2IPH0 Declarative Programming
Assignment

Bogdans Afonins 0969985

February 1, 2018

1 Introduction

This document is intended to give an overview of the code sample created for the Declarative Programming course. We explain the main principles of functional programming that form the basis of our application, mention tooling and some useful resources related to Haskell/functional programming, as well as we reflect on the results. The document contains three main parts that are of the main interest. The GUI section covers the main points on the GUI programming in Haskell and gives a brief introduction to the Gloss framework. Note, that we do not explain the boilerplate code on screen to program coordinate translation, because it is very specific and has many hardcoded values in order to make application look good enough for its users. Instead, we focus more on the ideas how to build GUI using the functional paradigm. The Program section describes certain parts of the code and explains design decisions made to implement the Minesweeper game. What is more, the Setup section has all necessary information on how to run the program.

One of the main parts of the course `2IPH0` is the group assignment where students are supposed to deliver a working project written in Haskell and apply theoretical knowledge gained during the course. We decided to create a well known game called Minesweeper, see [1] for the rules of the game. We will reference the rules further in the paper, but mainly to describe and analyze the way they are embedded into software, so we assume that the reader is familiar with the game.

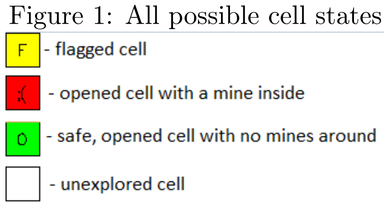
2 Setup

In order to be able to run the program, the following commands must be run :

```
> cabal update
> cabal install random-shuffle
> cabal install gloss
> cabal install either-unwrap
> ghc -o run Mines.hs
> ./run
```

Cells can be in 4 different states, see Figure 1.

Figure 1: All possible cell states.



Difficulty level of the game can be chosen in the beginning of the game by simply following instructions in a terminal. There are 3 levers to choose from :

- `Beginner` - 10x10 grid with 10 mines.
- `Intermediate` - 15x15 grid with 40 mines.
- `Expert` - 30x15 grid with 99 mines.

Controls :

- `Left-click` on an unexplored cell results in opening it.

- `Left-click` on a cell marked as the flag results in no reaction.
- `Right-click` on an unexplored cell results in marking this cell as the flag.
- `Button 'r'` when the game is over results in restarting the game with the same difficulty level.

3 Program

This section is indented to describe the main details and features of the program.

3.1 Entry point

The main setup of the game starts in the `main` function. We read the input and adjust the program to behave and initialize accordingly. We simply perform a number of IO operations within the do block to get users choices on the game difficulty and inform about the chosen mode. Besides that, we also set up the GUI of the program by calling the `play` function which is described further in the document, see `[3.3.1]`.

```
main :: IO ()
main = do
        putStr $ "Choose difficulty : \n"
            ++ "Type '1' for (Beginner) \n"
            ++ "Type '2' for Intermediate \n"
            ++ "Type '3' for Expert \n"
        hFlush stdout                               -- make sure the string is flushed
        m <- getLine                                -- read the users input
        generator <- getStdGen                      -- get a random number generator
        let mode = getMode m                        -- convert string to a Mode object
        putStr ((show mode) ++ " is chosen.")       -- inform the user about the mode
        hFlush stdout
        play (InWindow "2IPH0" (getWindow $ playgroundSize mode) (300, 300)) -- window settings
            white                                   -- background color
            50                                      -- update rate
            (getInitialModel generator mode)        -- initial state of the game
            view                                    -- graphics render
            actionHandler                           -- click/press listener
            stub
            where
                stub _ = id
                getWindow c = applyBoth (* (round cellSize)) c -- cellSize is hardcoded
```

3.2 Logic

In this section we go through all important points where the main design decisions were made while developing the program.

3.2.1 Types

First, we define the Minesweeper game in the context of software. We map the rules of the game into legal Haskell constructions and deal with them later on.

According to the rules, each cell can have a finite number of states, namely

- A cell can be opened by the player.
- A cell can be unexplored and contain a mine.
- A cell can be unexplored and marked by a flag.

Such setup can be represented as an algebraic type :

```
data CellStatus = Opened | Unexplored { flag :: Bool, mine :: Bool }
```

Despite the fact that this will work, the type is messy. We mix two states - visual, defining if the cell is opened or not, which influences the color of the cell, and its generated status during the program execution, defining if the cell contains a mine or not. We finish with the following setup :

```
data CellStatus = Mine | Flag | Opened Int
```

Now it is cleaner, visual and generated states are separated (we still need to keep the mines somewhere) and we do not include the unexplored state here.

As we know, the Minesweeper game is represented as a 2D grid, which makes the array data structure to be a good candidate for storing cells of the game. At least, that is what we would do in any imperative programming language, simply defining an array of arrays and index it according to the x and y coordinate of the cell we are interested in. Haskell, of course, supports mutable arrays (we do not really want to get a new field every time anything changes), but using `Data.Array.MArray` forces us to deal with mutability via `Data.Array.ST` and `Data.Array.IO` monads. Instead of adding redundant complexity into the program, we decided to stick to a simple dictionary defined in the `Data.Map` module. We use coordinates of the cell as the key, and the cell state as the value. This allows us to keep safe cells at one place and search for them in $O(logn)$ time.

```
type Cell = (Int, Int)
type Field = Map Cell CellStatus
```

As we mentioned previously, we separate the visual part and the state of the game. In order to store the state (mines) we use a set from the `Data.Set` module, where we store coordinates of the mines.

```
type Mines = Set Cell
```

3.2.2 World generation

Once we have all the main entities that form the core of the program, we continue with defining logic of the game. The first problem we have to solve is how to randomly generate some predefined number of mines (their coordinates) before every game. The standard `ghc` compiler lacks such functionality, but playing on the same grid over and over again is definitely not an option. So, we are either supposed to define some solution using the `IO` monad, because all functions in Haskell are pure and deterministic, or create our own pseudorandom generator, or we can look for something on the web. The first two solutions force us to invent something that definitely should exist already. After a quick search in Hoogle, we have found the `System.Random.Shuffle` module containing 3 functions that look almost similar.

```
shuffle :: [a] -> [Int] -> [a]
shuffle' :: RandomGen gen => [a] -> Int -> gen -> [a]
shuffleM :: MonadRandom m => [a] -> m [a]
```

According to the documentation the second function is exactly what we want. This function given a sequence, its length and a pseudo random number generator, produces the corresponding permutation of the input sequence. While the first two arguments are simple enough, the third one, `gen`, is free of a choice. We are using the standard random number generator - `StdGen`, because it completely satisfies all our requirements and there is no need to investigate something more complicated. So, we define the helper function `mineShuffle` that wraps the `shuffle'` function and is responsible for generating random permutations of coordinates within the context of the Minesweeper game. Its arguments are the generator or random numbers, the mode of the game (is mentioned later) from which the width and the height of the grid is taken and a list of cells to shuffle.

```
mineShuffle :: RandomGen g => g -> Mode -> [Cell] -> [Cell]
mineShuffle g m l = shuffle' l (h * w  - 1) g
```

How do we generate n random tupes? There many ways to that and it is difficult to say which one is better. We were aiming on making the code simpler, so in out program we simply generate all possible pairs within the range of the field with the use of a list comprehension and then take n of them.

```
generateMines :: RandomGen g => g -> Cell -> Mode -> Mines
generateMines rndGen initialCell mode = Set.fromList
    $ take mines
    $ mineShuffle rndGen mode
    $ [(a, b) | a <- [0 .. x - 1] , b <- [0 .. y - 1], not $ (a, b) == initialCell]
        where
        -- mode keeps configuration of the game
            x = fst $ playgroundSize mode
            y = snd $ playgroundSize mode
            mines = minesNr mode
```

Note, that according to the rules of the game, the first turn cannot result in opening a mine, so we additionally pass the cell that was clicked by the user on the first turn and make that the final set of tuples does not contain the clicked cell.

As mentioned earlier the Minesweeper game comes with different difficulty levels, namely `Beginner, Intermediate, Expert`. We model it a separate data type named `Mode` which encloses all necessary data. We store the number of mines to be placed on the grid, the size of the grid and the name of the level to display it to the user.

```
data Mode = Mode
 {
     name                 :: [Char],
     playgroundSize       :: (Int, Int),
     minesNr              :: Int
 } deriving Show
```

We also must keep the state of the game somewhere, at least we must keep track of all cells that are opened and where the mines are at every turn. For this purpose we create a new data type that encloses all necessary data we need. We do that by adding the `GameState` type, which stores the complete field, a set of mines and a flag determining if the game is over. Also, as we mentioned previously the mine placement is determined only after the first turn, so we box the `mines` field into the `Either` type. This allows to define different execution flow with the event handling and it is explained further in the document. So, if the mines value is of the `Left` type we generate the mine placement, if it is `Right` we just return the mine placement. The `mode` field stores all information about the current mode.

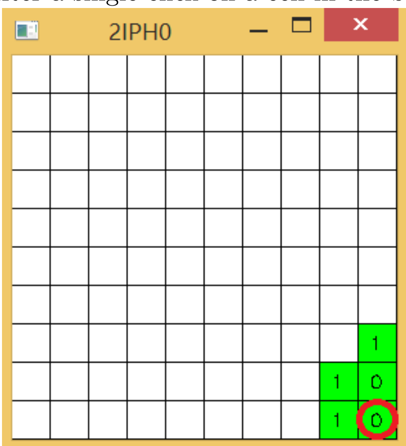
```
data GameState = GameState
    {
        field   :: Field,
        mines   :: Either StdGen Mines,
        mode    :: Mode,
        isOver  :: Bool
    }
```

3.2.3 Functionality

Since the source code of the program is long and contains some specific routines which are not of the main interest, we decided to devote this section to explain the main and non-trivial parts of the game.

According to the rules, once the user opens a safe cell, all neighbour cells must also open if there are no mines around. This should propagate till a safe cell is opened that has one or more mines around, see Figure 2.

Figure 2: Result after a single click on a cell in the bottom-right corner.



In order to implement this, we use a combination of standard build-in functions of `GHC` with our custom extensions, see the code sample below. First, we define the `event` function that given a cell and the current field returns completely new field. It is used to reproduce the field after the user interaction with the field. As described previously, we maintain two data structures, a dictionary to maintain the cells and a set to main the mines. The logic is as follows, we first check whether the processed cell is already in the map, if it is, then we simply return because the cell is already opened or it has the flag. Then we check whether the clicked cell is in the set of all mines, if it is, then, according to the rules, the game is over and we open all mines present on the grid. Further in the code, where all events are processed, we check if any cell is a mine in the map of all cells, and if it is the case then the corresponding flag is set within the `GameState` object (`isOver`), which disables any processing of events except for restarting the game by pressing the `'r'` key. If the first two conditions fail, then we know that this cell is unopened and we must explore it and all its neighbours. To perform this, we first explore all neighbours of the cell by defining a list of possible steps we can take (`moves`), mapping it with an anonymous function that simply adds the x and y coordinates of the current cell to each possible move resulting in a list of possible (`neighbours`) and finally filtering out the cells that are out of the bounds. In order to explore the neighbours of neighbours we define additional function, namely `openAllSafe`, that uses the `foldr` function and reduces all cells to a single `field` with all necessary fields being opened. Note, that this can be easily deducted from the type of the `foldr` function whose type is `(Cell -> Field -> Field) -> Field -> [Cell] -> Field`. How do we distinguish between the cases when we should explore the neighbours and when we should not? We simply count the number of mines surrounding the current cell (`countNeighbours`), if the the value is equal to 0 we start exploring, otherwise we just open the cell and return.

```
event :: Cell -> Field -> Field
event (c1, c2) f
    | Map.member (c1, c2) f         = f               -- do not process a cell more than once
    | Set.member (c1, c2) getMines  = openAllMines     -- lost, open all mines
    | otherwise = if isMineNeighbour
                  then openCellSafe           -- just open a cell
                  else openAllSafe            -- open all safe neighbours
    where
        -- explore neighbours
        neighbours :: [Cell]
        neighbours = Prelude.filter checkBounds
            $ Prelude.map (\(a, b) -> (a + c1, b + c2)) moves
                where
                    checkBounds = \(a, b) ->
                        (0 <= a && a < x) &&
                        (0 <= b && b < y) -- ensure we dont leave the grid
                    moves = [(1, 0), (0, 1), (-1, 0), (0, -1), (1, 1), (-1, -1), (1, -1), (-1, 1)]

        countNeighbours :: Int
        countNeighbours = length $ Prelude.filter (`Set.member` getMines) neighbours

        isMineNeighbour :: Bool
        isMineNeighbour = not $ (0 ==) countNeighbours

        openCellSafe :: Field
        openCellSafe = addCell (c1, c2) (Checked countNeighbours) f

        openAllSafe :: Field
        openAllSafe = Prelude.foldr event openCellSafe neighbours

        openAllMines :: Field
        openAllMines = Prelude.foldr event openCellMine $ Set.elems getMines
            where openCellMine = addCell (c1, c2) Mine f

        x = fst $ playgroundSize mode
        y = snd $ playgroundSize mode
```

3.3 Graphical User Interface (GUI)

3.3.1 Window

The most convenient way to present the Minesweeper game is via GUI. For this purpose we have chosen the Gloss graphical framework, see [2], which layers over OpenGL, hides the complexity of dealing with graphics and provides a simple set of functions to manage all kinds of graphical components.

Instead of regular imperative approach of dealing with graphics, when a programmer simply specifies what to draw first, what draw after this and so on, Gloss follows completely different path. Gloss uses a standard set of proprietary functional techniques, combinators and function composition. The main primitive in Gloss is `Picture`, that represents simple graphical units and provide functions manage them, see the code below. For example gloss allows to scale, rotate, color, rotate elements.

```
import Graphics.Gloss
--                            title          size      position color component
main = display (InWindow "Hello World" (400, 150) (10, 10)) white picture
```

```
picture        = Translate (-170) (-20) -- move text to the center
             $ Scale 0.5 0.5         -- display it half the original size
             $ Text "Hello World" -- text to display
```

Hello world example using Gloss

The `display` function is responsible for opening a new window and displaying the given picture with the specified attributes.

Since the program requires us to define some custom event handling, namely mouse and keyboard presses, we use the `play` function, which roughly does exactly the same as `display`, but allows us to define a custom way to react on user events, see [3] . Lets take a closer look at it.

```
play :: Display  -- Graphics.Gloss.play
        -> Color
        -> Int
        -> world
        -> (world -> Picture)
        -> (Event -> world -> world)
        -> (Float -> world -> world)
        -> IO ()
```

The first two arguments are trivial, they define a type of the applications window (full screen/window mode) and the background color. It is worth noting, that `Gloss` maintains some sort of "scene" or "internal state" of the application and it is stored in the `world` **type variable**, `world` is not a type and because of that the `play` function is polymorphic in some sense and can work with different states. In order to operate on `world` there are three functions.

```
(world -> Picture)
```

This function simply generates a `Picture` object out of the value of `world`. In the code sample we provide it is called `view`.

```
-- defines logic to render GUI
view :: GameState -> Picture
```

In the context of our program we define the way different kinds of cells must be drawn. For example, to determine the color of any cell we generate all coordinates within the game grid within the `generateRangeToScreen` function again using a list comprehension, then we look for each coordinate in the dictionary and finally match against a type of each cell. Note, the `pictures` function combines many objects of the type `Picture` into a single object and is a part of the `Gloss` framework. We apply exactly the same procedure also do draw the grid and draw labels for each cell.

```
cells :: Picture
cells = pictures [ combine c (colorCell $ Map.lookup c fld) drawCell
| c <- generateRangeToScreen ]
    where
        colorCell :: Maybe CellState ->  Color
        colorCell Nothing                = white
        colorCell (Just Mine)            = red
        colorCell (Just (Checked val))   = green
        colorCell _                      = yellow

-- combines properties of a picture into one complete image
combine :: (Int, Int) -> Color -> Picture -> Picture
combine cell clr fig = translate x y $ color clr $ fig
    where
        x = fst $ mapCell cell
        y = snd $ mapCell cell

-- generates a list of grid coordinates
generateRangeToScreen :: [(Int, Int)]
generateRangeToScreen = [(x, y) |
    x <- [0 .. (fst gridSize) - 1],
    y <- [0 .. (snd gridSize) - 1]]
```

This function is called each time when any event happened and is reponsible to react to change the value of `world`. We define it as `actionHandler` in the code sample. More on that is covered in 3.3.2.

```
(Event -> world -> world)
```

This function with combination with the third argument makes `world` to change according to some defined timings. This function is interesting for us, since they game we develop is static and in the code sample we use the identity function (`id`).

```
(Float -> world -> world)
```

Note, that all are pure functions, except the `play` function, of course.

3.3.2 Event handling

The program is mainly based on user inputs, se we must be able to process them and react properly. Gloss provides a convenient way to listen to different events - mouse clicks/buttons/swipes/etc. `Event` is an algebraic data type that provides a wide variety of constructors, see [4].

```
data Event = EventKey Key KeyState Modifiers (Float, Float) | EventMotion (Float, Float)
```

We are mainly interested in a certain key events, like right/left mouse clicks and a single key press, so we use only the `EventKey` constructor, which is defined as follows

```
data Key = Char Char | SpecialKey SpecialKey | MouseButton MouseButton
```

These two data types build the core of the event processing in Gloss, using those allows to completely cover all requirements for the Minesweeper game. In order to implement the program specific logic, we define a new function `actionHandler` of a type `Event -> GameState -> GameState`. Simply, it receives an event, the current game state and returns a new state of the game. Note that this type perfectly aligns with the required type by the framework. Previously, in section 3.3.1 we discussed the `play` function that receives a function of a type `(Event -> world -> world)`. Since the framework hides the complexity of managing such events, we only need to define the behaviour.

First, we discuss the smallest code sample, namely the one responsible for restarting the game with the same level of difficulty when the game is over, see the sample below. This can be done by simply pressing the `'r'` key on the users keyboard. As discussed previously, we shuffle the cells before the start every game by using `RandomGen` created in the beginning with the `getStdGen` function. The issue is that we cannot reuse it here, because if we do and the user starts the game with the same move right after restarting the game, the permutation would be exactly the same. To prevent this from happening, we create a new random generator by calling the `newStdGen` function, see [5], which updates the last random generator and returns it. Note, that in the code sample we use the "back door" into the `IO` monad, namely calling the `unsafePerformIO` function to perform the monadic computation (getting the new generator) immediately. The reason for this is simply that the feature was added in the end of the development process and the design at that time was not easily modifiable for passing the generator being wrapped in `IO`.

```
-- restart game
actionHandler (EventKey (Char 'r') Down _ _) GameState { isOver = True, mode = mode }
    = let gen = unsafePerformIO newStdGen in getInitialModel gen mode
```

In order to provide the users with functionality to mark cells that can possibly contain a mine with a flag and avoid occasionally opening those cells, we define a new listener. We define the right mouse button to flag cells. The logic is as follows, if the cell we process is not in the map, then it is unexplored and we can mark it with the flag, otherwise it is already marked and we remove it from the map. We also must cover the case if the user occasionally presses on any other cell that is covered here, hence we add additional pattern match and simply return the current state. Note, that this code sample is executed only if the game is not over, because we do not allow flagging cells once the game is finished.

```
-- right mouse click action
actionHandler (EventKey (MouseButton RightButton) Down m mouse) GameState
    {
        field = fld,
        mines = (Right mines),
        isOver = False,
        mode = mode
    } = case Map.lookup (mapScreen mouse gridSize) fld of
            Nothing -> GameState -- add flag
                {
```

```haskell
                field = Map.insert (mapScreen mouse gridSize) Flag fld,
                mines = (Right mines),
                isOver = False,
                mode = mode
            }
        (Just Flag) -> GameState -- remove flag
            {
                field = Map.delete (mapScreen mouse gridSize) fld,
                mines = (Right mines),
                isOver = False,
                mode = mode
            }
        (Just _) -> GameState -- ignore anything else
            {
                field = fld,
                mines = (Right mines),
                isOver = False,
                mode = mode
            }
        where
            gridSize = playgroundSize mode
```

The main complexity of handling the right mouse button contains in the `event` function, which was explained earlier. But there are still things to be clarified. As was mentioned several times, the `GameState` data type contains a set of mines which are wrapped in `Either`. This is because of the rules of the Minesweeper game - the first turn cannot result in opening a mine. Hence we need to distinguish between the first opened cell and all other, and the `Either` type is a good choice. We distinguish between two cases. First, when the type of the `mines` field is `Left StdGen`, which means that it is the first right-click in this particular game and we must generate mines. We accomplish that by using the helper function `getMines` that delegates the call to `generateMines`, passing the generator, the cell that is clicked and the mode. With such approach we are able to generate the field right after users first interaction. Second, when the type of the `mines` field is `Right Mines` we simply process the event (open the cell or change the value of `isOver` if needed) or simply return the current game state to disable event processing if the game is over.

```haskell
actionHandler :: Event -> GameState -> GameState
-- left mouse click action
actionHandler (EventKey (MouseButton LeftButton) Down _ mouse) GameState
    {
        field = fld,
        mode = mode,
        mines = ms,
        isOver = over
    } =
    case ms of
        (Left m) -> GameState
            {
                mines = (Right getMines),
                field = event (mapScreen mouse gridSize) fld,
                isOver = False,
                mode = mode
            }
        (Right m) -> if not over
            then GameState
                {
                    field = event (mapScreen mouse gridSize) fld,
                    isOver = isMine (mapScreen mouse gridSize) renewedField,
                    mines = (Right m),
                    mode = mode
                }
            else
                GameState
                {
                    field = fld,
                    mines = ms,
```

```haskell
            isOver = over,
            mode = mode
        }
where
    gridSize = playgroundSize mode
    renewedField = event (mapScreen mouse gridSize) fld
    getMines :: Mines
    getMines = case ms of
        (Right v) -> v
        (Left v) -> generateMines v (mapScreen mouse gridSize) mode
    ------------------------------------------------
    -- the event function here, see section 3.2.3 --
    ------------------------------------------------
```

References

[1] *http://www.wikihow.com/Play-Minesweeper*, WikiHow: How to play Minesweeper

[2] *http://gloss.ouroborus.net/*, Gloss wiki: Introduction

[3] *https://hackage.haskell.org/package/gloss-1.11.1.1/docs/Graphics-Gloss.html*, Hackage: Graphics.Gloss

[4] *https://hackage.haskell.org/package/gloss-1.1.1.0/docs/Graphics-Gloss-Game.html*, Hackage: Graphics.Gloss.Game

[5] *https://hackage.haskell.org/package/random-1.1/docs/System-Random.html*, Hackage: System.Random