N-Queens Analysis

Brennan W. Fieck
Nicholas Zustak
Department of Electrical Engineering and Computer Science
Colorado School of Mines

CSCI400 - Programming Languages November 2016

Comparison

Due in no small part to the fact that we used a brute-force approach, the strength of Haskell that really stood out was the list-comprehension that allowed speedy construction of arbitrarily-dimensioned lists. This really cannot be understated enough; Haskell's suite of list-structured manipulation tools is staggering both in scale and power. On the other hand, it's always annoying to attempt I/O in Haskell; it always feels as though we've ceased to use pure functional programming (though this isn't the case). Also, because functions are "first-class citizens" in Haskell, associating a given function with its respective arguments nearly always feels unwieldy, in the worst cases resulting in a cacophony of parenthesis (or a slightly smaller cacophony of `$`'s and/or `.`'s). It's also somewhat awkward to attempt to store intermediary results in Haskell, which also seems to break the pure functionality, or at the very least run counter to intuition and feel strange to implement.

By contrast, the Java approach uses well-broken-up statements and method calls to sequentially execute very basic operations which, as a whole, lead to the solutions. This means avoiding the parenthesis jungle and an ease of reading born of small statements spaced out vertically. This program used a different approach than our Haskell program. First of all, rather than store the solutions as a set of boards (`[[Bool]]`), the programmers recognized that because each row and column must contain no more than 1 queen, a solution can be fully described as a list of columns where each column contains the number of the row a queen may be found on, thus storing the solution in a 1-dimensional array. Because of this, it's not as easy to see that the major shortcoming of the Java approach is the fact that initializing an arbitrarily-dimensioned, non-trivial list/array structure is rather difficult, and working with such structures can also be more annoying than doing so in Haskell. For instance, the simple line of Haskell: `map (someFn) [0..10]` which applies a function (`someFn`) to the natural numbers on the interval from 0 to 10 (inclusive) would usually take the form of a `for` loop in Java, traditionally 2 to 4 lines. Additionally, these lines wouldn't necessarily share the usual property of being simpler/shorter than their Haskell equivalent; a `for` loop statement itself would likely span further on a line than the entire Haskell statement itself!

It's also worth mentioning that Haskell's lazy execution and downright exhaustive standard library also make things such as transposing a matrix or reversing a list utterly trivial, and while Java has equivalents for some of these functions, it does not fully replicate the flexibility (nor indeed even extensiveness) that such functions have in a language where functions are first-class.

Were we tasked with making the Java approach more "functional", we would likely do so largely by changing procedural `for` loops to mappings or filters. For example, in the function `placeNQueens`, one could use Java8's `Collections` library to rewrite the function as shown in Listing A.3. `canPlaceQueen` could be similarly be rewritten to turn the for loop into a filter or map, and `printQueens` could be written as a map of "`map`" onto a 2D array, with the inner mapping itself a map of `System.out.print`.

Our group agrees unanimously that lambda and delegate functions are a welcome addition to any object-oriented or procedural language, and the ability to pass functions around as values/inputs to other functions is invaluable. We also agree that it would do much more harm than good, however, to replace procedural and object-oriented programming entirely with pure functional programming, because the abilities to abstract datatypes, utilize inheritance, and define sequential flow are also fundamental tools for any programmer.

Source Code Listings

```haskell
import Control.Monad
import Data.List

--prints an individual solution
printBoard board = do
    mapM_ (putStrLn) [[ if entry == True then 'Q' else 'X' | entry <-
        row] | row <- board]
    putStrLn ""


--checks if a board layout is a solution by verifying that no one
   queen can attack another
isSln board = (validateRow board) && (validateCol board) && (
   validateDiag board)

--these three functions validate that queens cannot attack each other
--laterally, vertically, or diagonally
validateRow board = not (any (\row -> length (filter (==True) row) >
   1) board)
validateCol board = validateRow (map reverse (transpose board))
validateDiag board = validateRow (allDiagonals board)

-- ctc Tobias Weck, url: http://stackoverflow.com/questions/32465776/
   getting-all-the-diagonals-of-a-matrix-in-haskell
diagonals :: [[a]] -> [[a]]
diagonals []        = []
diagonals ([]:xss) = xss
diagonals xss       = zipWith (++) (map ((:[]) . head) xss ++ repeat [
   ]) ([]:(diagonals (map tail xss)))

--gets all of the diagonals of the board
allDiagonals board = (diagonals board) ++ (diagonals (map reverse (
   transpose board)))
```

```
--generates every possible row of a given size (helpful because each
   row can only contain 1 queen)
possibleRows length = [(replicate (num - 1) False) ++ (True:(
   replicate (length - num) False)) | num <- [1..length]]

--gets every possible configuration for a given size of n-queens
allBoards size = sequence (replicate size (possibleRows size))

--asks the user for the board size, then attempts to solve n-queens
--on that board, printing the solutions (or "No solutions found." if
   none)
nQueens = do
    putStrLn "Enter Board Size:"
    n <- readLn :: IO Int
    let solution = (filter (isSln) (allBoards n))
    mapM_ (printBoard) solution
    unless ((length solution) > 0) (putStrLn "No solutions found.")

--outputs only the number of solutions for a user-specified board
   size
nQueensNumSoln = do
    putStrLn "Enter Board Size:"
    n <- readLn :: IO Int
    print (length (filter (isSln) (allBoards n)))

--tests the two main functions
main = do
    putStrLn "Testing nQueens..."
    nQueens
    putStrLn "Testing nQueensNumSoln..."
    nQueensNumSoln
```

Listing A.1 Haskell Source

```java
/**
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package queens;

/**
 *
 * @author ACHCHUTHAN
 */
public class Queens {

    int[] x;

    public Queens(int N) {
        x = new int[N];
    }

    public boolean canPlaceQueen(int r, int c) {
        /**
         * Returns TRUE if a queen can be placed in
         * row r and column c otherwise it returns FALSE
         * x is a global array whose first (r-1) values
         * have been set
         */
        for (int i = 0; i < r; i++) {
            if (x[i] == c || (i-r) == (x[i]-c) ||(i-r) == (c-x[i]))
            {
                return false;
            }
        }
        return true;
    }

    public void printQueens(int[] x) {
        int N = x.length;
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                if (x[i] == j) {
                    System.out.print("Q ");
                } else {
                    System.out.print("* ");
                }
            }
```

```
            System.out.println();
        }
        System.out.println();
    }

    public void placeNqueens(int r, int n) {
        /**
         * Using backtracking this method prints all
         * possible placements of n queens on an n x n
         * chessboard so that they can't attack one another
         */
        for (int c = 0; c < n; c++) {
            if (canPlaceQueen(r, c)) {
                x[r] = c;
                if (r == n - 1) {
                    printQueens(x);
                } else {
                    placeNqueens(r + 1, n);
                }
            }
        }
    }

    public void callplaceNqueens() {
        placeNqueens(0, x.length);
    }

    public static void main(String args[]) {
        Queens Q = new Queens(8);
        Q.callplaceNqueens();

    }
}
```

Listing A.2 OO Source (Java)

Source located here. Code has been formatted to fit page.

```java
public void placeNqueens(int r, int n) {
    IntStream.rangeClosed(0,n-1).toArray().filter(c -> {
        canPlaceQueen(r,c);
    }).forEach(c -> {
        x[r]=c;
        if(r==n-1){
          printQueens(x);
        } else {
          placeNqueens(r+1,n);
        }
    });
}
```

Listing A.3 Functional ’placeNqueens’