Przegląd kodu projektu 2048Haskell

Karolina Cynk, Kamil Faber

6 czerwca 2017

1 src/Grid2048.hs

isHavingTwoConsequtive
Funkcję *isHavingTwoConsequtive* zapisać można prościej bez użycia rekurencji jako:

```
1 isHavingTwoConsequtive :: RowState -> Bool
2 isHavingTwoConsequtive list = or $ zipWith (==) <*> tail $ list
```

Ponieważ plansza do gry nie jest duża to sposób implementacji powyższej funkcji nie ma wpływu na szybkość działania programu. W przypadku jednak ogólnym funkcja nierekurencyjna będzie dużo szybsza (np. dla listy 100000 elementowej przyspieszenie jest około trzykrotne, a w przypadku listy 1000000 elementowej jest ono już sześciokrotne).

removeBlankFields
Podobnie w przypadku funkcji *removeBlankFields* - dla niedużych list nie ma większej różnicy w szybkości działań w przypadku wersji rekurencyjnej i nierekurencyjnej (przedstawionej poniżej). Jednak dla list o większej liczbie elementów wersja rekurencyjna jest znacznie wolniejsza.

```
1 removeBlankFields :: RowState -> RowState
2 removeBlankFields list = filter (/= 0) list ++ (replicate (length $
      filter (0 ==) list) 0)
```

isGridHavingElement
Funkcję *isGridHavingElement* również można napisać w wersji bardziej czytelnej, co jednak nie ma wpływu na szybkość wykonania.

```
1 isGridHavingElem :: Int -> GridState -> Bool
2 isGridHavingElem _ [] = False
3 isGridHavingElem el grid = or $ (map (\x -> elem el x) grid)
```

makeMove
Funcja makeMove aktualnie przyjmuje string z informacją o kierunku ruchu. Takie rozwiązanie jest podatne na błędy typu literówki, itp. Wydaje się, że bardziej sensownym rozwiązaniem mogłoby być skorzystanie tutaj z enuma, a obsługę translacji wejścia na enuma przenieść w inne miejsce. Pozwoliłoby to wydzielić część związaną z obsługą błędów w inne miejsce.

```
1 makeMove :: GridState -> String -> GridState
2 makeMove gridState "Up" = makeMoveUp gridState
3 makeMove gridState "Down" = makeMoveDown gridState
4 makeMove gridState "Left" = makeMoveLeft gridState
5 makeMove gridState "Right" = makeMoveRight gridState
6 makeMove gridState _ = gridState
```

2 src/GameGUI.hs

Kod GameGUI jest nieczytelny i trudny do zrozumienia przez zbyt małą liczbę komentarzy (moduł zawiera tylko jeden komentarz odnoszący się do ogólnego działania programu).

Przyciski nie powinny być trzymane w mapie *buttonsMap* na stałe lecz wyciągane z komponentu nadrzędnego w razie potrzeby (funkcja *handleMove*, a także obsługa przycisku *New game*).

Dużą zaletą jest elastyczność implementacji gwarantująca możliwość uniezależnienia się od wielkości planszy. Pomimo, że z założenia plansza do gry jest wymiaru 4x4 to zmiana wartości parametru *gridSize* umożliwia grę na większej planszy bez wpływu na działanie programu.

Funkcje, które mają tylko jedno wywołanie w *do* powinny zostać uproszczone do zwykłego przypisania, na przykład:

```
1 makeButton :: String -> IO (Button)
2 makeButton label = do
3   buttonNewWithLabel label
```

może zostać równie dobrze zastąpione przez:

```
1 makeButton :: String -> IO (Button)
2 makeButton label = buttonNewWithLabel label
```

z kolei funkcja:

```
1 refreshButton :: String -> Int -> Int -> Map (Int, Int) (Button) -> IO
      ()
2 refreshButton label x y buttonsMap = do
3   let button = buttonsMap Map.! (x,y)
4   buttonSetLabel button label
```

może zostać równie dobrze zastąpione przez:

```
1 refreshButton :: String -> Int -> Int -> Map (Int, Int) (Button) -> IO
      ()
2 refreshButton label x y buttonsMap = let button = buttonsMap Map.! (x,y)
       in buttonSetLabel button label
```

Można również zmienić formę funkcji *indexesList* i zastosować w jej definicji funktorów aplikatywnych:

```
1 indexesList :: Int -> [(Int, Int)]
2 indexesList size = (,) <$> [0..size-1] <*> [0..size-1]
```

Jednak wersja nieużywająca funktorów aplikatywnych może wydawać się bardziej czytelna i zrozumiała.