Functional Programming – Domino Effect

Martijn Slot

October 15, 2017

Structure of assignment

This assignment was part of the 'Critical Software Developer' Module of Nedap University, on functional programming. We were asked to design and implement an algorithm for the domino problem; e.g. fitting a set of dominoes to an input grid on corresponding 'pips' (or numbers, if you like). As a general comment: I am aware that there is some unfinished stuff in my code. I've spent a lot of time on getting the solve algorithm right, which seems to work very well in Haskell, but the output is not entirely as described in the assignment.

Algorithm

I tried to implement two different algorithms with Haskell. The first one being unnecessary intricate and difficult to implement, so I quit this tree and tried to fix the domino problem differently. (this code is not part of the deliverables, but I can show it to you if for some reason you might be interested) The second algorithm, that I eventually implemented in both Java and Haskell, was based on a recursive binary algorithm.

Input variables are the puzzle grid (input), the result grid (fitting of the dominoes) and a list of dominoes (also called bones). The algorithm was done when all bones were fitted on the puzzle grid and the result grid had all zeroes replaced by bone numbers. So every two adjacent grid locations have to be filled with a single bone. This means a singe grid location has at least two possibilities for a bone to fit. For example, on location *(1,1)* (the top left grid field) the bone can only be placed on locations *{(1,1),(1,2)} and {(1,1),(2,1)}*. If we recursively repeat placing bones in these two configurations on the rest of the grid locations: bam, a binary tree is born. The solver will recursively call itself until the list of stones is 'empty' (success!) or there are still empty fields on the grid and no possibilities for any bones to be placed.

Running the code

Haskell (using ghci)

1. Prelude> :l dominoeffect.hs
2. Main> solver *n* (where n is input board number)

If *n* is 1, it will solve input board 1, for anything else it will solve input board 2. sidenote: there is a bug in my IO () function with Haskell, it displays the grid slightly shifted.

Java

1. Open java project
2. Run main

There's no option to change input grid but for changing the input parameters in main code.

Java and Haskell implementations

I've tried to implement exactly the same algorithms and functionality in the Java and Haskell implementations. This makes my Java code a bit strange and still pretty functional, as far as Java can be functional with using for example lambda functions like foreach loops. And Java is a bitch with recursive functions. Java functions will store all intermediate results in memory which will lead to stack overflow when complicated recursive computations. Therefore my Java function will not work on my mac with input boards containing more than 20 grids. Im very positive it'll work on a bigger computer, but this signifies the functional differences between Java and Haskell; Java is not made for recursiveness as opposed to Haskell, because it is an interpreter language, that handles recursive functions much better because no intermediate code is generated. A side effect of calling recursive functions in Java is that a 'deepCopy' function is needed because Apache in all its wisdom designed Java to work with references to memory.

There are also other differences between Java and Haskell implementations like the number of code lines. The Haskell solution is about 30% of the size of the Java solution, which is mainly due to boilerplate code that is limited in Haskell, and relatively large in Java. But also the algorithm and all supporting functions are slightly shorter and clearer in Haskell; using guards and pattern matching is shorter than Java switches or other conditionals/loops. I do find Java to be more structured and clear, but that might also be because I'm more familiar with Java. Testing in Java is also much easier than in Haskell. I encountered some problems with testing Haskell code in GHCI. It is possible to test functions separately, but this

is usually done by hand. Though it is possible to automate testing with separate surrounding functions that check input and output.

Because Java is a compiler language and Haskell is a Interpreter language some there are also some differences in performance. Haskell is much faster (getting results as low as 0.06s) as opposed to Java that is around 10 times slower when running the same input grid. Because the Haskell program is translated line-by-line, I found it easy to determine errors and mistakes (all though some error messages could be clearer), but without using a run time debugger it was hard to debug recursive functions. I had encountered an error that possibly occurred in the 6th recursive cycle, and that I eventually found by performing all recursiveness on step at a time by hand.