Takuzu (Haskell) Project

Emanuel Stöckli
emanuel.stoeckli@gmail.com

Vahid Shirvani
vahid.shirvani.4301@student.uu.se

Introduction

We have chosen to implement the board game Takuzu in Haskell. The start phase of our project was characterized by a workday of pair programming in order to get to know each other and to build a good and reliable base structure of our code. Pair programming but also the usage of GitHub proved to be the key of our successful teamwork. We've not only created a repository but also issues for all upcoming tasks. This allowed us to be always up-to-date if one of us implemented something at home and we were also able to see how those changes affect certain issues. After implementing the whole base structure and some of the basic rules together, Vahid implemented the rest of the rules while Emanuel came up with cabal, HUnit and shUnit2 testing. He also started to implement the Sudoku part. The division of labor within the group was always balanced.

I. Design Question

How to solve a Takuzu puzzle iterativly?

Our iterative method to solve Takuzu puzzles is based on the iterative application of a bunch of rules that can be described in detail in the given source of ConceptisPuzzles [1].

As long as the game is not finished we apply a certain rule until the board does not change anymore. This starts with some easy steps, the so-called basic techniques. The first rule considers that no more than two consecutive cells can have the same value 1 or 0. This allows us look for pairs of cells with the same value and for triples where the middle cell is empty. This matters not only for each row but also for each column. Furthermore there has to be an equal number of 0's and 1's and there must not be two identical rows and columns.
It is worth mentioning that our functions are designed to solve the board row-wise. We simply transpose the whole board and treat every column just like a row. Once we are done with all the rows, we transpose the board back to its original setting. So every time we refer to rows in this report, that will also include columns as they are just transposed into rows.

Another design decision worth mentioning is related to efficiency. In order for our solver to be more efficient, we have preferred to use recursion (with accumulators) in favor of list comprehension in many functions. The key difference is that our recursive calls would return as soon as they find the answer unlike list comprehension that would only return once it's done iterating through all the elements in the give list. So the average complexity of our recursive calls are $< O(N)$ in contrast to list comprehension which is always $O(N)$.

II. Project Skeleton

To sum up, our project is based on four main pillars.

1. *Main*
   handles all the IO, calls the game specific functions and provides the output.
2. *AbstractGameLogic*
   defines the board and a bunch of generic high-order functions that are used in both games, e.g. to apply specific functions to the board
3. *TicTacLogic*
   provides Tazuku specific code
4. *SudokuLogic*
   provides Sudoku specific code

[1] `http://www.conceptispuzzles.com/index.aspx?uri=puzzle/tic-tac-logic/techniques`

III. Extra Requirements

I. Testing Functions

We wanted ensure quite early that changes on our code do not negatively affect properly working functionality that has already been impemented. Therefore we enriched our setup with unit tests. We've checked out the widely used frameworks "test-framework", "HTF" and "tasty". In the end we've chosen tasty because we found most of the tutorials and it seems to work with a variety of tests.

I.1 Cabal

By adding a testing framework and one or more testing libraries to our project, dependencies will also increase at the same time. As we want to make sure that any third party developer can understand, install and run our code we have to write those dependencies down in some way. This is one reason why we make use of Cabal [2]. On the one hand side it can be used to for building our project and on the other hand side for packaging. It would even be possible to build a documentation using cabal in combination with haddock. Howerver, in our *main.cabal* file we describe what our project is and what dependencies it has.

```
name:           Tic-Tac-Logic
author:         Emanuel Stckli, Vahid Shirvani
version:        1.0
Build-Type:     Simple
cabal-version:  >= 1.2

executable tic-tac-logic
   main-is:       Main.hs
   build-depends: base

Test-Suite test
   type:             exitcode-stdio-1.0
   main-is:          TicTacLogicTest.hs
   build-depends:    base,
                     tasty,
                     tasty-hunit
```

Every developer who is going to checkout our repository can see both, the dependencies of our test suite and the the one from our main project. As illustrated in the following code terminal snipped it is quite easy to install the project with all the dependencies and also to build the project and the tests using cabal:

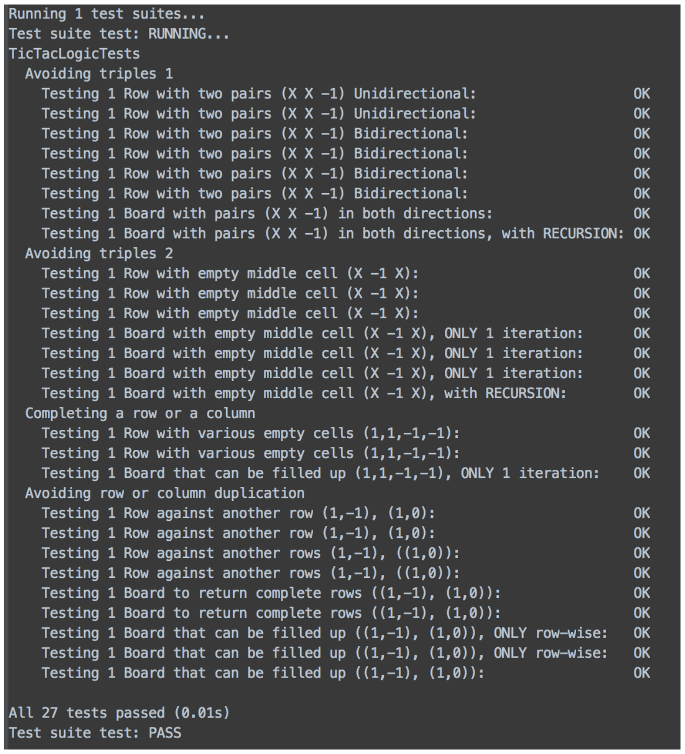
```
$ cabal sandbox init
$ cabal install --enable-tests

$ cabal build
$ cabal run

$ cabal build test
$ cabal test --show-details=always
```

I.2 HUnit Testing

As we both know Java and JUnit we started to use HUnit [3] and implemented tests for every rule we implemented. Tasty allows us to group multiple test cases in a test hierarchy. As shown in figure 1 we created a test group for every rule we have implemented. This allows us to change and optimize the rules and to easily check if the rule still provides proper results. In total, we have implemented 43 unit tests for Takuzu and 3 for Sudoku.



```
Running 1 test suites...
Test suite test: RUNNING...
TicTacLogicTests
  Avoiding triples 1
    Testing 1 Row with two pairs (X X -1) Unidirectional:                 OK
    Testing 1 Row with two pairs (X X -1) Unidirectional:                 OK
    Testing 1 Row with two pairs (X X -1) Bidirectional:                  OK
    Testing 1 Row with two pairs (X X -1) Bidirectional:                  OK
    Testing 1 Row with two pairs (X X -1) Bidirectional:                  OK
    Testing 1 Row with two pairs (X X -1) Bidirectional:                  OK
    Testing 1 Board with pairs (X X -1) in both directions:               OK
    Testing 1 Board with pairs (X X -1) in both directions, with RECURSION: OK
  Avoiding triples 2
    Testing 1 Row with empty middle cell (X -1 X):                        OK
    Testing 1 Row with empty middle cell (X -1 X):                        OK
    Testing 1 Row with empty middle cell (X -1 X):                        OK
    Testing 1 Board with empty middle cell (X -1 X), ONLY 1 iteration:    OK
    Testing 1 Board with empty middle cell (X -1 X), ONLY 1 iteration:    OK
    Testing 1 Board with empty middle cell (X -1 X), ONLY 1 iteration:    OK
    Testing 1 Board with empty middle cell (X -1 X), with RECURSION:      OK
  Completing a row or a column
    Testing 1 Row with various empty cells (1,1,-1,-1):                   OK
    Testing 1 Row with various empty cells (1,1,-1,-1):                   OK
    Testing 1 Board that can be filled up (1,1,-1,-1), ONLY 1 iteration:  OK
  Avoiding row or column duplication
    Testing 1 Row against another row (1,-1), (1,0):                      OK
    Testing 1 Row against another row (1,-1), (1,0):                      OK
    Testing 1 Row against another rows (1,-1), ((1,0)):                   OK
    Testing 1 Row against another rows (1,-1), ((1,0)):                   OK
    Testing 1 Board to return complete rows ((1,-1), (1,0)):              OK
    Testing 1 Board to return complete rows ((1,-1), (1,0)):              OK
    Testing 1 Board that can be filled up ((1,-1), (1,0)), ONLY row-wise: OK
    Testing 1 Board that can be filled up ((1,-1), (1,0)), ONLY row-wise: OK
    Testing 1 Board that can be filled up ((1,-1), (1,0)):                OK

All 27 tests passed (0.01s)
Test suite test: PASS
```

Figure 1: *HUnit Tests*

Test cases could also be implemented with other testing libraries, for example QuickCheck or SmallCheck. We focused on HUnit tests because we wanted to

[2]https://www.haskell.org/cabal/

[3]https://hackage.haskell.org/package/HUnit

check for particular cases. Every rule implements a specific case and we want to make sure that those particular cases are handled correctly. As an alternative, QuickCheck would have allowed us to create randomized tests. This is really useful for type based testing but for example the rule that checks for $(X, X, -1)$ rows only has to handle a reasonable number of different row types, thus it was clearer to specifically define those cases using HUnit tests.

I.3 HLint Code Convetions

While reading about Haskell testing frameworks in a blog that presents a Haskell project skeleton [4] we also stumpled upon HLint. This package knows a lot about code conventions and makes sure you hold them. With the use of this tiny little helper we found a lot, which means more than 80, ways to improve our code style. This also includes hints where guards might be better than if-else-structures.

I.4 Black-blox testing with shUnit2

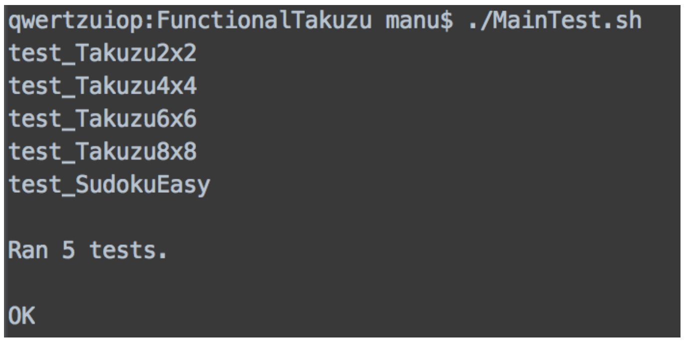
After implementing and testing all Tazuku rules in detail we wanted to make sure that our program in its entirety works as required. This leads to our decision to implement a system black-box test. The way how the program will be used was given in the project description:

```
$ ghc tic-tac-logic.hs
$ cat tic-tac.txt | ./tic-tac-logic
```

We decided to test our program the way it will be used later, which means with IO and .txt files. The implementation of the tests is based on the shUnit2 shell unit testing framework [5]. A nice tutorial of shUnit2 can be found here [6].

In order to generate boards we used a random Takuzu board generator that can be found on GitHub [7]. We added an example board of every size to our code base. You find it in the directory with the name *mainTests*. After building the code with cabal the bash script in the *MainTest.sh* file can easily be executed in the terminal (see figure 2).

As all puzzles within the context of this Advanced Functional Programming Project HT14 will have unique solutions we didn't implement a solver for ambiguous puzzles. Not surprisingly, we found a few ambiguous puzzles without solution while generating random puzzles. But all puzzles with unique solutions worked fine.



```
qwertzuiop:FunctionalTakuzu manu$ ./MainTest.sh
test_Takuzu2x2
test_Takuzu4x4
test_Takuzu6x6
test_Takuzu8x8
test_SudokuEasy

Ran 5 tests.

OK
```

Figure 2: *shUnit2 Tests*

II. Reasoning Techniques

Basic Technique 5

We start by creating a list of all the complete rows in the board. Next we go through the board and identify the rows that only have two empty cells left. At this point we compare the non-completed row with all the completed rows one by one. If the two rows are similar everywhere apart from empty cells then we know the values that should be set in the empty cells. It should be the opposite of the other row otherwise both of them would end up being same which is illegal.

Advanced Technique 1

Although this rule was among the most trickiest ones to implement the core concept of it is quite simple. We simply try to found out the cell value by making a wrong move. So we make a guess and draw a conclusion from the consequences. This rule is very similar to the Avoid Triples 3 rule, it just takes that rule one step further.

Here we are only interested in rows that are two steps away from being complete. For example a

[4]http://taylor.fausak.me/2014/03/04/haskeleton-a-haskell-project-skeleton/
[5]https://code.google.com/p/shunit2
[6]http://code.tutsplus.com/tutorials/test-driving-shell-scripts--net-31487
[7]https://github.com/geekster777/takuzu-generator

row with length of 10 which has 3 X:s is within our interest because the row only needs 2 more X:s in order to be complete. So we go though the board trying to find a row that meets our criteria. We put a value in one of the empty cells and then fill the rest of the row. The filling of the reaming cells can be done in many different ways. We need to try all the possible combinations. If all combinations result in invalid boards, then we know the right value for our initial guessed cell; It's the opposite of what we initially guessed.

Advanced Technique 2

This rule shares code with the Avoiding Triples 3 rule because of similarities. We are only interested in rows that are one steps away from being complete in both rules. We put a value in one of the empty cells and then fill the rest of the row (no different combinations this time, only one certain way to fill). At this stage, the board will be sent for verification. If the board is invalid then the right value is the opposite of what we guessed. Both rules got merged because the verification process checks for both triple values as well as duplicate rows.

III. Sudoku

In contrast to Takuzu all the Sudoku puzzles have 9 rows and 9 columns. Furthermore, rules are going to be applied not only on rows and columsn but also on 3x3 sub boards. Every sub board must contain all numbers from 1 to 9 and no duplicates. A plethora of things, however, be integrated between Takuzu and Sudoku. We still use the same general row-based approach. Therefore we implemented a mapper function that maps each 3x3 sub board to a row which results in a board of 9 rows of 3x3 sub board. We simply apply all rules we want and map the board back to its original form. This means we can still use all the *AbstractGameLogic* high-order functions like *applyOnceInBothDirections*, *runRule* and *solve*.

After implementing this base structure it is only about implementing Sudoku specific rules that can be passed to the generic solver. To prove that, we have implemented a first rule that allows us to solve easy Sudokus by completing rows, columns and 3x3 sub boards iteratively as long as the game board changes. You can find a test input in the *mainTests* folder as well.

IV. Implementation Highlights

There were several design decisions that we made early on, which helped us a lot during development and made our code of higher quality.

Single responsibility principle

Both of us agreed that we should avoid creating functions that have several responsibilities. This required us to create many small functions that we later on used in our rule implementation functions. This design made reasoning and eventually understating of the code much more easier. As every small function became a black box, debugging those functions also became easier. We just had to unit test each functionality before using it as a fundation in other parts of the system.

Higher-order function

It's always a good idea to reuse as much of you code in other parts of the system as much as possible in development. For example you should avoid copy and pasting code into other sections when possible. Although Single responsibility principle will provide you with this to a certain degree, but it's not simply enough. That's where Higher-order functions come in. In a language like Haskell where you have the ability to pass functions to other functions just like data, this will help you minimize the code repetition to a large extend. This is a feature that you will get used to very fast and will miss during programming in other languages that doesn't treat functions as first class citizens. We have used higher-order functions in several parts of our implementation, especially in the *AbstractGameLogic* part.

Columns are just as rows

This might seem a small thing but it played a large roll in reducing the amount of code. We don't have any dedicated function that operates on columns or 3x3 sub boards of the Sudoku. All the functions were designed to only handle rows. In order to manipulate columns, we simply transpose the board before passing it forward.

Unit testing

Testing is always as much important as the development itself. Although not many points were dedicated to this section in the project in contrast to how much effort went in to it, this part played a crucial roll. Several bugs were caught which in turn result in more polished code than it would have been without it. We have both testing on functional as well as program level.

Complexity

The complexity was an important factor that we always considered during the implementation of all functions. It wasn't about just coding a function that works at any cost, the performance efficiency was as important as the functionality itself. We tried to avoid functions that had quadratic complexity when possible. One way to achieve this was to always prefer tail recursion with accumulators over other possible implementations. There were some parts were we favoured recursion over list comprehension to get $< O(N)$ on average instead of $O(N)$.