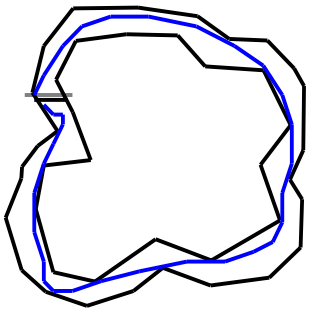
Utrecht Summer School in Computer Science: Applied Functional Programming 2013

Car Racing in Haskell



Jan Rochel <jan@rochel.info>

August 19, 2013

> This assignment is about a pen-and-paper game commonly called Racetrack. It is a car racing simulation which can be played by two or more competing players with pen and graph paper preferably not during lectures about functional programming. In this exercise you will be implementing the rendering and the game mechanics of the simulation.

The rules of the game are simple. On a sheet of graph paper a track is set down simply by free-hand drawing of its boundaries. There is also a finish line and a starting position for the cars. Only the crossings of grid lines are valid positions for the cars. Initialy all cars are on the starting position, then in each turn each player may make one move with his car. The aim is to reach the finish line with as few moves possible without crashing into the boundaries of the track. Each move must obide the laws of physics, which in this game comes down to acceleration of masses. More specifically, this means that the acceleration of a car is bounded such that the velocity of the car can only be altered in each move by changing each vector component of the velocity by at most one.
For more information see: `http://en.wikipedia.org/wiki/Racetrack_(game)`
To play the game online, visit: `http://www.harmmade.com/vectorracer/`

In the end the finished game should look something like this:
The program is invoked with a track definition file as an argument. The file, which describes the track as a set of straight line segments constituting the boundaries of the

track is parsed and displayed. The displaying is done via SVG output to a file which can be viewed during the execution in an external program (e.g. a browser). In each turn the player (we are content with a single player version) is asked in which way to accelerate the car. Once the decision is entered via the keyboard the SVG image is updated redisplay the now one step longer trace that the car has taken so far. If the move crashed the car into the boundaries of the track the game is aborted. If the car crosses the finish line, the game ends and the number of moves is displayed.

1. Implement a module `Types` to represent the entities from the rule description by type synonyms (or data types). Let us start with geometry and physics:

- a *Point* is a two-dimensional vector
- a *Line* connects two points
- *Velocity* is a two-dimensional vector

Type synonyms representing the state of a car:

- *CarState* comprises the current position and velocity of a car
- *Trace* representing a sequence of positions of a car

And a type for the track definition:

- a *Track* consists of 1. a list of lines each representing the boundaries of the track, 2. a starting point, 3. a finish line

Try to make the types as restrictive as possible. The valid positions of a car for instance are restricted to the crossing points of the grid lines. If you require component-wise operations or conversions on these types later on, add appropriate functions to this module.

2 (Parsing)**.** Add to the module `Parser` a function for parsing track definitions.
readTrack :: *String* → *Track*
The format of a track definition file is as follows:

- the first line contains the x- and y-coordinates of the starting point separated by space
- the second line contains four coordinates (order: x_1 y_1 x_2 y_2) separated by space defining the finish line of the track
- the same format holds for the remaining lines each of which defines one of the barriers constituting the boundaries of the track

There are track definition files supplied with this assignment that you can test your parser on. Use *readFile* from the Haskell Prelude to get the contents of the file. To

parse *Int*s and *Float*s you can the following functions defined in the module `Parser` supplied with this assignment:

readFloat :: *String* → *Float*

readInt :: *String* → *Int*

You might also find the functions *lines* and *words* from the Haskell Prelude to be useful for handling strings. You can test your *readTrack* function on the supplied track definition with: *readFile* `"muh.rtr"` ≫= *print* ∘ *readTrack*

3 (Rendering)**.** To display the track and the route of the cars we will produce SVG output. Make sure you have the `blaze-svg` package installed. Executing `cabal update` and then `cabal install blaze-svg` from the terminal should do the job; you might need to restart the interpreter too. Define a module `Render` with the function

raceToSvg :: *FilePath* → *Track* → [*Trace*] → *IO* ()

which will render a given track along with a number of traces as SVG and write it to a file. In order to implement this function use the following functions from the from the supplied `SVG` module:

renderLine :: *String* → *Line* → *Svg*

renders a single line. It takes as a first argument a string representing the colour of the line (e.g. `"white"` or `"green"`) and produces SVG output for the given line. With

linesToSvg :: *FilePath* → *Dimensions* → [*Svg*] → *IO* ()

you can write the collective of all the lines (both constituting the track boundaries as well as the traces of the cars) into a file of the given name. You also need to supply the function with the dimensions of the image. *Dimension* is defined as a four-tuple of *Int*s (*xmin ymin xmax ymax*). Extract the values for the four components from the set of lines to render. They should be the minimum and maximum `x`- and `y`- coordinates of those lines.

4 (Game Mechanics)**.** The module `Mechanics` shall contain the code for checking whether a given move crashes a into a barrier as well as for simulating acceleration of a car. The function

crashes :: *CarState* → *Position* → *Track* → *Bool*

should evaluate to *True* if the car is attempted to be moved onto a position such that the line connecting the previous and the new position crosses any of the barriers of the track. The function can therefore be expressed in terms of

intersects :: *Line* → *Line* → *Bool*

Refer to `http://paulbourke.net/geometry/pointlineplane/` for a simple line segment intersection test. Also add a test for whether a move completes the race:

completes :: *CarState* → *Position* → *Track* → *Bool*

Be reminded that it might be appropriate to add type conversion functions to the `Types` module, in order to implement vector arithmetics.

5 (Game Loop)**.** Implement in your `Main` module the function

gameLoop :: *Track* → *Trace* → *CarState* → *IO* ()

which should roughly proceed as follows:

1. Render the track including the current trace.

2. Ask for user input.

3. Check whether the move crashed the car.

4. Check whether the move completed the race.

5. If none of previous two points hold, recurse.

You can use *getChar* to receive keyboard input. You can map the keys `1 .. 9` (which can be entered via the numpad) to the nine possible accelerations.

6 (Invocation)**.** Implement a *main* function in your `Main` module to initiate the game loop. Use *getArgs* from `System.Environment` to retrieve command-line arguments.

Remark: Do not settle for something that just works, but code that is appealing to the eye. Beautiful code is usually concise and documents itself.

7 (Bonuses)**.** You can earn bonus points by implementing additional features, as for example:

- Multiple players
- Cars crashing into another
- Improve the visualisation
- A solver, which computes the optimal route