Složité výpočty s komentářem

oěžné výsledky numerických výpočtů mají jednu špatnou vlastnost — číslo na konci (typicky např. 42) vám většinou neřekne nic o tom, jakým způsobem se k němu došlo, jaké byly vlastně vstupy, jestli v průbehu výpočtu nedošlo k nějakým ohavným zaokrouhlovacím chybám, kolik iterací nějakého algoritmu bylo v N-tém kroku i-té části složitého výpočtu potřeba, nebo kdo a proč vlastně výpočet spustil a co s výsledkem zamýšlel dělat.

Poměrně očekavatelně, úkolem E a Haskellem tento problém nadobro vyřešíte.

Cíl

V Haskellu už umíme vyřešit např. kvadratickou rovnici:

```
solutions a b c = (\op -> (-b `op` sqrt d) / (2*a) ) `map` [(+), (-)]
  where d = b^2 - 4*a*c
```

Podstatně lepší (hlavně pro edukativní účely) by ale bylo tvářit se imperativně, a řešení si postupně okomentovat:

```
solutions a b c = do
  when (a==0) $ comment "Pozor, rovnice je ve skutecnosti linearni."
  let d = b^2 - 4*a*c
  comment $ "Diskriminant je " ++ show d
  if (d<0) $ do
    comment "Nemame reseni!"
    return []
  else do
    comment "Parada, mame alepon jedno reseni!"
    return $ (\op -> (-b `op` sqrt d) / (2*a) ) `map` [(+), (-)]
```

Komentované výpočty by navíc měly jít spojovat s jinými komentovanými podvýpočty, tj. následující program by měl nějak vrátit celé 2 komentáře o výpočtu kvadratických rovnic:

```
twoSolutions a b1 b2 c = do
  sol1 <- solutions a b1 c
  comment $ "Prvni rovnice ma " ++ show (length sol1) ++ " reseni"
  sol2 <- solutions a b2 c
  return $ sol1 ++ sol2
```

(Pozn. when z prvního příkladu je jako monádový “if bez else”, dokumentovaný tady.)

Úkol

Naprogramujte v Haskellu

- parametrizovaný datový typ Commented a, který ukládá výsledek výpočtu (typu a) doplněný seznamem textových komentářů,
- funkci comment :: String -> Commented () která vyrábí komentáře,
- funkci runCommented:: Commented a -> (a, [String]) která spustí komentovaný výpočet a dostane z něj výsledek a seznam komentářů
- podpůrné funkce a typové instance nutné pro to, aby komentované výpočty fungovaly s do-notací,

tak, aby příklady uvedené výše fungovaly.

Následně vyrobte funkci cFoldr :: Show a => (a->a->a) -> a -> [a] -> Commented a, což je “komentovaný foldr”, který do komentářů nějak rozumně zapíše průběh foldování se všemi mezihodnotami. Vyhněte se ručnímu vyrábění hodnot typu Commented — místo toho zkuste vyrobit přehledné řešení pomocí $comment$, $return$ a do-syntaxi.

(Pozn.: Běžný foldr zvládá s některými funkcemi zpracovat i nekonečné seznamy. U cFoldr tuto vlastnost nevyžadujeme.)

Jak na to

Commented a si představte jako kontejner na jediný výsledek typu a, který navíc obsahuje stringové komentáře. Například Commented Int nebo Commented (Tree String).

Jak jsme si ukázali na cvičení 11, do je jen syntaktický cukr. Výše uvedené příklady se ve skutečnosti odcukrují a funkce použité v do blocích budou spojeny pomocí operátorů (>>=) a (>>).

Konkrétně se bude spojovat váš datový typ Commented. Aby na něj zmíněné operátory fungovaly, musí mít instanci pro typovou třídu Monad (která obsahuje >>=, >> i return) a tranzitivně pro Applicative a Functor (protože monádová třída to z rozumných důvodů vyžaduje).

Pro řešení si tradičně můžete stáhnout šablonu. Do řešení jsem dopsal typy všech metod instancí, což by vás (spolu s mírným explorativním použitím ghci) mělo poměrně jednoznačně navést ke správné implementaci.

Na dvanáctém cvičení budeme podobnou funkcionalitu odvozovat pro velice obecnou strukturu stavových výpočtů, takže pokud náhodou nevíte jak na to, počkejte do dvanáctého cvičení.

SPOILER: Jak na to, trochu víc konstruktivně

Vezměme si následující jednoduchý program v do-notaci, jehož jediným účelem je vyrobit dva komentáře a vrátit s nimi výsledek jednoduchého výpočtu:

```
test = do
  comment "ahoj"
  comment "nazdar"
  return (2+3)
```

do-bloky nejsou nic speciálního — Haskell je na začátku kompilace přepíše na úplně normální výrazy spojené pomocí operátorů. V tomhle případě se funkce test transparentně přepíše na tohle:

```
test = comment "ahoj" >> (comment "nazdar" >> return (2+3))
```

Úplně polopaticky, vaším úkolem je jen definovat operátor >> a funkce comment a return tak, aby funkce test vrátila okomentovaný výsledek ve správné formě, tedy například:

```
Commented ["ahoj", "nazdar"] 5
```

Doporučený postup, jak toho dosáhnout, je představit si podvýraz

```
comment "nazdar" >> return (2+3)
```

jako:

```
Commented ["nazdar"] () >> Commented [] 5
```

(levý výraz obsahuje jen komentář a pravý jen výsledek)

Operátor >> by tyto 2 měl spojit poměrně triviálním způsobem do:

```
Commented ["nazdar"] 5
```

...což znamená něco jako “pětka s komentářem ‘nazdar’”. Pokud byste si nebyli jistí, slepovací operátor by (ve formě specializované pro Commented) měl mít následující typ:

```
(>>) :: Commented a -> Commented b -> Commented b
```

Jednoduchému řešení tradičně brání 3 drobné technické potíže:

- Operátor >> je přetížený, protože do-notace se používá i pro IO, stavové výpočty, různé ošetřování výjimek a selhávání (jako jsme si ukazovali s Maybe), výpočty generující seznamy možností, anebo výrobu parserů (více na posledním cvičení). Spolu s funkcí return a operátorem >>= je přetížený v typové třídě Monad. Tu můžete pro Commented přetížit úplně stejně, jako jsme přetěžovali libovolnou jinou třídu.
- Význam operátoru >>= z typové třídy Monad je trochu matoucí, níže je proto uvedený extra příklad.
- Typová třída Monad z různých praktických důvodů vyžaduje, aby typy, které se chovají jako Monad, byly zároveň v typové třídě Applicative; a typová třída Applicative podobně specifikuje, že tytéž typy musí být ještě v typové třídě Functor. Proto musíte typ Commented přetížit (v obráceném pořadí) pro všechny tři. (Mimochodem — Commented je funktor, protože jako datový typ si ho jde představit jako kontejner na nějaký výsledek, který navíc ještě obsahuje komentáře. Podobně to platí pro Applicative — pokud máte funkci s komentáři a nějaký parametr s komentáři, můžete jednoduše funkci pustit s parametrem a komentáře od obou prostě spojit. Příklad níže.)

Operátor >>= z typové třídy Monad existuje kvůli bindování mezivýsledků výpočtu do “normálních” nekomentovaných typů. Například, pokud použijeme definici funkce test seshora, následující kód:

```
test2 = do
  comment "Pred zavolanim test"
  a <- test
  comment ("test vratil " ++ show a ++ ", ja vracim dvojnasobek")
  return (2*a)
```

Způsobí, že a se uprostřed výpočtu objeví a bude k dispozici jako obyčejný nekomentovaný integer, ale “dodatečná informace” z komentářu ve funkci test se zachová a správně spojí s ostatními komentáři. Výsledkem tedy bude:
Commented ["Pred zavolanim test", "ahoj", "nazdar", "test vratil 5, aj vracim dvojnasobek"] 10

Operátor <- ve skutečnosti není operátor, ale speciální syntaxe v do-bloku, která se přepisuje přesně na operátor >>= takto:

```
test2 = comment "Pred zavolanim test" >> (
  test >>= \a -> (
    comment ("test vratil " ++ show a ++ ", ja vracim dvojnasobek")
    >>
    return (2*a)
  )
)
```

Výpočet s >>= si můžete představit následovně: Budeme zkoumat jednoduchý program:

```
prog = do
  x <- test
  comment ("test vratil " ++ show x)
```

Ten se přepíše na:

```
prog = test >>= (\x -> comment ("test vratil " ++ show x))
```

O funkci test už víme, na co se vyhodnotí:

```
prog = Commented ["ahoj", "nazdar"] 5 >>= (\x -> comment ("test vratil " ++ show x))
```

Operátor >>= vezme výsledek komentovaného výpočtu zleva (pětku) a předá ji funkci doprava, takže si můžete představit, že v programu bude chvíli existovat mezistav s dosazeným x, zjednodušený na operátor >>:

```
prog = Commented ["ahoj", "nazdar"] 5 >> comment ("test vratil " ++ show 5)
```

Mezistav se může dál vyhodnotit na tohle:

```
prog = Commented ["ahoj", "nazdar"] 5 >> Commented ["test vratil 5"] ()
```

a následně na:

```
prog = Commented ["ahoj", "nazdar", "test vratil 5"] ()
```

...což je finální výsledek programu.

Operátor >>= poslední 3 kroky (vytažení hodnoty zleva, dosazení do funkce doprava a spojení komentářů) dělá “najednou”. Pro jistotu přidávám typ >>= a return specializovaný pro Commented:

```
(>>=) :: Commented a -> (a -> Commented b) -> Commented b
return :: a -> Commented a
```

Poslední problematický bod v DÚ může nastat při přetěžování operátoru <*> z typové třídy Applicative. Zásadní intuice je taková, že <*> je jako aplikace funkce, jenomže funkce i parametr i výsledek jsou v nějakém kontejneru. Tedy, Commented se bude chovat podobně jako například Maybe, které přenáší “extra informaci” o selhávání následovně:

```
Just (\x -> x+1) <*> Just 5 == Just 6
```

Nothing <*> Just 76 == Nothing

Commented nebude přenášet extra informaci o selhání, ale jen extra komentáře:
Commented ["nejaka funkce"] (\x -> x+1) <*> Commented ["nejaky parametr"] 5 == Commented ["nejaka funkce", "nejaky parametr"] 6
Commented [] (\x -> x+1) <*> Commented ["komentar jen u parametru"] 5 == Commented ["komentar jen u parametru"] 6

Víc praxe na cvičení 12.