Project Group
Applied Functional Programming

Web Development with Haskell

Meike Grewing, Lukas Heidemann, Fabian Thorand and Fabian Zaiser

Informatik, Universität Bonn, Germany

Abstract

The goal of the project group "Applied Functional Programming" was to get an insight into web development using Haskell and the web framework *Yesod*. In this paper, we want to document our main project, a web implementation of the well-known game *Battleships* including an artificial intelligence. You can find the code on GitHub[1] and play the current production version[2].

1 Introduction

Our main project was to develop a *Battleships* web application providing a neat graphical user interface and an adjustable AI to play against. We decided to use the following rules:

- Each player begins with a total of ten ships: one ship of length five, two of length four, three of length three and four of length two.
- When a shot hit a ship, the player may fire again.

In addition to the usual rules, our version of the game includes some modifications:

- In addition to firing a shot, the player is allowed to move one of his ships forward or backward by one cell in each round (adhering to the placement regulations).
- As soon as one of your ships is completely sunk, other ships can be moved across the space it previously occupied.
- The number of turns is limited to prevent endless games. After the last turn, the player with the most remaining ships wins.

2 Architecture and Framework

For our project, we used the *Yesod* Web Framework[3] developed by Michael Snoyman and others. Its goals are performance, type-safety and conciseness of code.

For these purposes Yesod offers embedded DSLs[4] using template haskell for writing HTML, CSS and JavaScript code which allow using variables from Haskell code and are syntactically simpler (e.g. the DSL for HTML uses indentation instead of closing tags). There is also a lot of

[1] `https://github.com/zrho/afp/tree/master/battleships`
[2] `http://www-pg.iai.uni-bonn.de/battleships`
[3] `http://www.yesodweb.com/`
[4] Domain Specific Languages

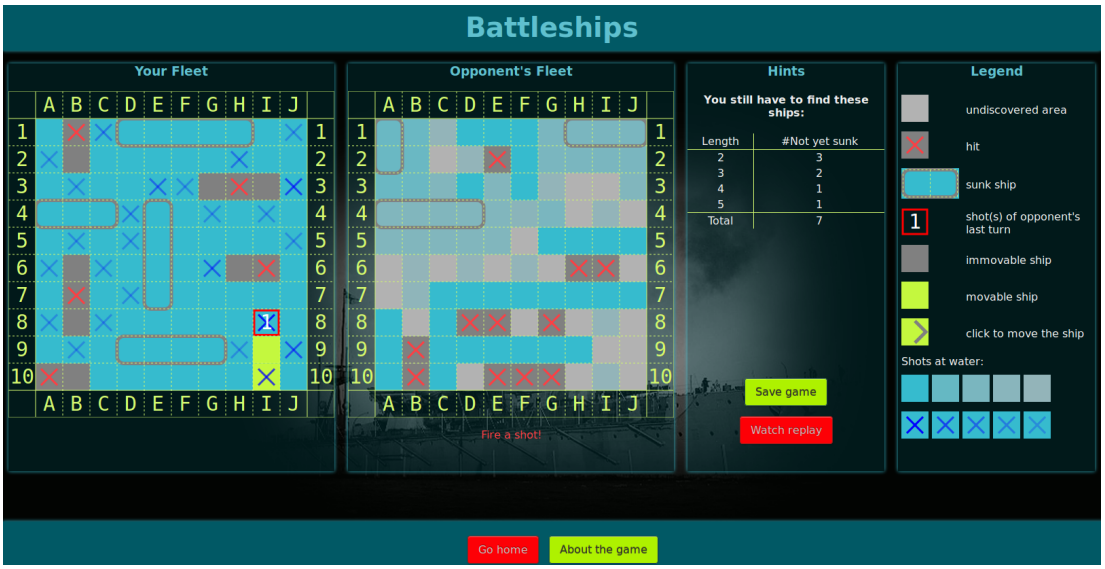

Figure 1: Typical game view. Shots are fired by clicking on the target.

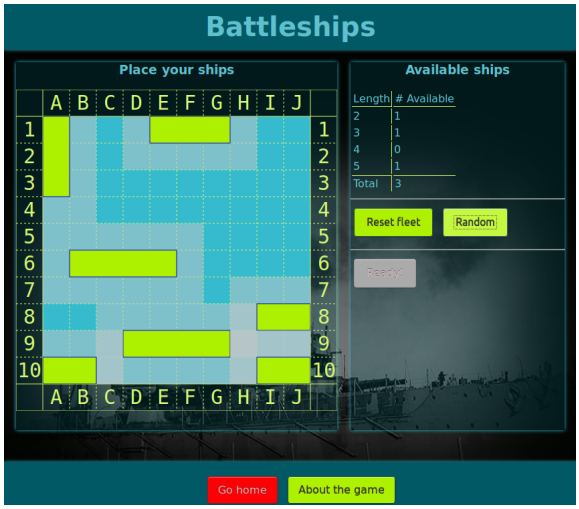

Figure 2: The ships can be placed randomly or via drag and drop.

compile-time checking involved. URLs in Yesod are type safe, meaning that one specifies which parameters a certain route expects and their types. Whenever it is referenced, e.g. in hyperlinks or redirections, the compiler checks if the expected and actual data formats match. This makes 404 errors for internal links almost impossible. Furthermore, the type system guarantees that dynamic content inserted into HTML pages is verified or escaped, preventing common attacks like cross site scripting (XSS). Yesod also offers good support for internationalization. In fact, our project web site is available in English and German.

The game logic resides on the server and is completely written in Haskell. On the client

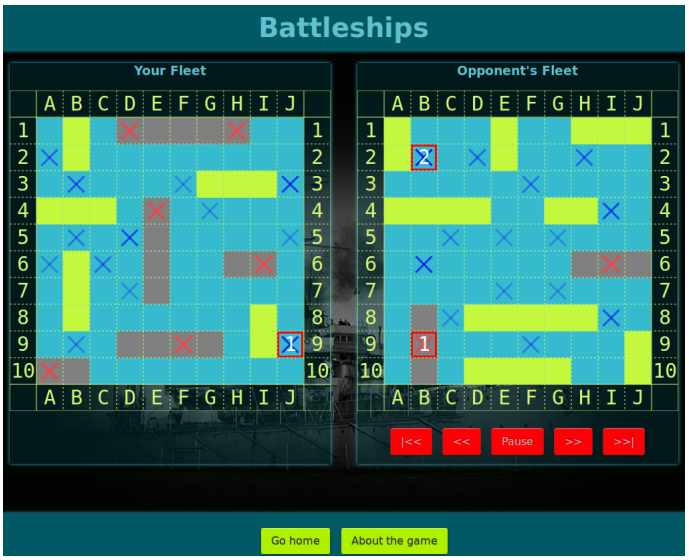

Figure 3: Recap of the game. (animated)

side, we use JavaScript for ship placement, handling of clicks on the boards and viewing the replay. It is important to note here that even the visual representation of the game is created in Haskell. The images of the current game state are generated using diagrams, a DSL for declaratively describing vector graphics. Then they are rendered to SVG and included in the web page. Thus, all the JavaScript code really does is display a sequence of graphics created by Haskell code.

The game state is not stored on the server but is sent back and forth between client and server. To prevent manipulations on the client side the state is encrypted before being exposed.

Of course we could have implemented the game with only HTML and JavaScript which would have made the transfer of the game state between client and server obsolete. On the other hand, that would have meant having to go without everything Haskell has to offer. For instance, the type system not only enforces writing code that works but also offers ways to clarify that the AI is not cheating (see section 3). Besides, Diagrams makes it pleasant to write the rendering code. In a non-declarative language like JavaScript this would have been a lot more frustrating and error-prone. Besides, our goal was not to write another version of Battleships but to learn about the advantages and disadvantages of using a functional programming language for web development. Battleships was merely a means to an end.

3 Demonstrating that the AI Plays Fair

Of course, the AI should play fair and observe the same rules as the human player. Allowing ship movement makes it harder to check the AI for fair play (because one cannot observe its movements), so we tried our best to clarify that the AI cannot “cheat”:

- After the game ends, we produce an animated recap of the course of the game to help the player comprehend what happened at what time.
- We use Haskell’s type system to make it easy to find the places in the code where the AI

does get information and interact with the game logic. This way, it is easier to check the code to convince oneself that the the AI cannot cheat.

More precisely, every AI is an instance of the type class[5] `AI` which is defined as follows:

```
class AI a where
  aiInit     :: (MonadRandom m)
             => Rules -> m (a, FleetPlacement)
  aiFire     :: (MonadRandom m, MonadState a m) => m Pos
  aiResponse :: (MonadRandom m, MonadState a m)
             => Pos -> HitResponse -> m ()
  aiMove     :: (MonadRandom m, MonadState a m)
             => Fleet -> TrackingList -> m (Maybe (ShipID, Movement))
```

On the one hand, Haskell's type system restricts the AI to use the given random number generator (via `MonadRandom`) and its own state (via `MonadState`), in addition to the information given by the game rules (`Rules`), the result of the AI's last shot (`HitResponse`) and the AI's own fleet (`Fleet`) and the shots fired at the AI's fleet (`TrackingList`). The type system prevents the AI from having any other side effects (like IO etc.) and by using Safe Haskell (a subset of Haskell whithout functions to circumvent the type system) this constraint can be enforced by the compiler.

On the other hand, the game engine is polymorphic in the AI type, so it can only interact with the AI via the functions prodived by the `AI` type class. While it may be possible to pass additional information to the AI via these functions by misusing the data types in the parameters, one can easily verify that this is not the case by looking at the places where the four aforementioned functions are invoked.

4 Battleships AI

In addition to a simple AI which basically does everything at random, we implemented a much more sophisticated AI which we are going to describe in some detail here.

4.1 Scoring Cells

When deciding on a cell to fire at, it is necessary to know which cells cannot currently be occupied by a ship. This is the case if we just found out that there is water at a certain position, or just sunk a ship there, or hit a ship at a position that is diagonally adjacent to the cell.

When ships are movable, you often cannot tell for sure whether a cell is blocked or not. Ships may move over water cells or completely sunk ships. Thus, we model the probability for a cell to be water as exponential decay with factor 0.98.

The AI selects its next target by computing a *score* for each position. A high score indicates a high probability to hit a ship at that position, so the AI chooses the position with the highest score as its next target. To make playing more interesting, we added some randomness to calculating the scores. The amount of randomness depends on the selected difficulty level; the scoring method depends on whether ships are movable or not.

[5] Type classes in Haskell correspond very roughly to interfaces in OOP

Immovable Ships. Since ships will always stay at their position, the AI never has to hit a position more than once. Given that all ships have a minimum length of two, it is sufficient to apply a *checkerboard pattern* when searching for ships. For each position on the board the AI considers all potential ships that this position is part of. This way it can score the position according to the likeliness of hitting a ship there.

Movable Ships. In this case there are two phases because the amount of turns is limited: During the beginning, the *search phase*, the AI follows a checkerboard pattern to find (and not sink!) all ships. Sinking is not beneficial at this stage because it allows ships to move around more freely. When the AI has found enough ships or there are only few turns left, the AI switches to the *sink phase* and tries to completely sink all the previously discovered ships, applying the same scoring function as in the immovable case.

4.2 Moving Ships

In each turn, the AI generates all possible movements of all movable ships and chooses one at random, or – also randomly – decides not to move at all.

5 Conclusion

Admittedly, web programming in Haskell still has some minor flaws (especially the use of Template Haskell in combination with a lack of proper documentation sometimes led to difficulties). This, however, is only due to Yesod being still under active development and not a general flaw with web development in Haskell by itself. Then again, Haskell's expressive type system and powerful abstraction mechanism are strong advantages when developing web applications, especially regarding security and maintainability. And using a declarative DSL for graphics simplifies the drawing process, as thinking about what should be drawn is easier than thinking about how it should be drawn.