Cvičení 3

3.1 Částečná aplikace

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

Příklad 3.1.2 Které z následujících výrazů jsou ekvivalentní?

a) `f 1 g 2` ≡ `f 1 (g 2)`
b) `(f 1 g) 2` ≡ `(f 1) g 2`
c) `(+ 2) 3` ≡ `2 + 3`
d) `(+) 2 3` ≡ `(+ 2) 3`
e) `81 * f 2` ≡ `(*) 81 f 2`
f) `fact n` ≡ `n * fact n - 1` (uvažujíce klasickou rekurzivní definici funkce `fact`)
g) `sin (1.43)` ≡ `sin 1.43`
h) `sin 1.43` ≡ `sin 1 . 43`
i) `8 - 7 * 4` ≡ `(-) 8 (* 7 4)`

Příklad 3.1.3 Definujte *unární* funkci `nebo` pro realizaci logické disjunkce a pomocí modifikátorů `curry` a `uncurry` definujte ekvivalenci mezi vámi definovanou funkcí `nebo` a předdefinovanou funkcí `(||)`.

Příklad 3.1.4 Analogicky k funkcím `curry` a `uncurry` definujte funkce

a) `curry3 :: ((a, b, c) -> d) -> a -> b -> c -> d`
b) `uncurry3 :: (a -> b -> c -> d) -> (a, b, c) -> d`

Příklad 3.1.5 Lze funkce `curry3, uncurry3` vyjádřit pomocí funkcí `curry, uncurry`?

Příklad 3.1.6 Převeďte funkce do pointfree tvaru:

a) `\(x, y) -> x + y`
b) `\x y -> nebo (x, y)` (`nebo = uncurry (||)`)
c) `\((x, y), z) -> x + y + z` (dodržte asociativitu operátoru `+`)

Příklad 3.1.7 Zaveďme funkci `dist f g x = f x (g x)`.

a) Vyjádřete funkci `dist (curry id) id` pomocí λ-abstrakce.
b) Co dělá funkce `pair = uncurry (dist . ((.) (curry id)))`

3.2 Skládání funkcí

Příklad 3.2.1 Vyhodnoťte následující výrazy:

a) `((== 42) . (2 +)) 40`
b) `(even . (* 3) . max 4) 5`
c) `filter ((>= 2) . fst) [(1,"a"), (2,"b"), (3,"c")]`

Příklad 3.2.2 Určete všechny implicitní závorky v následujících výrazech:

a) `f.g x`
b) `f (.) g (h x) . (.) f g x`

3.3 Typování funkčních aplikací a definic

Příklad 3.3.1 Určete typy výrazů:

a) `const True`
b) `const True False`
c) `(:[])`
d) `(:[]) True`

Příklad 3.3.2 Určete typy výrazů:

a) `(&&) True`
b) `id "foo"`
c) `[]:[]:[]`
d) `([]:[]):[]`

Příklad 3.3.3 Určete typy následujících výrazů:

a) `map fst`
b) `map (filter not)`
c) `const id '!' True`
d) `fst (fst, snd) (snd, fst) (True, False)`
e) `head [head] [tail] [[]]`

Příklad 3.3.4 Určete typy funkcí:

a) `swap (x,y) = (y,x)`
b) `caar = head . head`
c) `twice f = f . f`

Příklad 3.3.5 Určete typy funkcí:

a) `cadr = head . tail`
b) `comp12 g h x y = g (h x y)`

Příklad 3.3.6 Určete typy následujících funkcí:

a)

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

b)

```
aOrX 'a' x = True
aOrX  _  x = x
```

Příklad 3.3.7 Další příklady na určovaní typu funkce:

a)

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

b)

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

c)

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

Příklad 3.3.8 Určete typy následujících výrazů:

a) `(+ 3)`
b) `(+ 3.0)`
c) `filter (>= 2)`
d) ``(> 2) . (`div` 3)``

Příklad 3.3.9 Určete typy následujících výrazů:

a) `id const`
b) `takeWhile (even . fst)`
c) `fst . snd`
d) `fst . snd . fst . snd . fst . snd`
e) `map . snd`
f) `head . head . snd`
g) `map (filter fst)`
h) `zipWith map`

Příklad 3.3.10 Definujte funkce tak, aby jejich nejobecnější typ byl shodný s typem uvedeným níže.

a) `f1 :: a -> (a -> b) -> (a, b)`
b) `f2 :: [a] -> (a -> b) -> (a, b)`
c) `f3 :: (a -> b) -> (a -> b) -> a -> b`
d) `f4 :: [a] -> [a -> b] -> [b]`
e) `f5 :: ((a -> b) -> b) -> (a -> b) -> b`
f) `f6 :: (a -> b) -> ((a -> b) -> a) -> b`

Příklad 3.3.11 Proč jsou první dva výrazy v pořádku (interpret je akceptuje), třetí však nikoli?

- `id id`
- `let f x = x in f f`
- `let f x = x x in f id`

Příklad 3.3.12 Určete typ funkcí `f1` až `f6` v následujících výrazech. Jestli se funkce vyskytuje ve vícero výrazech/výskytech, určete její typ jednak pro každý výraz/výskyt samostatně, a také pak unifikujte vzniklé typy (tj. zohledněte omezení na typ ze všech výrazů/výskytů).

a)
```
f1 []
snd (f1 [id])
```
b)
```
fun t = f2 ((x:y):(z:q), t)
flip (curry f2)
```
c)
```
fun s = f3 (fst f3 s + 10)
```
d)
```
(,) 1 x : f4
head f4 `elem` ['a'..'z']
```
e)
```
f5 []
1 + f5 [x:xs]
```
f)
```
f6 4
id flip f6 id
```

Příklad 3.3.13
Jaký typ má výraz `filter ((4 >) . maximum)`?

Řešení

Řešení 3.1.1 Funkce `min` vrací menší z dvou argumentů. Tedy máme dva případy. Když druhý argument bude menší než 6, výsledkem funkce bude tento argument. V opačném případě, když druhý argument bude alespoň 6, výsledkem bude 6 jako to menší z dvojice čísel.

```
min6 :: (Num a, Ord a) => a -> a
min6 x = if x < 6 then x else 6
```

Typ funkce `min6` je poněkud složitější. Jeho význam není teď důležitý a bude vysvětlen později.

Řešení 3.1.2

a) Ne, netřeba se nechat zmást konkrétními hodnotami a intuicí, které mohou nabádat k odpovědi ano. První výraz je díky implicitním závorkám částečné aplikace ekvivalentní `((f 1) g) 2` a odpovída 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`.
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 operátora `(+)` nevynucuje). Nezapomínejme, že všechny operátory můžeme předefinovat. *Poznámka*: (pokročilejší) Toto by bylo možné pouze v případu, že by komutativity vyžadovali axiomy typové třídy, ve které je daný operátor/funkce definována. Ani to by však nezaručovalo skutečnou korektnost – interpret/kompilátor platnost axiom nekontroluje (ani to není v jeho silách). Zůstává pouze důvěra v programátora, že jeho implementace je korektní.
d) Ne, opět není možné přehodit parametry operátoru `+`.
e) Ne, je nutné uzávorkovat druhý argument.

   `81 * f 2` ≡ `81 * (f 2)` ≡ `(*) 81 (f 2)`

f) Ne, je nutné přidat závorku k argumentu na konci.

   `fact n` ⇝ `n * fact (n - 1)`

g) Ano (použít závorky v tomto případě není nutné).
h) Ne, protože `.` ve výrazu `sin 1 . 43` je operátor (skládání funkcí), zatímco ve výrazu `sin 1.43` se jedná o desetinnou tečku.
i) Ne, `8 - 7 * 4` ≡ `(-) 8 ((*) 7 4)`.

Řešení 3.1.3

```
nebo :: (Bool, Bool) -> Bool
nebo (x, y) = x || y

curry nebo ≡ (||)
uncurry (||) ≡ nebo
```

Řešení 3.1.4

```
curry3 :: ((a, b, c) -> d) -> a -> b -> c -> d
curry3 f x y z = f (x, y, z)
```

```
uncurry3 :: (a -> b -> c -> d) -> (a, b, c) -> d
uncurry3 f (x, y, z) = f x y z
```

Řešení 3.1.5 Ne. Dvouargumentové funkce pracují s uspořádanými dvojicemi a tříargumentové funkce s uspořádanými trojicemi. Uspořádané dvojice a trojice však mezi sebou nemají žádný speciální vztah.

Řešení 3.1.6

a)

```
\(x, y) -> x + y
\(x, y) -> (+) x y
\(x, y) -> uncurry (+) (x, y)
uncurry (+)
```

b)

```
\x y -> nebo (x, y)
\x y -> curry nebo x y
curry nebo
```

c)

```
\((x, y), z) -> x + y + z
\((x, y), z) -> (x + y) + z
\((x, y), z) -> ((+) x y) + z
\((x, y), z) -> (uncurry (+) (x, y)) + z
\((x, y), z) -> (+) (uncurry (+) (x, y)) z
\((x, y), z) -> ((+) . uncurry (+)) (x, y) z
\((x, y), z) -> uncurry ((+) . uncurry (+)) ((x, y), z)
uncurry ((+) . uncurry (+))
```

Řešení 3.1.7

a)

```
dist (curry id) id
\x -> dist (curry id) id x
\x -> (curry id) x (id x)
\x -> ((\f x y -> f (x, y)) id) x x
\x -> (\f x y -> f (x, y)) id x x
\x -> id (x, x)
\x -> (x, x)
```

b)

```
uncurry (dist . ((.) (curry id)))
(\f (x, y) -> f x y) (dist . ((.) (curry id)))
\(u, v) -> (\f (x, y) -> f x y) (dist . ((.) (curry id))) (u, v)
\(u, v) -> (dist . ((.) (curry id))) u v
\(u, v) -> ((dist . ((.) (curry id))) u) v
```

```
\(u, v) -> (dist (((.) (curry id)) u)) v
\(u, v) -> dist (((.) (curry id)) u) v
\(u, v) w -> dist (((.) (curry id)) u) v w
\(u, v) w -> (\f g x -> f x (g x)) (((.) (curry id)) u) v w
\(u, v) w -> (((.) (curry id)) u) w (v w)
\(u, v) w -> (.) (curry id) u w (v w)
\(u, v) w -> ((.) (curry id) u w) (v w)
\(u, v) w -> ((curry id . u) w) (v w)
\(u, v) w -> (curry id . u) w (v w)
\(u, v) w -> ((\f x y -> f (x, y)) id . u) w (v w)
\(u, v) w -> ((\x y -> (x, y)) . u) w (v w)
\(u, v) w -> (((\x y -> (x, y)) . u) w) (v w)
\(u, v) w -> ((\x y -> (x, y)) (u w)) (v w)
\(u, v) w -> (\x y -> (x, y)) (u w) (v w)
\(u, v) w -> (u w, v w)
```

Řešení 3.2.1

a) Intuitivně se výraz vyhodnocuje tak, že postupně aplikujeme skládané funkce odzadu, výsledek je tedy `True`. Po krocích můžeme výraz vyhodnotit takto (s ohledem na definici `(.) f g x = f (g x)`):

`((== 42) . (2 +)) 40` $\rightsquigarrow$ `(== 42) ((2 +) 40)` $\rightsquigarrow$ `(== 42) 42` $\rightsquigarrow$ `True`

b)

```
(even . (* 3) . max 4) 5
    ≡  (even . ( (* 3) . max 4 )) 5
    ⇝ even ( ((* 3) . (max 4) 5 )
    ⇝ even ((* 3) (max 4 5)) ⇝ even ((* 3) 5)
    ⇝ even 15

     ⇝ False
```

c) Filtrujeme seznam funkcí `((>= 2) . fst)`, která očekává dvojice a rozhoduje, zda je první složka této dvojice větší než 2 (první složka tedy musí být číslo, druhá může být cokoli).
Aplikací této funkce na náš seznam tedy dostaneme seznam těch hodnot, které mají první složku větší nebo rovnou 2, tedy

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

Řešení 3.2.2

a) `f . (g x)`
b) `(((f (.)) g) (h x)) . ((((.) f) g) x)`

Řešení 3.3.1

a) `const :: a -> b -> a`, `True :: Bool`. Reálný parametr má konkrétnější typ – substituce `a` $\sim$ `Bool`, tedy celkový typ je `const True :: b -> Bool`.

b) Obdobně jako v předchozím případě, jen navíc aplikujeme na `False`, tedy substituce `b` ∼ `Bool`. Výsledek je `const True False :: Bool`.
c) `(:[])` je pravá operátorová sekce operátoru `(:) :: a -> [a] -> [a]`. Typ reálného argumentu `[] :: [a]` souhlasí s typem formálního argumentu, můžeme tedy aplikovat (opět druhý argument). Výsledný typ je tedy `(:[]) :: a -> [a]`.
d) `(:[]) :: a -> [a]`, `True :: Bool`. Typ reálného argumentu je konkrétnější, substituujeme `a` ∼ `Bool`, výsledný typ je `(:[]) True :: [Bool]`.

Řešení 3.3.2

a) Typy základních podvýrazů jsou `(&&) :: Bool -> Bool -> Bool` a `True :: Bool`. Typ reálného prvního argumentu funkce `(&&)` souhlasí s typem prvního argumentu v typové deklaraci funkce, tedy `(&&)` lze aplikovat na parametr `True`, čímž se tento parametr naváže a výsledná funkce je typu `(&&) True :: Bool -> Bool`.
b) `id :: a -> a`, `"foo" :: String`. Typ prvního reálného parametru je konkrétnější, než typ formálních parametrů v deklaraci, tedy substituujeme `a` ∼ `String`. Po aplikaci na jediný parametr nám vychází typ `id "foo" :: String`.
c) `(:) :: a -> [a] -> [a]` sdružuje zprava, tedy výraz odpovídá seznamu `[ [], [] ]`. Jeho oba prvky jsou typu `[] :: [a]`, a tedy seznam je homogenní, a tedy otypovatelný. Výsledný typ je `[]:[]:[] :: [[a]]`.
d) Můžeme zapsat jako seznam `[[[]]]`, což odpovídá typu `[[[]]] :: [[[a]]]`.

Řešení 3.3.3

a) Podvýrazy jsou typů `map :: (a -> b) -> [a] -> [b]` a `fst :: (c, d) -> c` (je nutné volit různé typové proměnné v různých podvýrazech, abychom nedostali do výpočtu závislosti, které tam nemají být).
Nyní musíme unifikovat typ prvního parametru v typu funkce `map`, tedy `(a -> b)` s typem skutečného prvního parametru: `a -> b` ∼ `(c, d) -> c`. Jedná se o funkční typ, tedy unifikujeme první parametr levé strany s prvním parametrem na pravé straně a tak dále. Tím dostáváme substituci `a` ∼ `(c, d)` a `b` ∼ `c` a budeme dosazovat pravou stranu do levé, protože pravá strana je specifičtější typ.
Typ funkce `map` v tomto výrazu je tedy `map :: ((c, d) -> c) -> [(c, d)] -> [c]`. Nyní již můžeme funkci `map` dosadit první parametr a dostat typ celého výrazu: `map fst :: [(c, d)] -> [c]`, tedy naše funkce bere seznam dvojic a vrací seznam obsahující první složky těchto dvojic.
b) Typy podvýrazů jsou: `map :: (a -> b) -> [a] -> [b]`, `filter :: (c -> Bool) -> [c] -> [c]` a `not :: Bool -> Bool`.
Nejprve musíme otypovat podvýraz `filter not` a podle jeho typu potom určit typ celého výrazu. Unifikujeme tedy typ prvního parametru v definici `filter` s reálným typem prvního parametru: `c -> Bool` ∼ `Bool -> Bool`, a tedy `c` ∼ `Bool`. Po dosazení za `c` tedy dostaneme typ aplikace `filter not :: [Bool] -> [Bool]`.
Nyní unifikujeme typ prvního parametru funkce `map` s typem `filter not`: `a -> b` ∼ `[Bool] -> [Bool]`, a tedy `a` ∼ `[Bool]`, `b` ∼ `[Bool]`.
Dosazením do typu funkce `map` a aplikací dostáváme `map (filter not) :: [[Bool]] -> [[Bool]]`.
c) `const :: a -> b -> a`, `id :: c -> c` (nutno zvolit různé typové proměnné v různých

výrazech), `'!' :: Char`, `True :: Bool`. Argumenty dosazujeme postupně a substituujeme:

- pro `const id`: substituce `a` $\sim$ `c -> c`, dosazujeme konkrétnější do obecnějšího a dostáváme typ aplikace `const id :: b -> c -> c`
- dále aplikujeme `const id` na `'!'`, substituce `b` $\sim$ `Char`, výsledek `const id '!' :: c -> c`
- aplikujeme `const id '!'` na `True`, substituce `c` $\sim$ `Bool`, konečný výsledek `const id '!' True :: Bool`.

d) V tomto případě použijeme alternativní pohled na typování, kdy si výraz zkusíme vyhodnotit, a podle toho určit jeho typ. Nejprve připomeneme, jak je tento výraz implicitně uzávorkovaný: `((fst (fst, snd)) (snd, fst)) (True, False)` a nyní vyhodnocujeme:

```
((fst (fst, snd)) (snd, fst)) (True, False)
    ⇝ (fst (snd, fst)) (True, False)
    ⇝ snd (True, False)
    ⇝ False
```

A tedy nám vychází typ `((fst (fst, snd)) (snd, fst)) (True, False) :: Bool`. Zde je však třeba být opatrný – pokud by výsledný typ mohl být polymorfní, je jistější udělat si všechny typové substituce.

e) Opět nejprve zkusíme výraz vyhodnotit:

```
head [head] [tail] [[]]
    ⇝ head [tail] [[]]
    ⇝ tail [[]]
    ⇝ []
```

Zdálo by se tedy, že výsledek je typu `[] :: [t]`. To však není pravda, protože typ výsledku je ovlivněn všemi substitucemi, které nastaly při typování výrazu. Proto musíme výraz s polymorfním návratovým typem skutečně otypovat:
`head :: [a] -> a`, `[head] :: [[b] -> b]`, `[tail] :: [[c] -> [c]]`, `[[]] :: [[d]]`.
V podvýrazu `head [head]` unifikujeme `[a]` $\sim$ `[[b] -> b]`, a tedy `a` $\sim$ `[b] -> b`, tudíž `head [head] :: [b] -> b`, a tedy jej lze dále aplikovat na `[tail] :: [[c] -> [c]]` se substitucí `[b]` $\sim$ `[[c] -> [c]]`, a tedy `b` $\sim$ `[c] -> [c]`. Dostáváme `head [head] [tail] :: [c] -> [c]`.
Tento výraz je nyní aplikován na výraz `[[]] :: [[d]]`, což znamená unifikaci `[c]` $\sim$ `[[d]]`, a tedy `c` $\sim$ `[d]`. Správný výsledný typ je tedy `head [head] [tail] [[]] :: [[d]]`, což je typ seznamu seznamů, a tedy různý od `[t]`, který jsme odhadly dříve (a je moc obecný).

Řešení 3.3.4

a) Funkci lze jednoduše intuitivně otypovat, protože vidíme, že ve výsledku jenom prohodí argumenty uspořádané dvojice. Tedy vstupní typ `(a, b)` převede na typ `(b, a)`. Platí tedy `swap :: (a, b) -> (b, a)`.

b) Víme, že typ funkce `head` je `[a] -> a`, což v dvojnásobné aplikaci znamená, že vstup musí být typu `[[a]]` a výstup typu `a`. Výsledkem je tedy `[[a]] -> a`.

c) V tomto případě vidíme, že `twice` vytvoří dvojitou aplikaci zadané funkce. Tento případ možná vypadá stejně jako předchozí, avšak je v něm významný rozdíl v tom, že zatímco dva výskyty funkce `head` byly zcela nezávislé, a tedy mohli mít odlišně specializovaný typ, `f` je fixována formálním argumentem funkce `twice` a musí mít v obou výskytech stejný typ. Avšak vidíme, že typ vstupu musí být stejný jako typ výstupu, a tedy `f :: a -> a`. Ve výsledku tedy máme `twice :: (a -> a) -> a -> a`.

Řešení 3.3.5

a) Opět otypujeme intuitivně. Víme, že `tail :: [a] -> [a]` a `head :: [a] -> a`. Pozor! Normálně je takovéto východiskové otypování cestou k záhubě. Při otypovávání funkcí/výrazů/proměnných je vždy potřeba použít nové, čerstvé typové proměnné. Jinak zavedeme nežádoucí a nepravdivou rovnost mezi typy, která způsobí, že určený výsledný typ výrazu nebude správný (nebude dostatečně obecný, případně nebude možné výraz vůbec otypovat). V tomto případě si to můžeme dovolit, protože tam rovnost je (vstup `head` je výstupem `tail`). Argument, který vstoupí do funkce `cadr`, je tedy typu `[a]`, protože to vyžaduje funkce `tail`. Z ní dostaneme hodnotu opět typu `[a]`, a ten dáme jako argument funkci `head`, načež dostaneme hodnotu typu `a`. Tedy ve výsledku máme typ `[a] -> a`.

b) Vidíme, že `g` a `h` jsou funkce, tedy nechť `g :: a -> b, h :: c -> d -> e`. Na základě shody typů díky aplikaci funkce na argumenty také vidíme, že `x :: c, y :: d, a` ∼ `e`. Další typová omezení už nejsou. Funkční typ, který budeme hledat, sestává z typů `g, h, x, y` a typu těla definice `comp12`. Ve výsledku tedy dostaneme `(a -> b) -> (c -> d -> a) -> c -> d -> b`. `comp12 :: (a -> b) -> (c -> d -> a) -> c -> d -> b`.

Řešení 3.3.6

a) 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 :: [t] -> String`.

b) Z použitých vzorů můžeme odvodit, že funkce bere dva argumenty a že první je typu `Char`. Z návratové hodnoty prvního vzoru 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`

Řešení 3.3.7

a) Z použitých vzorů můžeme odvodit, že funkce bere dva argumenty, první typu `Bool` a druhý je dvojice. Uvažujme tedy, že bude mít typ `(a, b)`.

   Z toho tedy usoudíme na typy argumentů `x :: a`, `y :: b`. Nyní můžeme odvozovat typy výrazů na pravé straně definice: `(y, x) :: (b, a)` a `(x, y) :: (a, b)`.

   Avšak návratový typ funkce musí být jednoznačný, a tedy oba typy si musí odpovídat: `(b, a)` ∼ `(a, b)`, z čehož vidíme, že oba prvky dvojice musí být stejného typu.

   Celkový typ je tedy `mswap :: Bool -> (a, a) -> (a, a)`.

b) Funkci není možné otypovat, protože podle prvního vzoru je parametrem dvojice, podle druhého trojice a podle třetího čtveřice. Tyto tři typy však nejsou vzájemně unifikovatelné.

c) Funkce je syntakticky špatně zapsaná, protože jednotlivé definice mají různý počet argumentů. Nelze ji tedy otypovat.

Řešení 3.3.8

a) Číselný literál může být libovolného numerického typu, tedy `3 :: Num a => a`, `(+) :: Num b => b -> b -> b`. Dostáváme `Num a => a` $\sim$ `Num b => b`, z čehož dostáváme `a` $\sim$ `b` a typový kontext se nemusí rozšiřovat. Celkově tedy `(+ 3) :: Num a => a -> a`

b) Desetinný číselný literál je typu `3.0 :: Fractional a => a`, `(+) :: Num b => b -> b -> b`. Dostáváme tedy unifikaci `Num b => b` $\sim$ `Fractional a => a`, z čehož dostáváme `a` $\sim$ `b`, avšak zároveň nesmíme zapomenout na to, že obě proměnné mají nyní oba typové kontexty. Celkový typ je tedy `(+ 3.0) :: (Fractional a, Num a) => a -> a`.

Poznámka: Ve skutečnosti je ve standardní knihovně řečeno, že každý typ, který splňuje `Fractional`, nutně splňuje i `Num`, a tedy lze `Num` v tomto případě vynechat: `(+ 3.0) :: Fractional a => a -> a`. Jeho nevynechání však není chyba (nicméně interpretr jej automaticky vynechává).

c) Typy použitých podvýrazů jsou: `filter :: (a -> Bool) -> [a] -> [a]`, `(>=) :: Ord b => b -> b -> Bool`, `2 :: Num c => c`.

Nejprve určeme typ `(>= 2)`. Aplikací `2` jako druhého argumentu dostáváme unifikaci `Ord b => b` $\sim$ `Num c => c`. Pro typ `(>= 2)` tedy dostáváme dosazením do `Ord b => b -> b -> Bool` typ `(Ord b, Num b) => b -> Bool`.

Teď určíme typ celého výrazu. Aplikace funkce `filter` na `(>= 2)` nám vynucuje `a -> Bool` $\sim$ `(Ord b, Num b) => b -> Bool`. Tedy `a` $\sim$ `b` (plus typové kontexty). Hledaným typem je `[a] -> [a]`, což dosazením na základě výše určených informací dává výsledný typ `(Num a, Ord a) => [a] -> [a]`.

d) Typy použitých podvýrazů jsou: `2 :: Num a => a`, `(>) :: Ord b => b -> b -> Bool`, `div :: Integral c => c -> c -> c`, `3 :: Num d => d`, `(.) :: (f -> g) -> (e -> f) -> e -> g`.

Z aplikace `(> 2)` dostáváme unifikaci `Num a => a` $\sim$ `Ord b => b`, celkově je tedy výraz typu `(> 2) :: (Num a, Ord a) => a -> Bool`.

Z aplikace ``(`div` 3)`` dostáváme `Integral c => c` $\sim$ `Num d => d`, a tedy ``(`div` 3) :: (Integral c, Num c) => c -> c``.

Nyní, z použití ve skládání funkcí, dostáváme unifikaci `(Num a, Ord a) => a -> Bool` $\sim$ `(f -> g)`, a tedy `(Num a, Ord a) => a` $\sim$ `f`, `Bool` $\sim$ `g`. Dále potom `(Integral c, Num c) => c -> c` $\sim$ `e -> f`, a tedy `(Integral c, Num c) => c` $\sim$ `e` $\sim$ `f`. Nyní máme pro `f` dvě unifikace a musíme je dát dohromady: `(Integral c, Num c) => c` $\sim$ `(Num a, Ord a) => a` $\sim$ `f`.

Celkově dostaneme typ odpovídající `e -> g` (z definice `(.)`). Pro jednoduchost bereme lexikograficky první proměnnou, pokud může unifikace probíhat oběma směry:
``(> 2) . (`div` 3) :: (Num a, Integral a, Ord a) => a -> Bool``.

Poznámka: To lze dále zjednodušit (protože `Integral` vynucuje `Num` a `Ord`) na ``(> 2) . (`div` 3) :: Integral a => a -> Bool``

Řešení 3.3.9

a) Výraz `id const` se vyhodnotí na výraz `const` a má tedy stejný typ.
`id const :: a -> b -> a`

b) Funkce `takeWhile` musí dostat 2 parametry – funkci a seznam. Jelikož na prvky seznamu se bude aplikovat funkce `(even . fst)`, musí být tyto prvky uspořádané dvojice (aby bylo možno aplikovat na ně `fst`), jejichž první složka musí být celé číslo (přesněji být v typové třídě `Integral`, aby bylo možno na ni aplikovat `even`). Víme, že `takeWhile` vrací seznam stejného typu, jako seznam, který bere.
`takeWhile (even . fst) :: Integral a => [(a, b)] -> [(a, b)]`

c) Na vstupu musí být uspořádaná dvojice (abychom mohli aplikovat `snd`), druhou složkou které musí být opět uspořádaná dvojice (abychom pak mohli aplikovat `fst`).
`fst . snd :: (a, (b, c)) -> b`

d) Tenhle případ je analogický předešlému, jenom má více stupňů.
`fst . snd . fst . snd . fst . snd :: (a,((b,((c,(d,e)),f)),g)) -> d`

e) Na první argument budeme nejdříve aplikovat funkci `snd`, musí tedy jít o uspořádanou dvojici. Druhou složkou této dvojice musí být unární funkce, jelikož ta je použita jako první argument funkce `map`. Druhým argumentem je pak seznam, jehož prvky mají stejný typ, jaký vyžaduje zmíněná unární funkce. Výsledkem bude seznam prvků po aplikaci této unární funkce.
`map . snd :: (a, b -> c) -> [b] -> [c]`

f) Opět podobná úvaha: musí jít o uspořádanou dvojici, kde na druhou složku je možno aplikovat funkci `head` (je to tedy seznam). Na jeho první prvek je opět možné aplikovat `head`, prvky tohoto seznamu jsou tedy opět seznamy.
`head . head . snd :: (a, [[b]]) -> b`

g) Výraz vezme seznam a vrátí seznam, kde na každý prvek bude aplikována funkce `filter fst`. Tyto prvky musí být tedy opět seznamy (protože se na ně aplikuje `filter` podle predikátu `fst`). Prvky těchto vnitřních seznamů pak musí být uspořádané dvojice, jelikož na ně aplikujeme `fst`. A první složkou musí být `Bool`, protože ta je použita přímo jako predikát funkce `filter`.
`map (filter fst) :: [[(Bool, a)]] -> [[(Bool, a)]]`

h) Výraz vezme dva seznamy a vrátí třetí seznam (vychází z typu funkce `zipWith`). Prvek prvního a prvek druhého seznamu musí tvořit vhodné argumenty pro spojovací funkci `map`. První seznam tedy obsahuje nějaké unární funkce a druhý seznamy, kterých prvky je možno zpracovávat těmito unárními funkcemi. Výsledky aplikací zmíněných funkcí na seznamy v druhém seznamu jsou vráceny ve formě seznamu.
`zipWith map :: [a -> b] -> [[a]] -> [[b]]`

Řešení 3.3.10

a)

```
f1 :: a -> (a -> b) -> (a, b)
f1 x g = (x, g x)
```

b)

```
f2 :: [a] -> (a -> b) -> (a, b)
f2 s g = (h, g h) where h = head s
```

c)

```
f3 :: (a -> b) -> (a -> b) -> a -> b
f3 f g x -> head [f x, g x]
```

d)

```
f4 :: [a] -> [a -> b] -> [b]
f4 = zipWith (flip id)
```

e)

```
f5 :: ((a -> b) -> b) -> (a -> b) -> b
f5 g f  = head [g f, f undefined]
f5' g f = head [g f, f arg]
    where arg = arg
```

f)

```
f6 :: (a -> b) -> ((a -> b) -> a) -> b
f6 x y = x (y x)
```

Řešení 3.3.11 U prvního výrazu je každé `id` samostatnou instancí, tedy má svůj vlastní typ: první je typu `(a -> a) -> a -> a`, zatímco druhé je typu `b -> b`. Podobná situace je u druhého výrazu, jenom místo `id` používáme pro stejnou funkci pojmenování `f`.

Ve třetím výrazu je `id` argumentem s formálním jménem `x`, který však musí mít jeden konkrétní typ. Problém nastává v určování typu výrazu `f: f x = x x` – uvažujme, že `x :: a`. Aplikace na pravé straně nás ale nutí specializovat, tedy `x :: a1 -> a2`. Jelikož je však `x` aplikováno samo na sebe, dostáváme typovou rovnost `a1 = a1 -> a2`, která vytváří nekonečný typ. Třetí výraz tedy není otypovatelný, a tudíž ani korektní.

Řešení 3.3.12 Nejdříve jsou uvedeny typy jednotlivých výrazů (případně výskytů požadované funkce), na posledním řádku za implikační šipkou se pak nachází výsledný typ, tedy typ průniku předchozích. Typové proměnné v jednotlivých řádcích spolu nejsou nijak provázané.

a)

`[a] -> b`
`[a -> a] -> (b, c)`
⇒ `[a -> a] -> (b, c)`

b)

`([[a]], b) -> c`
`(a, b) -> c`
⇒ `([[a]], b) -> c`

c)

`(Num a) => a -> b`
`(Num c) => (a -> c, b)`
⇒ Typy jsou nekompatibilní, průnik je prázdný.

d)

```
(Num a) => [(a, b)]
[Char]
```

⇒ Typy jsou nekompatibilní, průnik je prázdný.

e)

```
[a] -> b
(Num b) => [[a]] -> b
```

⇒ `(Num b) => [[a]] -> b`

f)

```
(Num a) => a -> b
a -> (b -> b) -> c
```

⇒ `(Num a) => a -> (b -> b) -> c`

Řešení 3.3.13 Určíme si typy jednotlivých podvýrazu:

- Operátorová sekce `(4 >)` bere parametr, který musí být numerického typu protože typ musí být shodný s typem prvního argumentu, kterým je číslo 4(dostáváme `Num a`). Také musí být srovnatelného typu odvozeného od operátora `(>)` (dostávame `Ord a`). Výsledný typ tedy je: `(Num a, Ord a) => a -> Bool`.
- Funkce `maximum` vrátí maximálny prvek ze seznamu. Prvky seznamu tedy musí splňovať srovnatelnost aby bolo možné určit maximální prvek - musí patřit do typové třídy `Ord`. Dostávame tedy typ `Ord b => [b] -> b`
- `(.) :: (d -> e) -> (c -> d) -> c -> e`
- `filter :: (f -> Bool) -> [f] -> [f]`

Rovnosti: `f = [b], a = b, c = [b], e = a = b, d = b`