bool -> (c, d) -> (c, d)`. Jelikož typy na odpovídajících pozicích musíme unifikovat (tedy `(a, b)` ∼ `(c, d)` a `(b, a)` ∼ `(c, d)`), jediná možnost je, že oba typy ve dvojici budou stejné. `maybeSwap :: Bool -> (a, a) -> (a, a)`

c) Z toho, že v obou vzorech je právě jeden argument a funkce vrací `String`, vidíme, že nejobecnější možný typ funkce je `a -> String`. Typ argumentů funkce však není závislý jen na jejich použití na pravé straně definice, ale i na vzorech. Jelikož `[]` je vzor prázdného seznamu, musí být argument funkce seznamového typu. Další omezení již nejsou, dostáváme tedy `sayLength :: [a] -> String`.

d) Z použitých vzorů můžeme odvodit, že funkce bere dva argumenty a že první je typu `Char`. Z návratové hodnoty prvního řádku můžeme odvodit typ `Bool`. Zbývá už jen určení typu druhého argumentu. Ve druhém vzoru si můžeme všimnout, že vracíme hodnotu, kterou bereme ve druhém argumentu. Takže obě mají stejný typ, a protože návratová hodnota má typ `Bool`, i druhý argument bude mít typ `Bool`. Dostáváme tedy `aOrX :: Char -> Bool -> Bool`

Řeš. 3.1.4 Prvně otypujeme řádky jednotlivě:

1. První argument musí jistě být číslo a porovnatelný, typ druhého argumentu musí být seznamový, protože se objevuje v `then` větvi `if`u, kde se v `else` větvi objevuje seznam, typ třetího argumentu je `Bool`. Návratová hodnota je seznam stejného typu jako druhý argument, protože můžeme vracet přímo druhý argument.

   Dostáváme tedy `(Num a, Ord a) => a -> [b] -> Bool -> [b]`

2. Z druhého řádku o prvním argumentu nevíme nic, o druhém víme, že je to seznam a že je stejného typu jako návratová hodnota. O třetím argumentu opět víme, že je to `Bool`

   `c -> [d] -> Bool -> [d]`

3. O argumentech nevíme nic, ale víme, že návratová hodnota je řetězec.

   `e -> f -> g -> String`

V těchto případech je vhodné nechat zatím typové proměnné v jednotlivých typech různé, abychom zabránili náhodnému propojení typů, které spolu nesouvisí.

Nyní zbývá unifikovat typy na pozicích, které si v definici funkce odpovídají.

- `a` $\sim$ `c` $\sim$ `e`, zde nesmíme zapomenout, že s `a` se pojí typový kontext. Unifikace je naopak jednoduchá, protože unifikuje jen samotné typové proměnné. Nadále budeme místo nich všech používat `a` (substituce `c` $\mapsto$ `a` a `e` $\mapsto$ `a`).
- `[b]` $\sim$ `[d]` $\sim$ `f` vyřešíme substitucí `d` $\mapsto$ `b` a `f` $\mapsto$ `[b]`.
- `Bool` $\sim$ `Bool` $\sim$ `g`, tu je substituce jednoduchá: `g` $\mapsto$ `Bool`.
- `[b]` $\sim$ `[d]` $\sim$ `String`, což už ale máme substituováno za `[b]` $\sim$ `[b]` $\sim$ `String` (protože `d` se nahradilo za `b`).

  Toto na první pohled nevypadá moc dobře, protože se zdánlivě snažíme unifikovat seznam s něčím, co není seznam. Avšak `String` je jen alias pro `[Char]`, a tedy unifikovat se seznamovými typy jej lze. Dostáváme `b` $\mapsto$ `Char`.

Nyní je třeba provést substituce v typech z jednotlivých řádků definice. Nemělo by záležet na tom, který řádek vezmeme za předpokladu, že substituujeme, dokud můžeme. Celkově dostaneme `f :: (Num a, Ord a) => a -> [Char] -> Bool -> [Char]`, neboli `f :: (Num a, Ord a) => a -> String -> Bool -> String`. Je důležité nezapomenout na kontext svázaný s typovou proměnnou `a`.

Řeš. 3.1.5 a) `cm :: (a -> [b]) -> [a] -> [b]`

Při typování rekurzivních funkcí může být výhodné dívat se nejprve na bázový příklad. V tomto případě z něj však moc nezjistíme: vidíme, že má typ `a -> [b] -> [c]`, tedy víme jen, že druhý argument je seznam a návratová hodnota je taktéž seznam. Z rekurzivní části definice pak plyne, že typ návratové hodnoty celé funkce musí být stejný jako typ návratové hodnoty funkce `f` (plyne z typu `++`), a že funkce `f` musí brát jako argumenty hodnoty ze seznamu v druhém argumentu `cm`.

Funkce je podobná funkci `map`, ale z každého prvku původního seznamu vytvoří seznam prvků a tyto seznamy spojí. Najdeme ji i mezi základními funkcemi v Haskellu pod názvem `concatMap`.

b) `mm :: Ord a => [a] -> (a, a)`

Nejprve si musíme určit typ pomocné funkce `mm'`. Z prvního řádku (který je zároveň

bází rekurze) vidíme vztah mezi jejími návratovými hodnotami a prvními dvěma argumenty a také, že třetí argument musí být seznam: `a -> b -> [c] -> (a, b)`. Z druhého řádku a z typu funkcí `min` a `max` pak vidíme, že první dva argumenty mají stejný typ, který je zároveň stejný jako typ položek v seznamu v třetím argumentu, dostáváme tedy `mm' :: Ord a => a -> a -> [a] -> (a, a)` (kontext plyne z použití `min` a `max`).
V samotné definici funkce `mm` pak není první řádek příliš zajímavý, jeho typ je `[a] -> b` (protože `error :: String -> b`). Pro typ druhého řádku už jen stačí dosadit za první dva argumenty v typu `mm'`.
Funkce `mm` počítá minimum a maximum z hodnot v seznamu a dělá to v jednom průchodu.

Řeš. 3.1.6 a) Jedná se o funkci, která dokáže (pro typy, které to umožňují) převést hodnoty daného typu na jejich textovou reprezentaci. Např. `show 42` ⇝* `"42"`, `show [16, 42]` ⇝* `"[16,42]"`. Funguje pro většinu typů s výjimkou funkčních typů. Textová forma vyprodukované `show` by typicky měla být zápis validního Haskellového výrazu.

b) Jedná se o funkci, která převádí textovou reprezentaci na hodnotu požadovaného typu. Typ výsledné hodnoty se odvodí z použití funkce `read`, v případě potřeby je možné jej vynutit explicitním otypováním:

```
(read "42" :: Int) ⇝* 42
(read "42" :: Float) ⇝* 42.0
(read "[1, 2, 3, 4]" :: [Int]) ⇝* [1, 2, 3, 4]
read "40" + read "2" ⇝* 42
```

c) Funkce pro převod celého čísla na libovolnou reprezentaci čísla, např. funkci pro výpočet e-té odmocniny čísla n, kde e je celé číslo a n je číslo s plovoucí desetinnou čárkou (např. `Double`), lze zapsat jako:

```
root :: (Floating a, Integral b) => a -> b -> a
root n e = n ** (1 / fromIntegral e)
```

(Operátor `(**)` slouží k umocňování čísel s plovoucí desetinnou čárkou.)

d) Funkce pro zaokrouhlování desetinných čísel na celá čísla (existuje i `floor` a `ceiling`). Např. `round 1.6` ⇝* `2`, `floor 1.6` ⇝* `1`. (Typová třída `RealFrac` obsahuje právě čísla, která lze zaokrouhlovat, z běžných typů do ní patří `Float` a `Double`.

Řeš. 3.2.1 a) `(+) 2 40` ⇝* `42`
Funkce, která k dané hodnotě přičte zleva číslo 2.

b) `(* 2) 21` ⇝* `42`
Funkce, která dané číslo vynásobí zprava dvěma.

c) `(- 2)` ⇝ `-2`
Číslo -2. Jelikož `-` je binární i unární operátor, nelze jej použít v pravé operátorové sekci. Místo toho však existuje funkce `subtract`: `subtract 2 44` ⇝* `42`.

d) `(2 -) 1` ⇝* `1`
Funkce, která dané číslo odečte od dvojky.

e) Syntakticky špatně utvořený výraz. Operátorové sekce musí být vždy v závorkách.

Řeš. 3.2.2

a)
```
upper' :: String -> String
upper' xs = map toUpper xs
```

b)
```
embrace' :: [String] -> [String]
embrace' xs = map ('[' :) (map (++ "]") xs)
```

Nebo lze pro `(:)` použít prefixový tvar:

```
embrace'' :: [String] -> [String]
embrace'' xs = map ((:) '[') (map (++ "]") xs)
```

c)
```
sql' :: (Ord a, Num a) => [a] -> a -> [a]
sql' xs lt = map (^ 2) (filter (< lt) xs)
```

Řeš. 3.2.3

a) První výraz je díky implicitním závorkám částečné aplikace ekvivalentní `((f 1) g) 2` a odpovídá funkci `f` beroucí tři parametry a druhý je ekvivalentní `(f 1) (g 2)`.
b) Ano, `(f 1 g) 2` ≡ `f 1 g 2` ≡ `(f 1) g 2` (tedy funkce `f` tu bere dva argumenty).
c) Ne, `(* 2) 3` ≡ `(*) 3 2` ≡ `3 * 2`. Neexistuje pravidlo, které by zaručovalo, že `3 * 2` se bude rovnat `2 * 3` (standard jazyka Haskell komutativitu operátoru `(*)` nevynucuje). Nezapomínejme, že všechny operátory definované typovými třídami můžeme předefinovat. *Poznámka*: (pokročilejší) Toto by bylo možné pouze v případě, že by komutativity vyžadovaly axiomy typové třídy, ve které je daný operátor/funkce definována. Ani to by však nezaručovalo skutečnou korektnost – interpret/kompilátor platnost axiomů nekontroluje (ani to není v jeho silách). Zůstává pouze důvěra v programátora, že jeho implementace je korektní.
d) Ano, `(* 3)` je pravá sekce.

Řeš. 3.3.1

a) `map even :: Integral a => [a] -> [Bool]`
b) `map head . snd :: (a, [[b]]) -> [b]`
c) `filter ((4 >) . maximum) :: (Ord a, Num a) => [[a]] -> [[a]]`
d) `const const :: a -> b -> c -> b`

Řeš. 3.3.2

a)
```
failing' :: [(Int, Char)] -> [Int]
failing' sts = map fst (filter ((== 'F') . snd) sts)

failing'' :: [(Int, Char)] -> [Int]
failing'' = map fst . (filter ((== 'F') . snd))
```

b)
```
embraceWith' :: Char -> Char -> [String] -> [String]
embraceWith' l r = map ((l :) . (++ [r]))
```

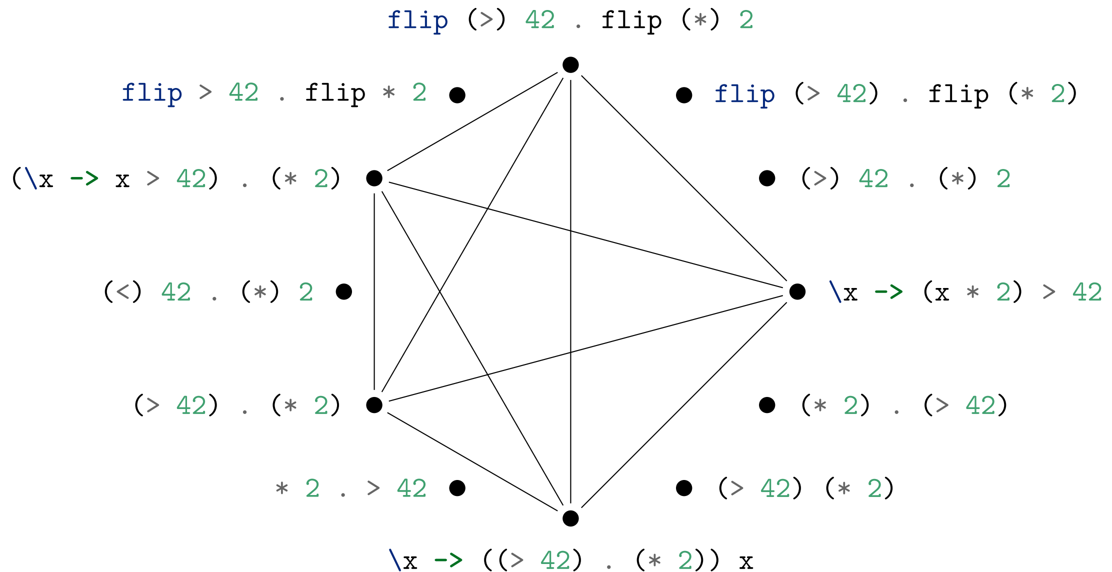
Argumenty `l` a `r` nelze rozumně odstranit.

c)
```
divisibleBy7' :: [Integer] -> [Integer]
divisibleBy7' = filter ((== 0) . (`mod` 7))
```

d)
```
letterCaesar' :: String -> String
letterCaesar' = map (chr . (3 +) . ord) . filter isLetter
```

e)
```
zp' :: (Integral a, Num b) => [a] -> [b] -> [b]
zp' = zipWith (flip (^))
```

Řeš. 3.3.3 Vzájemně ekvivalentní funkce jsou spojeny:



Všechny vzájemně ekvivalentní výrazy můžeme získat různými ekvivalentními úpravami z `\x -> (x * 2) > 42`.

Zbývající výrazy:

- `(* 2) . (> 42)` by nejprve porovnával vstup s hodnotou `42` a poté teprve přičítal `2` k výsledku typu `Bool`, je tedy typově nesprávný.
- `(> 42) (* 2)` aplikuje sekci `(> 42)` na `(* 2)`, `(> 42)` však vyžaduje na vstupu číslo, ale `(* 2)` je typu `Num a => a -> a`. Výraz je tedy typově nesprávný.
- `* 2 . > 42` je syntakticky nesprávný, operátorové sekce je vždy potřeba uzávorkovat.
- `(<) 42 . (*) 2` není nutně ekvivalentní pro všechny vstupy, protože nic negarantuje, že `(*)` je komutativní a při prohození argumentů lze zaměnit `(>)` za `(<)`. Jelikož jsou tyto funkce definovány zvlášť pro každý datový typ v dané typové třídě, je možné, že nějaká implementace toto splňovat nebude.
- `flip > 42 . flip * 2` se uzávorkuje jako `flip > ((42 . flip) * 2)`, a pokouší se tedy skládat číslo 42 a funkci `flip` a dokonce tento výsledek složení násobit dvěma. Tento výraz je tedy typově nesprávný.
- `flip (> 42) . flip (* 2)` – `flip` očekává binární funkci, ale `(> 42)` a `(* 2)` jsou nutně unární.
- `(>) 42 . (*) 2` je ekvivalentní s `\x -> 42 > (2 * x)`, argumenty jsou tedy otočeny.

Řeš. 3.3.4 a) Naším cílem je ze zadané funkce vytvořit negovanou funkci. Z typu funkce `negp` vidíme, že můžeme uvést dva argumenty – predikát a hodnotu. Pak jen na výsledek volání `f` zavoláme funkci `not`, která realizuje logickou negaci.

```
negp :: (a -> Bool) -> a -> Bool
negp f x = not (f x)
```

b) Funkci z předchozího příkladu můžeme přepsat do tvaru složení funkcí:

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

Odtud můžeme následně odstranit formální argument:

```
negp f = not . f
```

K tomuto výsledku můžeme dojít i přímo, uvědomíme-li si, že negace predikátu je složením predikátu s funkcí negace.

c) Dále lze tělo funkce přepsat do prefixového tvaru:

```
negp f = (.) not f
```

A následně lze odstranit poslední formální argument `f`, čímž dostaneme definici plně bez formálních argumentů:

```
negp = (.) not
```

Alternativně lze tělo funkce upravit pomocí operátorové sekce:

```
negp f = (not .) f
negp = (not .)
```

Poznámka: Z hlediska elegance a čistoty kódu by byla většinou programátorů v Haskellu pravděpodobně preferována varianta `negp f = not . f`.

Řeš. 3.3.6

a)
```
\x -> (f . g) x
f . g
```

b)
```
\x -> f . g x
\x -> (.) f (g x)
\x -> ((.) f . g) x
(.) f . g
```

c)
```
\x -> f x . g
\x -> (.) (f x) g
\x -> flip (.) g (f x)
\x -> (flip (.) g . f) x
flip (.) g . f
```

Řeš. 3.3.7

a)
```
(^ 2) . mod 4 . (+ 1)
\x -> ((^ 2) . mod 4 . (+ 1)) x
\x -> (^ 2) (mod 4 ((+ 1) x))
\x -> (mod 4 (x + 1)) ^ 2
```

b)
```
(+) . sum . take 10
\x -> ((+) . sum . take 10) x
\x -> (+) (sum (take 10 x))
\x y -> (+) (sum (take 10 x)) y
\x y -> sum (take 10 x) + y
```

c)
```
map f . flip zip [1, 2, 3]
\x -> (map f . flip zip [1, 2, 3]) x
\x -> map f (flip zip [1, 2, 3] x)
\x -> map f (zip x [1, 2, 3])
```

d)
```
(.)
\f g -> (.) f g
\f g -> f . g
\f g x -> (f . g) x
\f g x -> f (g x)
```

Řeš. 3.3.8

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

b)
```
f :: Num a => a -> b -> a
f x y = const (3 + x) y
f x = const (3 + x)
f x = const ((3 +) x)
f x = (const . (3 +)) x
f = const . (3 +)
```

Řeš. 3.3.9

a)
```
\_ -> x
\t -> x
\t -> const x t
const x
```

b)
```
\x -> f x 1
\x -> flip f 1 x
flip f 1
```

c)
```
\x -> f 1 x True
\x -> (f 1) x True
\x -> flip (f 1) True x
flip (f 1) True
```

d)
```
const x
```

e)
```
\x -> 0
\x -> const 0 x
const 0
```

f) Není možno převést, poněvadž `if` ... `then` ... `else` ... není klasická funkce, ale syntaktická konstrukce, podobně jako `let` ... `in` ....

g)
```
\f -> flip f x
\f -> flip flip x f
flip flip x
```

Řeš. 3.3.10

a) Postupně převádíme:

```
f1 x y z = x
f1 x y z = const x z -- přidáme z tak, abychom ho mohli odstranit
f1 x y = const x
f1 x y = const (const x) y -- přidáme y
f1 x = const (const x)
f1 x = (const . const) x
f1 = const . const
```

b)
```
f2 x y z = y
f2 x y z = const y z
f2 x = const -- eta-redukujeme obojí
f2 x = const const x -- přidáme x
```

```
f2 = const const
```

c)
```
f3 x y z = z
f3 x y z = id z -- přidáme uměle identitu
f3 x y = id
f3 x y = const id y -- přidáme y
f3 x = const id
f3 x = const (const id) x -- přidáme x
f3 = const (const id)
```

Řeš. 3.3.11 Několikrát po sobě použijeme funkci `flip`.

```
g = flip (flip ... (flip (flip f c1) c2) ... cn)
```