Allan Guigal

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Minesweeper - Haskell

I) Minesweeper rules implementation

The minesweeper implemented has the following features:

- The board is randomly generated for each game
- The first square you open is never a mine
- It is possible to configure the game within the code with the settings structure. You can change the width, height, and bombs count.
- You can flag (or mark) a cell as a potential mine. For this, there is a “Mode” button, when you press it, it toggles the mode, going from Dig to Flag and Flag to Dig and so forth. In flag mode, left clicking a hidden cell will flag it, or unflag it if it was already flagged. In dig mode, it will “open” the cell as normally expected.
- If you open a square with no neighboring mines, all its neighbors will automatically open. This can cause a large area to open quickly. It is the automatic propagation of the discovering of the cells without bombs in its neighboring cells.
- Visual and colorful indication of the bombs count in the neighboring cells when the cell is opened.
- Show the bomb locations when the game is over (either won or lost) so that the player can understand how the game went.
- Ability to restart the game with a button (it will generate a new random board)
- To win the game, you need to open all the safe squares
- You lose the game if you open a mine
- Game status, indicate if you have won, lost, survived your move, or made an invalid play (for example already played this move)
- Possibility to let the AI play, it will find and play a safe move. Either flagging a cell or playing a safe square. If no safe move can be determined, it will just pick the first cell it can play. Just press the button “IA Play”.

This mean all the features have been satisfied except that the IA could go further in evaluating unsafe moves with probabilities.

II) Program design and Haskell usage

The game logic

Considering the data structure, what were the motivations in choosing one? As an experienced imperative and object-oriented programmer, the most intuitive approach was to make a 2-dimensional array holding the equivalent of an object, therefore a data type with records in Haskell.
This would have looked like this:

```haskell
type Board = [[Square]]
```

However, after seeing what the accessors of an array looked like in Haskell “!!”, it didn’t seem right to have them floating all over the code. That’s when I came across multiple sources on the web that arrays aren’t idiomatic in Haskell. Moreover, arrays are error prone with their indexes. It is quite easy to access an index outside the boundaries of the array and crash the program that easily. Handling these cases in Haskell didn’t look easy and clean. Another solution that was more functional must exist. Instead, declaring 3 sets to hold the essential information from which we could build all the relevant functions proved to be easier and more elegant:

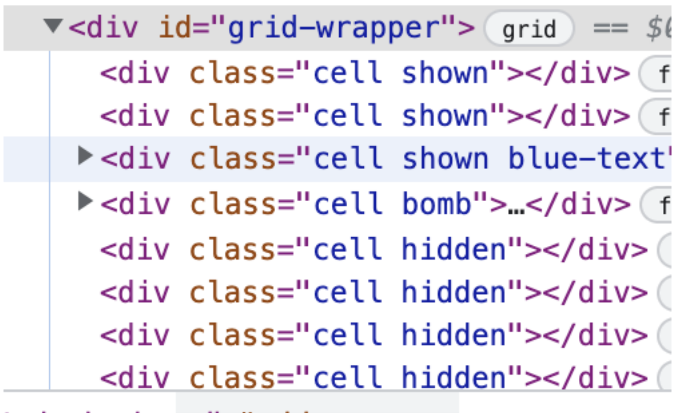
```haskell
data Board = Board
 {
   discoveredCells :: Set Location
 , flaggedCells :: Set Location
 , bombCells :: Set Location
 } deriving (Show)
```

Lists and sets over arrays allow to use induction which is a strength to use in functional programming like Haskell.

The UI

The UI was built to leverage the web browser the framework is based onto and the reactive programming paradigm with its specific module “Reactive.Threepenny”. Reaching a functional reactive programming (FRP) design was the aim and is the goal achieved.

To begin with, the styling is done thanks to a CSS stylesheet with simple instructions and classes. The grid is represented by a div wrapping other divs, individually being a cell. The DOM looks like this:



```html
▼<div id="grid-wrapper"> grid == $
    <div class="cell shown"></div> f
    <div class="cell shown"></div> f
   ▶<div class="cell shown blue-text'
   ▶<div class="cell bomb">…</div> f
    <div class="cell hidden"></div>
    <div class="cell hidden"></div>
    <div class="cell hidden"></div>
    <div class="cell hidden"></div>
```

We notice the different classes “shown, bomb, blue-text” that give the proper styling to each cell.

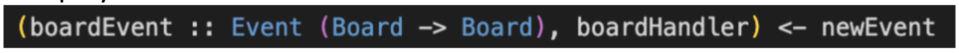
As we have seen throughout the lecture and the incrementing examples, we tried to tend to a reactive programming UI.

Using IORefs would have been much straight forward and the classical way of thinking with a mutable state through direct accessors. However, having to build “Behaviors” and manipulate “Events” gives us a better design and better performance by only performing UI updates and actions when the behaviors (data) are updated. This is quite a gain of efficiency. Also, the design is more robust and less prone to errors that could arise with side effects such a modifying a global state akin to IORefs.

We have 2 custom event streams (and their handlers to fire events):

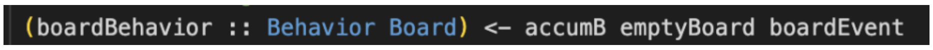
```
(boardEvent :: Event Board, boardHandler) <- newEvent
(modeEvent :: Event (PlayMode -> PlayMode), modeHandler) <- newEvent
```

Also, there are two behaviors built on top of the streams, one for each stream. The “board behavior”, initially it was an « accumB » accumulating the behavior based on its event stream. However, to add more state to the game such as the « PlayResult » to determine if the player won, lost or survived and should continue playing – the result being returned by the play move, the accumulation “Board -> Board” didn’t stand anymore because it used this play move.



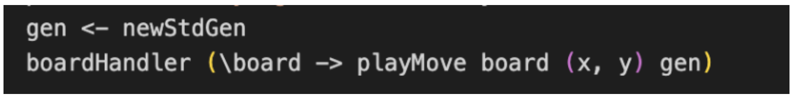
```
(boardEvent :: Event (Board -> Board), boardHandler) <- newEvent
```

Figure 1 The old board event stream



```
(boardBehavior :: Behavior Board) <- accumB emptyBoard boardEvent
```

Figure 2 The old board behavior creation



```
gen <- newStdGen
boardHandler (\board -> playMove board (x, y) gen)
```

Figure 3 The old board handler usage

Unless the winning and losing conditions would be checked elsewhere. This could have been another option but integrating the play result as an output to the playMove function seemed more intuitive. The event stream and behavior had to be changed to simply be an “Event Board”.
Therefore, I converted the behavior to a stepper. This time, to prepare for the next board to send in the event stream, we need to read the behavior’s current value.

Now it is as such:

```
(boardBehavior :: Behavior Board) <- stepper emptyBoard boardEvent -- the initial board is an empty board
```

```
   board <- currentValue boardBehavior
   let (newBoard, newPlayResult) = playMove board (x, y) gen
   boardHandler newBoard
```

This allows to extract and use the play result for other logical and visual intents.

The second event stream and behavior are the ones for the play mode. (To change the play mode and act accordingly). The play mode is either to dig or flag a cell.

```
  (modeEvent :: Event (PlayMode -> PlayMode), modeHandler) <- newEvent
```

```
(modeBehavior :: Behavior PlayMode) <- accumB Dig modeEvent -- the initial play mode is "dig"
```

```
liftIO $ modeHandler toggleMode
```

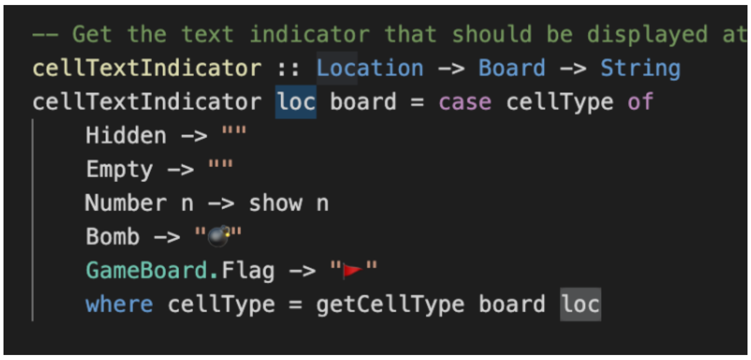
The handlers are invoked and thus the events for the Board and PlayMode are fired within events themselves. For the board, there is one event handler per cell that will play the corresponding move and fire an event with the new board. For the play mode, it is when the mode button is clicked, and it toggles the mode.

And now, the most elegant and powerful part for having built this entire design with streams and behaviors. The three sink operations that can be used thanks to the behaviors:

```
element mode # sink UI.text (modeToIndication <$> modeBehavior)
element cell # sink UI.class_ ((cellClass (x, y)) <$> boardBehavior)
element cell # sink UI.text ((cellTextIndicator (x, y)) <$> boardBehavior)
```

This allows a clean way to maintain the UI elements (visuals) up to date with the behaviors. We now have the behaviors encapsulating the state of the play mode and board at any time and varying through the time. And the button will display the current mode when updated. And more importantly, each cell will display the right text (nothing or the bomb count or the flag char or the bomb char) and have the correct CSS class at any time.

However, the behavior gives a Board and not the exact information we need. For this purpose, I developed "mappers" functions, that map model types or data that we have thanks to the behaviors to actual UI information to be used in the rendering and display. For example:



```
-- Get the text indicator that should be displayed at
cellTextIndicator :: Location -> Board -> String
cellTextIndicator loc board = case cellType of
    Hidden -> ""
    Empty -> ""
    Number n -> show n
    Bomb -> "💣"
    GameBoard.Flag -> "🚩"
    where cellType = getCellType board loc
```

Figure 4 example of a mapper from a cell location and the board (in the behavior) to a string for the number or character to be displayed in a cell at any time

I use the <$> operator to create a new behavior that converts the original behavior to a String holding an actual display information. The sink operation can therefore use this new specifically tailored behavior.

III) Conclusion

We have covered the main program design. We have seen the game and model structure, the functional reactive programming in the UI. The event streams and behaviors built and how they convey information to the UI elements through the mappers and "sink" operations.

To add some thought about Haskell and the threepenny gui library. Having different layers of monads is counter intuitive sometimes, and it was quite painful to get to use them correctly. Building UIs in Haskell is quite cumbersome. But once the right monad transformers are found it was fine, the only downside remaining was the syntactic surplus from going to a monad to another when it was needed. That’s why I limited it as much as possible.

Moreover, the laziness and purity combined and all the optimization that comes along with it really alleviates our mind about “performance, performance, performance”. We focus on elegantly shaping our code and declaring the most senseful functions. Of course, performance is to be kept in mind when coding the algorithms and picking the data structures, but it’s less prevalent.

Finally, and more importantly, once the logic was set, the gameboard being completely pure. There wasn’t much if any debugging needed. The rare cases were solved in a fraction of second using “Debug.Trace”. After the execution only missing features were spotted but never a real bug or unexpected behavior. The purity of not having any side effects really helps to program bug free programs. Moreover, the reactive programming really separated how to update the state by firing the proper events and then how to render the behaviors that arise from the streams. Having this flow really made the conception and programming clear, which means less errors and bugs.

IV) Game screenshots

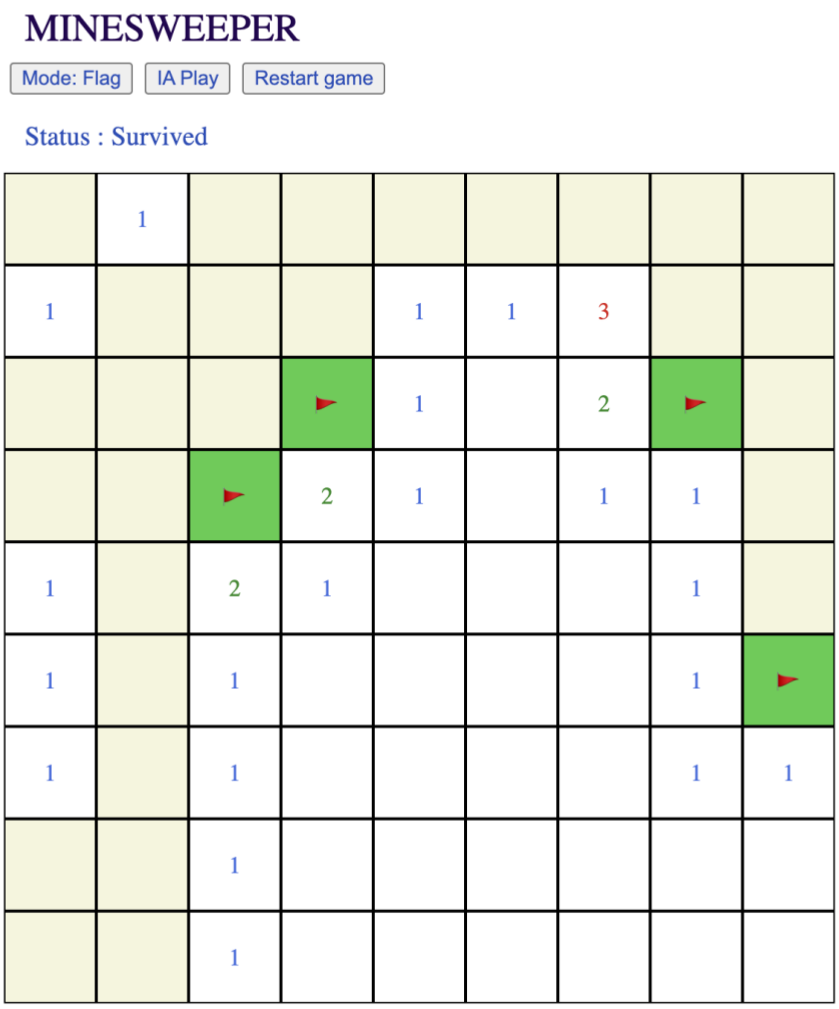


Figure 5 Game in progress, different level of indications, some flags set and the status as "Survived"

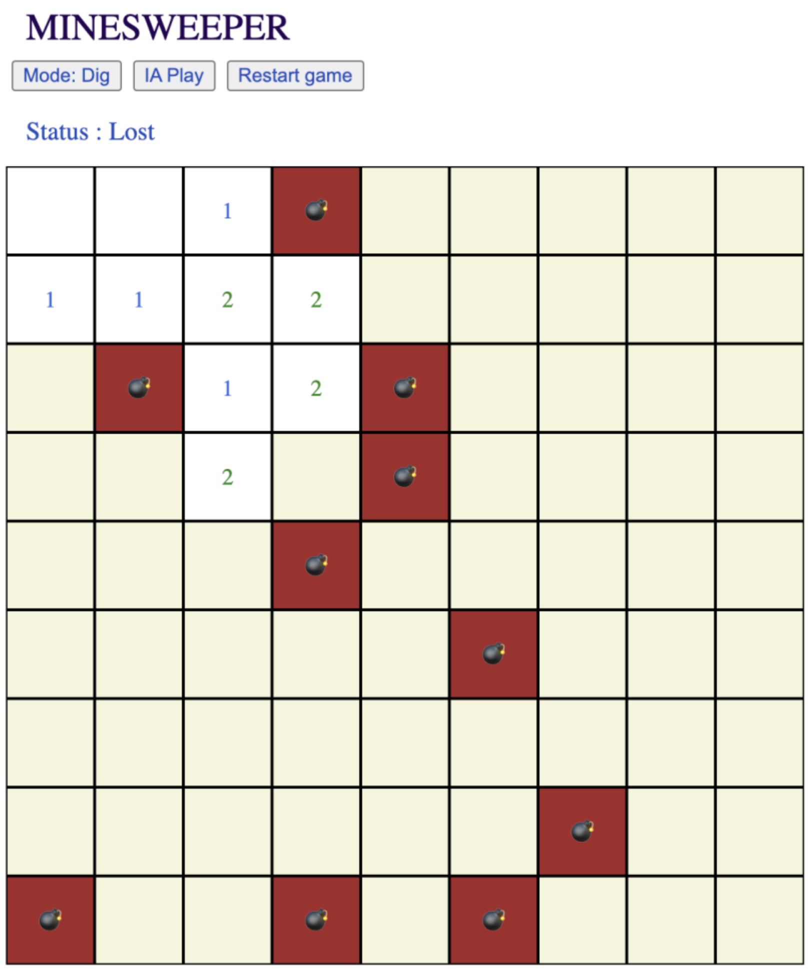
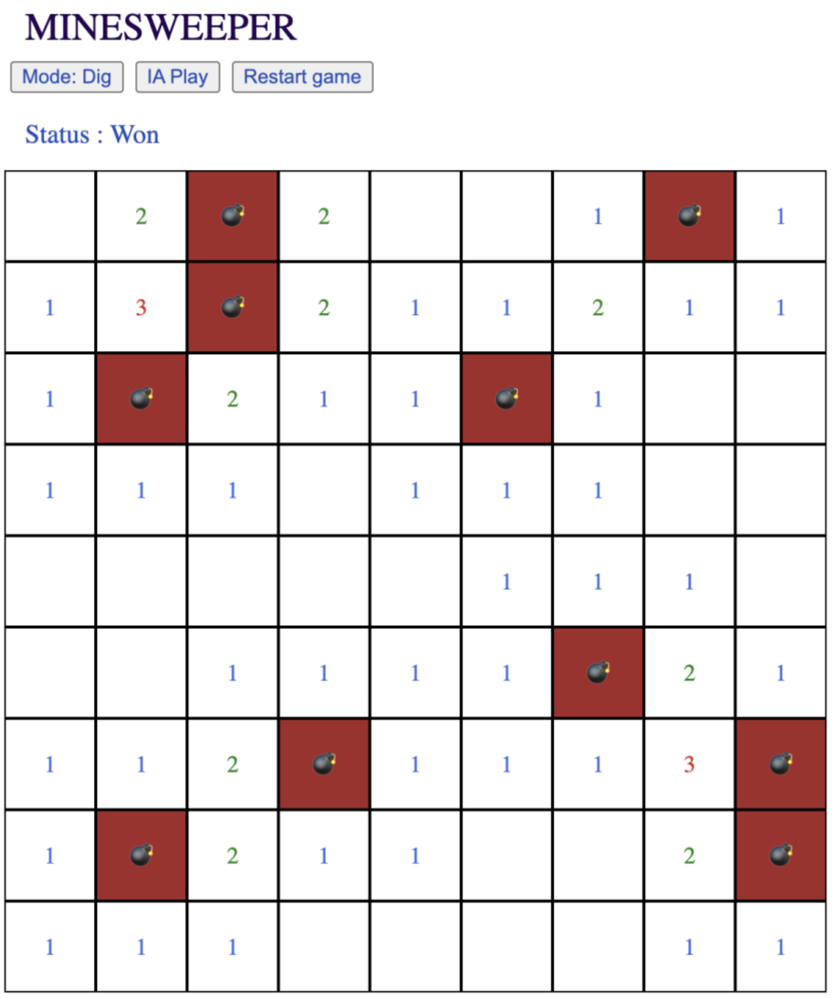


Figure 6 Game lost (user clicked on a bomb), the bombs are revealed



Status : Won

	2	💣	2			1	💣	1
1	3	💣	2	1	1	2	1	1
1	💣	2	1	1	💣	1		
1	1	1		1	1	1		
					1	1	1	
		1	1	1	1	💣	2	1
1	1	2	💣	1	1	1	3	💣
1	💣	2	1	1			2	💣
1	1	1					1	1

Figure 7 Game won (status won), the bombs are revealed. All the safe squares were opened.