Functional Programming 2017/2018
Final Project

Alejandro Serrano

In this final project you need to develop your own game in Haskell, using the Gloss library for graphics. This is a larger and more "realistic" assignment than the previous ones, but you have a lot of freedom on what and how to implement it.

1 Introduction

In short, the goal of this final project of *Functional Programming* is to develop a small game with a bit of "action" and a bit of "intelligence" completely in Haskell. The focus is, of course, not in how great the graphics look or whether there is a charming story behind the game. Rather, the grade depends on the general architecture of the project, and in the use of good functional coding practices.

Each project group – we recommend working in pairs – may implement a different game. But not every game is allowed; in particular, games where there any very few "moving parts" to manage – like chess, tic-tac-toe or RPGs – are not allowed. We look for arcade, platform, shooters, sports, race game; all those were there a many items to take care of in the screen.

As stated above, one of the key points of this project is to architect your solution using good programming practices. Because coming up with a good design is a difficult task, we ask you to send a preliminary design document, stating what game you plan to implement and a rough idea of how, during the first week of the practical. Please use those suggestions as input to write nicer code.

2 Requirements

There are some *minimal requirements* that you game must implement in order to get a passing grade. On top of that, the design and coding style is an important part of the final grade. A good game with a good style would receive a grade of 8. In order to receive a higher grade, you may implement *optional requirements*. But note that grades are not linear: in order to improve your grade further, you will have to do increasingly more work.

2.1 Minimal requirements

Player. The player controls (at least) one character, using the keyboard.

Enemies. The game has antagonistic characters in some sense: they try to destroy your ship, win the race, and things like that. You should implement a minimal "intelligence" for those enemies.

Randomness. The game has to include random components: the place where new enemies appear, the place where bonus are located, and things like that.

Animation. Some events should trigger animations which span several frames. For example, distroying an enemy may create an explosion effect.

Pause. It should be possible for the player to pause the game by pressing a key of your choice.

Interaction with the file system. The program must read and write data in one or more files. You are free to decided what those files contain, a typical example would be a list of high scores.

2.2 Optional requirements

Here are some suggestions for additional features and design variations. Each of these requirements, in turn, may be developed in different degrees; the grade will reflect this accordingly. Implementing a requirement at its fullest is worth 1 point. Note that this list of optional requirements is not ordered by any property, in particular not by how difficult it is to implement each feature.

Levels. The game does not have a single screen, but different ones with different characteristics. The player should be able to choose the level from a list.

Different enemies. Add multiple types of enemies, each of them with different appearance and behavior. Make the game more interesting by making "harder" enemies appear later.

Custom levels. Develop a file format which players can use to create new levels for the game. Provide the ability to load one of those files, *in addition* to the default ones, either via a menu in the game or via the command line.

Mouse input. Allow the user to interact with the game using the mouse *in addition* to the keyboard. In contrast to keyboard actions, the effect of clicking on a place in the screen usually involves several frames until the player arrives to the selected location.

Multi-player. Either in co-operative or antagonistic mode, two players play the game at the same time. You may implement this by having each player use part of the keyboard. Be sure that the actions of one player do not kill the actions of the other one!

Networking. Implement multi-player via network communication. You may use the Network library[1]. Chapter 27 of *Real World Haskell*[2] provides a tutorial of this package.

Complex graphics. Refine the graphical part of your game by using techniques such as *parallax scrolling*[3] or by loading images from the file system. But remember: we are interested in your implementation of the techniques, not in the graphical quality *per se*.

Automated testing. Design the game in such a way that you can prove some interesting properties about it using QuickCheck.

Use JSON to save information. Instead of using a home-grown format to keep any interesting information you have in your game, use the standard JSON format to do so. Of course, we do not expect that you create a parser and printer for that format, rather you should use the Aeson[4] library. The documentation for that library has several examples of its intended usage patterns.

2.3 Design document

Before the end of the first week you *have to* submit a design document. The document itself does not have to be very long, but has to include at least:

- A description of your game, which should satisfy all the minimal requirements stated above.

[1]`http://hackage.haskell.org/package/network`
[2]`http://book.realworldhaskell.org/read/sockets-and-syslog.html`
[3]`https://en.wikipedia.org/wiki/Parallax_scrolling`
[4]`http://hackage.haskell.org/package/aeson/docs/Data-Aeson.html`

- A description of the most important data types and type classes that you want to use in your code. Try to follow the guidelines below so we can give you useful feedback.

2.4 Style

An important part of the grade of the final project depends on the way in which you have implemented the game. Keep the following in mind while programming.

Data type definition. Think carefully about how you represent the game data in memory. Create different data types for different goals, as explained during the lectures.

Use abstraction. Remove duplication in your code by means of (higher-order) functions. Use type classes to implement functionality common to several data types.

Separate pure and impure parts. Do not put all functionality in *IO*. Most of the rendering and update functions can be written in a pure style.

Modularity. Split your code into functions, and those functions into modules. Keep their size under control. Feel free to introduce as many modules as required.

Follow good practices. Try to use higher-order functions such as *foldr* or *map* instead of recursion, use pattern matching to define equations, do not use magic numbers.

Document your code. Always write type signatures for all your functions. Explain complex or subtle parts of your program in comments.

2.5 Outcome

Send your assignment via the Submit system in a zip file. This file must contain:

- The original design document, updated to reflect the design that was implemented. Please ensure that this document explains how to play the game and how all the requirements are approached.
- The source code as a Cabal project. Please do *not* include the `dist` folder.

3 Libraries and tooling

We provide an example project using Gloss which just updates the screen with a random number every now and then. You can use this example project as a template for your own code or initialize a new one following the steps in Lecture 8a.[5] In any case, please ensure that you can run your project with the files you submit to us.

3.1 The Gloss library

The Gloss library provides a nice interface to build small 2-D games. It is also a good example of a domain-specific language; in this case for the definition of what to show on the screen. In order to define a game in Gloss you need to write four functions, all of them operating on a shared *state of the world w*, which you ought to define.

- What is the initial state of the world.
- How to turn the world *w* into a *Picture*, which is then displayed in the screen.

[5]`http://www.staff.science.uu.nl/~f100183/fp/slides/fp-08a-project.pdf`

- How to handle user input in the form of *Event*s.
- How to update the state of the world every time some amount of seconds have elapsed.

This design is commonly known as *model-view-controller*. The model consists of the definitions of the involved data types; the view is the function which displays the picture; and the controller how to update that state in response to either time or user interaction. In the example project, each of these components is defined in a different module.

Gloss provides a pure interface. Alas, this makes it difficult to have random elements in the game, and impossible to access the file system. For that reason we suggest you to use the *IO* version of Gloss, which can be found in the module *Graphics.Gloss.Interface.IO.Game*. The key function is *playIO*, which after a bunch of configuration options receives the four aforementioned functions,

$$
\begin{aligned}
&\mathit{initial} :: \mathit{world}\\
&\mathit{view} \;\; :: \mathit{world} \to \mathit{IO}\ \mathit{Picture}\\
&\mathit{input} :: \mathit{Event} \to \mathit{world} \to \mathit{IO}\ \mathit{world}\\
&\mathit{step} \;\; :: \mathit{Float} \;\; \to \mathit{world} \to \mathit{IO}\ \mathit{world}
\end{aligned}
$$

You can look at the *main* function for an example of how to initialize Gloss.

The *Picture* data type, defined in the *Graphics.Gloss.Data.Picture* module,[6] provides several functions to build the entire scene of your game:

- Primitive shapes, such as *circle*, *polygon*, and *text*.
- To change the color of a picture, use the function *color* with one of the colors defined in the module *Graphics.Gloss.Data.Color*. For example, here is the code to display a green `A`:

  color green (*text* `"A"`)

- Transformations over a picture, like *translate* and *rotate*.
- Finally, a way to compose several pictures into another one. Note that pictures are drawn on top of each other, so you need to translate them first. For example:

  picture [*text* `"A"`, *translate* 20 20 (*color green* (*circle* 10))]

A neat way to explore drawing with Gloss is by means of an interactive session with the interpreter.

```
Prelude> import Graphics.Gloss
Prelude Graphics.Gloss> let d = InWindow "example" (800, 600) (0, 0)
Prelude Graphics.Gloss> display d black (color green (circle 100))
```

3.2 Using records

In the lectures you have learnt to access information inside a value by using *pattern matching* and to build new values by calling the *constructors* of the data type. This is very simple, but breaks when we need to handle larger data types with several levels of nesting, as will be the case with the state of your game. For this scenario, Haskell provides a nice feature called *records*.

Definition. A record is like any other Haskell data type, with an additional piece of information: a name for each of the fields. For example, here is the definition of *GameState* in the example project:

```
data GameState = GameState {
                     infoToShow :: InfoToShow
                   , elapsedTime :: Float
                   }
```

For comparison, here is the definition without field names:

```
data GameState = GameState InfoToShow Float
```

[6] `https://hackage.haskell.org/package/gloss/docs/Graphics-Gloss-Data-Picture.html`

Accessing information in a record. Pattern matching is available as usual for records,

$view\ (GameState\ info\ elapsed) = ...$

In addition, records introduce other *three* forms of accessing the information:

1. We can also use the name of the field as a function which retrieves that piece of information,

   $infoToShow :: GameState \to InfoToShow$
   $elapsedTime :: GameState \to Float$

   For example, in the example project we are only interested in the first field to display:

   $view\ gstate = \textbf{case}\ infoToShow\ gstate\ \textbf{of} ...$

2. Pattern matching is extended to retrieve the information of a field using a key/value syntax:

   $view\ GameState\ \{infoToShow = info\} = ...$ -- use 'info' here

   The great advantage is that we only need to include those fields we are interested in, whereas with usual pattern matching we need to give names (or use underscores) for *every* field.

3. We can even make it shorter if we enable the "named field puns" extension by including the following line at the beginning of your file:

   ```
   {-# language NamedFieldPuns #-}
   ```

   Then you do not even need to use *field* = *value* in pattern matching, only the name of the field:

   $view\ GameState\ \{infoToShow\} = ...$ -- use 'infoToShow' here

Updating part of a record. Another problem with normal data types is that if you want to change only one part of a value you are forced to write all the other fields in your source. With records you can tell the compiler to reuse part of a previous record and only "update" one or more fields.

$step\ secs\ gstate = gstate\ \{elapsedTime = elapsedTime\ gstate + secs\}$

In this example, we want to keep the whole game state – we do not even match on it – except for the *elapsedTime*, which is increased by the given amount of seconds. An alternative way to write this function is:

$step\ secs\ gstate@(GameState\ \{elapsedTime\}) = gstate\ \{elapsedTime = elapsedTime + secs\}$

In this case the final *elapsedTime* refers to the value in the original game state, which we obtain using the named field syntax, whereas the other appearances refer to the field name.

Lenses. The problem of accessing larger and nested pieces of data has been "itchy" issue for Haskellers since a long time ago. Records are a simple solution, but many other people use a more powerful notion called *lenses*. If you want to dive into it – but not before having a working game! – look at the packages `lens-tutorial` and `microlens`.