Learning, unlearning, and learning again; dominoes in Haskell and Java

Lars Ran

September 23, 2021

"Don't you hate code that's not properly indented? Making it part of the syntax guarantees that all code is properly indented!"

– G. van Rossum

Introduction

For this assignment I created two programs, one in Haskell and one in Java, solving a problem involving dominoes. Roughly put, the assignment came down to finding all solutions in a decision tree consisting of decision where to put a certain domino tile. Here I will present a small comparison of using Haskell and Java for this. More abstractly this can be seen as a comparison of using functional programming or object-oriented programming to solve this problem.

Shifting perspectives

It was great fun to learn program in Haskell. It is much more like math, in that everything is immutable which makes reasoning about the program way easier. This made my transition to programming in Haskell quite smooth. Some recursions can be real mind-benders but this makes the programming exciting and challenging but also rewarding. One trick that I used is to make sure that I never thought about any of the exercises in Java code. I found it easier to think "natively" than to translate. The real surprise came to me when I was finished with the Haskell program and started on the Java part. I really struggled for a couple of hours to think and program in Java again. What should be an object, what not, using for loops and above all, all the missing semicolons. So it took me some time to get used to Java again. Having done this does give extra spice to programming though, and I see a whole lot more potential Java streams now.

Code properties

The biggest difference I found are the aforementioned immutable variables. They are your best friends but they do make some things harder. In my Java program I could take the liberty to keep track of state without much thinking. In Haskell I struggled with keeping track of certain things that eventually turned out to be not needed. For this reason Java programming feels like it has a bit more options to play with. Recursion and list comprehensions are very strong in Haskell and I missed those in traditional Java programming. Of course Java has streams now which helps a lot but I tried not to use them to exemplify the difference (otherwise the whole "Board" class in the Java program would have consisted of streams). The idea of "moving" through lists in Haskell is really clear and I did not get this feeling in Java. On the other hand, one of the optimization steps I took was picking a domino with the least options and use it for an action. In Java this was easy, picking the minimum "et voila". In Haskell however, this made me sort the whole list so that the element was in front and I could use pattern matching. It feels a bit overkill but I am still not sure how I could have done this better.

Performance

As a first note, this problem was basically an exhaustive search, so this comparison is not totally fair. However I had the feeling that writing in Haskell made me think of efficient solutions automatically. It is almost that if the Haskell code is correct then it is quite likely an efficient implementation of that specific algorithm (I know this sounds tautological but I think the idea is clear). Whereas in Java you can be tricked into looping over complete lists or being inefficient with states quite fast. Surprisingly, I must say that I did not expect Java to be as fast as it was. Haskell, interpreted, ran the program in 6.48 seconds to solve a board with 70004 solutions. I was astonished that it took Java only 947 milliseconds. But then again, Java was compiled. If we compile our Haskell program (with -O2) then it takes 328 milliseconds. So Haskell has an advantage here. This could be due to stored function results which can occur often in an exhaustive search which is a great benefit in Haskell that is not possible in Java due to unpureness.

Code confidence

In Java it is easy to have confidence in your code, and if you do not then you write some more tests or print some debug lines. In contrast Haskell can feel somewhat magical sometimes. Often when it compiles, it works (or is off by a lot). But it sometimes feels that somewhere along the function call on function call on function call there can go something wrong ruining the process. What worked for me is going through the whole program in my head again to gain confidence in the correct working of the program. This is annoying but it might wear off after some time programming in Haskell. For now, it is definitely there.

Conclusion

All in all I am happy that I learned about Haskell. I think it is great for algorithmic problems and it definitely deserves a space in my toolkit. The only downside I found is that I cannot yet see how Haskell would be used in a product. The upshot is though that I can apply all this knowledge to Java streams and/or Kotlin in the future and I definitely will. I have to watch out though, the bug that took me the longest to fix in this Java program was where I made a shallow copy of a HashMap instead of a deep one. Immutable variables are only your friends if they are indeed immutable.