Recenzja projektu

"Haskell-fft"

Agnieszka Pulnar i Jakub Posłuszny

Projekt, który poddaliśmy recenzji jest autorstwa Adama Szczerby i Kamila Jarosz.

Link do repozytorium: https://github.com/adamszczerba/haskell-fft

Struktura projektu, jakość kodu i uwagi ogólne

Pozytywne aspekty, jakie zauważyliśmy:

- czytelny podział na moduły,
- struktura kodu jest przejrzysta, przegląda się go i czyta "przyjemnie"
- zespół nie zapomniał o opracowaniu komentarzy do wygenerowania dokumentacji

Aspekty negatywne:

- miejscami niewystarczająca liczba komentarzy, przez co ciężko jest zrozumieć, co dokładnie dzieje się w projekcie o tak skomplikowanym temacie, jeśli nie jest się zaznajomionym z *FFT*
- projekt wydaje się być niejednolity - nie rozumiemy, jaką rolę w *Szybkiej transformacji Fouriera* ma odgrywać moduł *Queue*
- autorzy opracowali testy tylko dla modułu *Queue*, a więc w projekcie brakuje jakichkolwiek testów do najbardziej skomplikowanych modułów - *Sound*, *GUI* czy *Config*
- nie jesteśmy pewni, czy można uznać to za wadę - autorzy projektu ustawili w `haskell-fft.cabal` build zależny od uruchamiania programu w systemie uniksowym, co zmusza użytkowników innych systemów, np. Windows, do samodzielnych poprawek w tym pliku w celu uruchomienia projektu

Uwagi dotyczące samego kodu

W funkcjach:

```
render :: IORef Spectra -> IO ()
render spectraRef = do
  clear [ColorBuffer]
  spectra <- get spectraRef
  renderSpectra spectra 0 (Conf.dataWidth/fromIntegral Conf.samplesPerFreq) 40
  flush

renderSpectra :: Spectra -> GLfloat -> GLfloat -> GLfloat -> IO ()
renderSpectra spectra from dx dz = renderPrimitive Quads $ do
  z <- newIORef (from :: GLfloat)
  mapM_ (\spec -> do
    zv <- get z
    renderSpectrum spec (zv) dx
    modifyIORef z (+(dz))) spectra

renderSpectrum :: Spectrum -> GLfloat -> GLfloat -> IO ()
renderSpectrum spec z dd = do
  x <- newIORef (0 :: GLfloat)
  mapM_ (\val -> do
    xv <- get x
    -- colorize
    color $ Color3 (xv/fromIntegral Conf.samplesPerFreq) 0 (1-xv/fromIntegral
Conf.samplesPerFreq)
    vertex $ Vertex3 (xv*dd) 0 z
    vertex $ Vertex3 (xv*dd+dd) 0 z
    vertex $ Vertex3 (xv*dd+dd) val z
    vertex $ Vertex3 (xv*dd) val z
    modifyIORef x (+1)) spec
```

Bardzo podoba nam się "rozbijanie zadań" w procesie renderowania na trzy kolejne, spójne ze sobą funkcje. Spójność występuje również w obrębie nazewnictwa. Uznajemy to za dobrą praktykę programistyczną i w naszej nauce chcemy dążyć do podobnych wzorców.

Naszym zdaniem, funkcja:

```
mfft :: [Complex Double] -> [Double]
mfft p = map (\l -> realPart $ abs l) (fft p)
```

Posiada dość niejednozaczną nazwę, chociaż przez to, że jest nieskomplikowana, nie jest trudno domyślić się, o co w niej chodzi. Dla zachowania spójności bardzo dobrego projektu proponowalibyśmy `takeRealPartOfResult` lub przynajmniej `modifyFFT` .

Wyjdziemy pewnie na totalnie "czepialskich" (ale ciężko znaleźć więcej błędów w naprawdę dobrym projekcie :-) - w linijkach takich jak:

```
  let freqDomain2 = map (\f -> realToFrac $ (8 * log(f+1))) freqDomain :: [GLfloat]
```

czy

```
vertex $ Vertex3 (xv*dd+dd) 0 z
```

Przyzwyczajeni do własnej praktyki programistycznej chętnie widzielibyśmy większe odstępy, np.:

```
(xv * dd + dd) czy (8 * log(f + 1)))
```

Sprawiają one, że kod staje się jeszcze bardziej czytelny.

Podsumowanie

Projekt uznajemy za naprawdę dobry i przy naszym poziomie znajomości języka Haskell trudno było znaleźć w nim więcej błędów. Analiza projektu o wiele bardziej skomplikowanego i ambitnego od naszego pozwoliła nam dostrzec bardzo interesujące zastosowania programowania funkcyjnego i dużo się nauczyć.