Interludium: Psaní hezkého kódu

Zanedlouho budou pocvikové odpovědníky vystřídány velkými úlohami. Ty budou kromě automatického testování hodnoceny ještě cvičícími na kvalitu kódu. Co činí kód kvalitním, není snadné vyjádřit, a ani to není ambicí této vsuvky. Ta vám ale může pomoci vyhnout se nedostatkům, s nimiž se při opravování úloh setkáváme nejčastěji.

Takřka ke každému z pravidel níže by se daly najít výjimky. Jako základní přehled tzv. *antipatternů* by vám tento seznam měl ale posloužit dobře a určitě nic nezkazíte, pokud si před odevzdáním svoje řešení ještě jednou projdete a ujistíte se, že se žádných ze zde uvedených poklesků nedopouštíte. Naopak, pokud některý z těchto vzorů ve svém kódu máte, je celkem vysoká šance, že vám jej opravující vyčte.

Typ `Bool` a výrazy `if`

Používat při rozhodování podmíněný výraz je zcela přirozené, takže si ani nemusíme všimnout, že se rozhodujeme mezi dvěma logickými hodnotami. Takové situace se ale často dají elegantněji řešit použitím logických spojek, a to zejména tehdy, je-li alespoň jednou z větví `if`u konstanta `True` nebo `False`.

Nechť `p` a `r` jsou výrazy typu `Bool`. Následující konstrukce jsou považovány za nečitelné a je žádoucí je nahradit, neboť jen komplikovaným způsobem popisují samotné `p` (nebo `not p`):

- Výraz tvaru `p == True` je možné nahradit výrazem `p`.
- Podobně `p == False` → `not p`.
- `if p then True else False` → `p`.
- `if p then False else True` → `not p`.

Také ve většině případů následujícího tvaru bude použití logické spojky čitelnější:

- `if p then r else False` → `p && r`.
- `if p then True else r` → `p || r`.
- `if p then False else r` → `not p && r`.
- `if p then r else True` → `not p || r`.

Jiným zneužitím `if`u je jeho použití jako v imperativních jazycích, kde je to podmíněný *příkaz*. V Haskellu se ale jedná o *výraz* (a odpovídá třeba operátoru `?:` z jazyka C nebo výrazu `t if p else e` v jazyce Python), který nemusí být nejvnějšnějším výrazem funkce. Umožňuje nám to použít ho třeba jako parametr jiné funkce. Pokud jsou obě větve `if`u hodně podobné, zkuste společný kód „vytknout“ mimo něj:

- `if p then f x z else f y z` → `f (if p then x else y) z`

Je-li takto přepsaný kód dlouhý a nepřehledný (což dost možná bude), můžete čitelnosti výrazně pomoci přesunutím podmíněného výrazu do lokální definice:

```
let w = if p then x else y
in f w z      -- samozřejmě s lepším jménem než ilustračním ‚w‘
```

Takovéto případy se naštěstí v kódu snadno detekují a snadno přepisují.

Používání `if` a pomocných funkcí místo vzorů

V rekurzi je vhodné oddělit bázový případ do samostatných definičních řádků (pokud to jde). Je to výrazně přehlednější než používat `if`. Uvažme následující funkci:

```
digitsSum :: Integral i => i -> i
digitsSum x = if x == 0 then 0 else x `mod` 10 + digitsSum (x `div` 10)
```

Zde je jediný bázový případ, bylo by tedy lepší funkci přepsat takto:

```
digitsSum' :: Integral i => i -> i
digitsSum' 0 = 0
digitsSum' x = x `mod` 10 + digitsSum' (x `div` 10)
```

Výhodou tohoto zápisu je, že je na první pohled vidět, co je báze a co je rekurzívní část funkce.

Obdobně **při práci se seznamy je vhodné je dekomponovat pomocí vzorů**. Tedy například místo

```
listSum :: Num n => [n] -> n
listSum [] = 0
listSum x  = head x + listSum (tail x)
```

je lepší použít vzor pro neprázdný seznam:

```
listSum' :: Num n => [n] -> n
listSum' []     = 0
listSum' (x:xs) = x + listSum xs
```

O to více je tento přístup důležitý, když například **zkombinujeme seznamy a ntice**. Uvažme:

```
larger :: Ord a => [(a, a)] -> [a]
larger [] = []
larger xs = max (fst (head xs)) (snd (head xs)) : larger (tail xs)
```

Tato funkce je již poměrně nepřehledná, lze ji přitom napsat i velmi krátce:

```
larger' :: Ord a => [(a, a)] -> [a]
larger' [] = []
larger' ((a, b) : xs) = max a b : larger' xs
```

To je lepší a (po troše tréningu na vzory) přehlednější. *Poznámka*: tuto konkrétní funkci by bylo ještě lepší vyřešit pomocí `map`:

```
larger'' :: Ord a => [(a, a)] -> [a]
larger'' xs = map (\(x, y) -> max x y) xs
```

Vymýšlení kola

Nedostatky z této kategorie jsou obtížnější na detekci, neboť již vyžadují nějaký přehled o tom, jaké funkce balík `base` nabízí. Rozhodně se vám tak vyplatí se alespoň přibližně seznámit s poskytovanými funkcemi v dokumentaci. Často se může hodit zeptat se: „není tohle natolik základní funkcionalita, že by mohla být v `Prelude` nebo `Data.List`?“. Pokud jste se s ním doposud nesetkali, `Data.List` je modul, který, jak název napovídá, obsahuje užitečné funkce pro práci se seznamy. Máte-li již zkušenost s jinými programovacími jazyky, může být dobrým vodítkem i to, co nabízí v základu ony.

- **Používání explicitní rekurze místo použití funkcí `map`, `filter`, `zipWith`, `all`, `and` apod.** – výrazně zhoršuje čitelnost, protože na první pohled je jasné jen to, že funkce pracuje se seznamem. Naproti tomu použití těchto standardních funkcí spolu s dobře pojmenovanou pomocnou funkcí nebo krátkou a čitelnou lambdou umožňuje zápis přečíst téměř přirozeně.

  – Pokud potřebujete nějaký seznam zároveň protřídit i transformovat, můžete použít skládání `map` a `filter`, ale ještě čitelnější může být třeba použití intensionálního zápisu seznamu, pokud už jste se s nimi seznámili.

- **Definování vlastních verzí funkcí jako `fst`, `snd`, `last`, `replicate`, `elem`, `take` apod.** – zhoršuje čitelnost ostatními (tj. v tomto případě cvičícími), přidává vám práci a zvyšuje riziko zanesení chyby.

- **Nevytáhnutí duplicitního nebo podobného kódu do pomocné funkce** – tento bod je součástí širšího a složitějšímu problému, kterým je vhodná dekompozice problému na podproblémy. Na tomto místě se dá jen doporučit, že pokud se vám zdá, že nějaký kousek kódu nebo konstrukci píšete častěji než jednou (tím spíš pokud nějaký kód kopírujete), může to být indikátor toho, že by se ten kód dal vytáhnout do pomocné nebo obecnější funkce. Její vhodné pojmenování navíc opět umožní čtenáři pochopit myšlenku řešení rychleji.

- **Konkrétní výrazy, kterých se dá elegantně zbavit:**

  – `s == []` nebo `length s == 0` → `null s` – tyto tři způsoby dokonce ani nejsou ekvivalentní: první zavádí omezení, že rovnítko musí dávat smysl i pro prvky seznamu (tedy musí být v typové třídě `Eq`), druhý bude na dlouhých seznamech pomalý (a na nekonečných bude cyklit). Jedině `null` nebo použití vzoru je ta správná volba.

  – `(n `mod` 2) == 0` → `even n`, obdobně pro `odd n`

  – `all (== True)` → `and`, obdobně pro `any (== True)` → `or`.

Nadbytečná výřečnost

Programy v Haskellu je často možné napsat velice úsporně a zároveň čitelně. Je pochopitelné, že než se s funkcionálním programováním spřátelíte, napíšete sem tam něco delším způsobem, než by bylo možné. Najít rovnováhu mezi přílišnou stručností a extrémní explicitností není vůbec jednoduchý úkol, ale některých konkrétních chyb si můžete poměrně snadno všimnout:

- **Závorky navíc** – obzvláště snaha o volání funkcí jako v Pythonu (`(last(ys)) == (head(xs))`) vede k nečitelnému kódu. U funkcí s více parametry to navíc volání funkcí v Pythonu nebo C přestane připomínat: `max(a)(0)`. Přípustné varianty jsou `last ys == head xs`, `max a 0` nebo `a `max` 0`.

  – Neznamená to však, že se musíte zbavovat úplně všech syntakticky nadbytečných závorek. Ve výrazech s velkým množstvím infixových operátorů naopak psaní i některých závorek neměnících pořadí vyhodnocení přináší zlepšení čitelnosti – kdo by si pamatoval, zda má vyšší prioritu `&&`, nebo `==`?

  - Závorek může být příliš mnoho, i když žádné nejsou, syntakticky řečeno, navíc. Příliš velká úroveň zanoření se zkrátka špatně čte, takže pokud máte vnořeny třeba čtvery závorky, možná by bylo vhodnější nějakou část vytáhnout do lokální definice nebo pomocné funkce.

- **Redundantní vzory** – není potřeba navazovat seznam na `(x:xs)`, pokud ho vzápětí celý předhodíme funkci `map`. Navíc je pak potřeba vzor pro prázdný seznam, čehož se týká i další bod.
- **Redundantní ošetřování krajních případů** – typicky zbytečné „zdvojení“ dna rekurze (pro mocnění stačí případy `0` a `n`, je zbytečné ošetřovat `1` zvlášť) nebo vzor navíc pro prázdný seznam, s nímž by si `filter` poradil stejně dobře.
- **Nepoužití vzorů** – pokud ve funkci bereme `x` a následně všude píšeme `fst x` a `snd x`, nejspíš by bylo vhodnější použít rovnou vzor `(x, y)`.
  - Pokud ale zároveň potřebujeme pracovat s celou dvojicí zároveň, není to tak jasné. Můžeme si vypomoci zvláštní syntaxí pro tyto účely: `t@(x, y)` naváže na `x` a `y` prvky dvojice a na `t` celou dvojici. Jen pozor, že na místě `t` může být jen proměnná, ne komplikovanější vzor.
- **Lambda-funkce místo (jednoduché) částečné aplikace/složení** – používání lambda-funkcí je hezké, ale v opravdu jednoduchých případech to bez nich může vypadat lépe. Například `(\x -> x ^ 2 - 3)` je pro většinu lidí čitelnější než `(subtract 3) . (^2)`, ale použít `(\x y -> x + y)` místo `(+)` smysl nemá.
- **Opakování podvýrazů** – pro zkrácení a zpřehlednění slouží lokální definice (`where`, `let ... in`) nebo, pokud se výraz opakuje ve více funkcích, pomocné funkce.
- **Zbytečné komentáře** – krátké a jasné funkce, jako třeba `getPid (i, _, _) = i`, nepotřebují vůbec žádný komentář. Rozhodně není potřeba doslova popisovat funkci („Vrací první prvek trojice `ProcessInfo`“). U zadaných funkcí taky není žádoucí parafrázovat zadání. Naopak se může hodit popsat argumenty a výsledek (není-li to zřejmé třeba z jejich názvu) nebo hlavní myšlenku složitější funkce.

A pak jsou tu drobnosti, které nutně nevadí, ale vždycky potěší, když v kódu nejsou:

- **Pojmenování argumentu, ačkoli není použit** – obvykle se na místě nepoužívaných argumentů píše podtržítko.
- **Nepoužití η-redukce na *vhodných* místech**. Například pokud funkce jen třídí vstupní seznam funkcí `filter`. Naopak převádět všechno násilně do point-free tvaru, jen aby bylo možné η-redukovat, čitelnosti velmi škodí.

> K odhalení některých chyb se dají použít varování překladače/interpretu. Buď při spouštění můžete použít přepínač `-Wall`, nebo v interpretu příkaz `:set -Wall`. Místo `all` se dá použít konkrétní varování. Doporučujeme třeba:
>
> `unused-matches` (nepoužité nepodtržítkové formální argumenty)
>
> `unused-binds` (nepoužité funkce a lokální definice)
>
> `incomplete-patterns` (nedostatečné pokrytí vzory)
>
> `overlapping-patterns` (řádek definice se nikdy nepoužije; zapnutý implicitně)

Bílá místa a formátování

Čím byste naopak šetřit neměli, jsou mezery a zalomení řádků. Považuje se za samozřejmost psát mezery kolem operátorů, protože se to mnohem snáze čte. Pokud chcete, můžete „provzdušňovat“ i závorky a psát třeba `( length s )`.

Haskell vám dovoluje mezerami okrášlit kde co. Třeba v kostrách úloh si můžete povšimnout, že testovací data jsou úhledně zapsána na několik řádků. Vašemu oku může (a nemusí) lahodit třeba i zarovnávání vzorů pod sebe:

```
zip []     _      = []
zip _      []     = []
zip (x:xs) (y:ys) = (x, y) : zip xs ys
```

(Můžete si všimnout, že ve vzorech kolem dvojtečky mezery nepíšeme, ale v definici ano. Přísně vzato je to nekonzistentní, ale je to obecně přijímaná konvence.)

Dlouhé řádky rozhodně rozdělujte na více kratších; pokud možno tak, aby to dávalo smysl a například oddělovalo argumenty funkce. Jakýkoli řádek delší než 120 znaků si téměř jistě zaslouží zalomit, ale například podmíněné výrazy se leckomu lépe čtou zalomené bez ohledu na délku větví. Nezapomínejte taky na rozumné odsazování. Například

```
sgn x = if x < 0
        then -1
        else if x > 0
             then 1
             else 0
```

Potřebujete-li popsat argumenty funkce, využijte toho, že i typová signatura může zabírat víc řádků (primárně je však vhodné volit hezké názvy argumentů):

```
formatDouble :: Bool      -- always print sign
             -> Bool      -- scientific mode
             -> Int       -- max. decimal digits
             -> Double
             -> String
```

Zkrátka a dobře – nebojte se využívat mezer a řádkových zlomů k tomu, aby byl váš kód pěkný nejen myšlenkou, ale i čistě vizuálně.

> Odsazujte vždy jedině mezerami. Interpret vás na použití tabulátorů i sám upozorní. Je tomu tak proto, že místy je Haskell velmi vybíravý, co se velikosti odsazení týče. Například všechny lokální definice v jednom bloku `let` nebo `where` musí začínat ve stejném sloupci, a „sloupec“ je s tabulátory ošemetný pojem.

Neefektivní kód

Při psaní kódu je záhodno se zamýšlet i nad efektivitou jeho vyhodnocování. Při práci se seznamy nezapomeňte, že se nejedná o pole jako v jazycích C nebo Python, ale o jednosměrně zřetězený seznam, takže téměř všechny operace mají lineární složitost (vzhledem k délce vstupu) – vždy (nebo v nejhorším případě) se musí projít celý seznam. Výjimkami jsou v podstatě pouze operace přidávání a odebírání ze začátku seznamu, které mají konstantní složitost. Pokud například v rekurzivní funkci zjišťujeme v každém kroku délku vstupního seznamu, dostaneme funkci s kvadratickou složitostí.[1]

[1] Seznam délky n se projde n-krát, provede se tedy n^2 operací.

Jako příklad si představme funkci `piz`, která se chová jako `zip`, ovšem zarovnává seznam od konce: z delšího seznamu se zahodí prvky ze začátku, ne z konce. Tedy

```
piz [1,2] [3,4,5] ⇝* [(1,4), (2,5)]
```

Mohlo by nás napadnout něco takového (bázové případy vynechány):

```
piz xs ys = piz (init xs) (init ys) ++ [(last xs, last ys)]
```

Takové řešení je vcelku pěkně čitelné a je z něj jasné, co funkce dělá, nicméně má kvadratickou složitost[2]. To má hned několik příčin: `init` má lineární složitost a vlastně kopíruje celý seznam. `last` má lineární složitost. `(++)` má složitost lineární vzhledem k délce prvního seznamu a opět jej kopíruje. Dá se přitom dosáhnout lineární složitosti. Možným řešením je převést pomocí funkce `reverse` problém tak, abychom mohli použít funkci `zip`:

```
piz xs ys = reverse $ zip (reverse xs) (reverse ys)
```

Funkce `reverse` má také lineární složitost, ale stačí ji použít třikrát. To je u delších seznamů řádově lepší, nežli používat lineární funkci pro každý jeden prvek seznamu.

Ještě mnohem horší situace nastává, pokud **v rekurzi opakujeme rekurzívní aplikaci vícekrát**. Pokud voláme funkci rekurzivně dvakrát na stejný podproblém, pak i obě rekurzívní volání volají na své podproblémy tutéž funkci dvakrát a tak dále, z čehož vznikne složitost exponenciální.

Krásně je to vidět na následující funkci na deduplikaci prvků v seznamu:

```
unique :: Eq a => [a] -> [a]
unique [] = []
unique (x:xs) = if x `elem` unique xs then unique xs else x : unique xs
```

Vidíme, že podvýraz `unique xs` se opakuje. Překladač to ovšem nevidí a vyhodnocení rekurzívní aplikace proběhne dvakrát. Nápravu zjednáme lokální definicí:

```
unique (x:xs) = if x `elem` uxs then uxs else x : uxs
    where uxs = unique xs
```

To vyřeší i problém s opakovaným výpočtem – díky líné strategii se `uxs` (a tedy ani rekurzívní aplikace) zaručeně nebude vyhodnocovat vícekrát.

Podobně se zbytečné výpočty mohou opakovat i bez rekurze:

```
piz xs ys = zip (fst (sameSize xs ys)) (snd (sameSize xs ys))
```

Zde se opět stačí pomocí lokální definice zbavit opakovaného podvýrazu:

```
piz xs ys = zip xs' ys'
    where (xs', ys') = sameSize xs ys
```

To je nejen efektivnější, ale také se to lépe čte. Na rozdíl od příkladu s deduplikací se však složitost dramaticky nezměnila – obě implementace `piz` jsou lineární[3]. Přesto má smysl se opakovaným výpočtům vyhnout.

[2] Možná si říkáte, že seznamy se přece zmenšují; a skutečně to není přesně n^2, ale něco kolem $\frac{n^2}{2}$. To je však stále kvadratická funkce délky vstupních seznamů.

[3] za předpokladu lineárního `sameSize`