Dominosa Solver
A Haskell and Java comparison

Anne-Greeth van Herwijnen

October 13, 2017

1 Introduction

The goal of this assignment is to program a solver for 8 x 7 grids which contains domino stones (called bones). This solver should be built with Haskell and Java to be able to compare these two implementations, because the nature of these languages differs a lot.

1.1 Definitions

Bone	A domino stone, described by its value and the pips
Pips	The dots on a bone, for example (4,5)
Square	A number of the field and its position in the array
Field	An array that describes the input grid as one long array and is used to represent the solution
Grid	A list of squares
Stone	A description of two squares, combined to represent the pips and the position in the field

Table 1: Definition of used words

2 Algorithm

To solve this problem I took a naive and simple approach. You can describe the solution space as a combination of all the different options. An option is a combination of two values on the grid that are horizontally or vertically connected. The algorithm takes the original grid, all the options and the available bones. It creates a tree that gradually fills the grid with the values of the bones. It takes the first bone from the set of available bones and finds all the options that have the same pips. Each of these options creates a branch from their parent. The bone value is placed in the original grid, the options that are no longer possible are removed from the options, and the bone is removed from the set of available bones. If the bone cannot be placed on the grid because no option contains those amount of pips the grid cannot be solved. If all the bones are placed it's a valid solution. These valid solutions are always leaves.

3 How-To

The solvers are in the folders `Haskell` and `Java`, these solvers solve 8 x 7 grids. The solvers in the folders that are appended with `-small` can solve a 3 x 4 grid. The programming language of the solvers correspond with the name of the folder. To run the solvers, navigate to the specific folder and follow the language specific instructions below.

3.1 Haskell

To run the solver in Haskell, you load `domino.hs` with ghci and you type `solveAll` to solve all the grids from the assignment. The restriction is that this code only works for 8 x 7 grids. For a faster solver you can load `domino-improved.hs` and use the same command. This algorithm is extended with the removal of the bones that can only be placed once on the board, called `uniques`. This addition speeds up the algorithm to take half the time. Figure 1 shows the partial output of the `solveAll` function.

```
Grid no.3                                 Solution(s):
  6   6   2   6   5   2   4   1             16  16  24  18  18  20  12  11
  1   3   2   0   1   0   3   4              6   6  24  10  10  20  12  11
  1   3   2   4   6   6   5   4              8  15  15   3   3  17  14  14
  1   0   4   3   2   1   1   2              8   5   5   2  19  17  28  26
  5   1   3   6   0   4   5   5             23   1  13   2  19   7  28  26
  5   5   4   0   2   6   0   3             23   1  13  25  25   7  21   4
  6   0   5   3   4   2   0   3             27  27  22  22   9   9  21   4

                                            16  16  24  18  18  20  12  11
                                             6   6  24  10  10  20  12  11
                                             8  15  15   3   3  17  14  14
                                             8   5   5   2  19  17  28  26
                                            23   1  13   2  19   7  28  26
                                            23   1  13  25  25   7   4   4
                                            27  27  22  22   9   9  21  21
```

Figure 1: The partial output of `domino.hs`

3.2 Java

To run the solver in Java you run the `Main` class. This will solve the same grids as the Haskell `solveAll` function. The `TestSolver` class contains three different grids, of which one grid is corrupt.

4 Comparison

The comparison between an object-oriented language like Java and a functional language like Haskell leads to obvious differences. The clarity of the Java implementation comes from the resemblance to the world. You can imagine what a board is and how functions interact with a field. However, the code is bloated by boilerplate code like obvious getters and setter. On the contrary, Haskell's clarity comes from the obvious recursion steps and use of base cases. Even though the code becomes more complex because things can't be stored in a public variable but have to be passed around through all the functions. The experience I have with Java helped especially in writing tests, which made it easier to test the functions. In Haskell it's easier to just test a single function from the console by providing it with input and checking the output. This allowed me to quickly build the building blocks for the solver in Haskell.

The more efficient algorithm as implemented in Haskell was relatively straightforward. I already wrote the `uniques` function, even when I did not use it in `domino.hs`. However replicating this in Java would be more complex and bloated, because in Haskell the standard functions like `elem`, `map` and `filter` could be used directly on the tuples. The Java code misses tuples, which would mean to compare the options I would need to override the `equals` function to be able to compare the options in an efficient way.

5 Improvements

There are still improvements that can be made to both implementations, the first is the flexibility of the implementations. Both implementations can create bones dynamically if you provide them with the input. This could also be derived from the input grid or from the grid size, which would require the user to provide only the grid. Both solvers can only solve 8x7 grids and are not dynamic in this respect. Another improvement could be to find a way to check the neighbours efficiently. I've left out the neighbour check because it would require a lot of calculations and a more complex tracking of the placed stones. If you can check if you cut of one of your neighbours you can cut off that branch sooner. The last, and probably most rewarding improvement would be to find a way not to have to traverse over the tree twice. Once to construct it and once to find the solutions. So if the solutions could be passed along, there would be no need for the whole keeping track of the tree and the search for the solutions.