EDAF40: Lab 4
The interactive Sudoku solver, ver. 1.3

Adrian Roth and Jacek Malec

May 17, 2019

The goal of this lab is to figure out how to write an interactive Sudoku puzzle solver in Haskell. In the process we will learn about monadic computations and the bind function together with the somewhat tricky IO concept in Haskell. All functions written in this lab are expected to guide you towards one possible way of solving Assignment 2F, though there are definitely many other approaches. If you find an approach which you think is more logical and easier to understand than our version, we would be happy to hear about it!

Part 1: Reusing earlier concepts

The struggles you went through in the first assignment will now pay off as you will reuse the concepts and your understanding of peers and squares. At *EDAF40 Lab material* menu on the course web page you can find and download a `http://fileadmin.cs.lth.se/cs/Education/EDAN40/classes/SolveSudoku.hs` file containing an implementation of these structures and some additional stuff such as the `parseGrid` function. The `parseGrid` function is not yet working since `assign` function is not implemented yet: you will think more about it later today.

First consider the type `Grid` that represents a partially filled puzzle (the one you have been verifying in Assignment 1F or the one you will be solving in Assignment 2F). Why is it defined this way?

```
Grid :: [(String, [Int])]
```

We shall begin by implementing a bunch of functions which can be used in the later implementations.

Task 1: Implement the function:
`map2 :: (a -> c, b -> d) -> (a, b) -> (c, d)`.

Task 2: Implement the function:
`mapIf :: (a -> a) -> (a -> Bool) -> [a] -> [a]`.

Task 3: Implement the function:
`maybeOr :: Maybe a -> Maybe a -> Maybe a`.

Task 4: Implement the function:
`firstJust :: [Maybe a] -> Maybe a`.

Task 5: Implement the function:
`lookupList :: Eq a => a -> [(a, [b])] -> [b]`.

These are helper functions that will take care of possible (partial) solutions of a Sudoku puzzle.

Part 2: Binding your knowledge with bind

The use of the bind function will be a required part of the solver in Assignment 2F and therefore we will cover it more thoroughly in this part of the lab. We will use the bind function on the `Maybe` data type and therefore we can write a specific implementation of it:
`maybeBind :: Maybe a -> (a -> Maybe b) -> Maybe b`
As seen in the type definition this function takes a `Maybe` value as its first argument, while the second argument is a function. This function takes a type `a` and after manipulation and possible conversion returns the type `b` inside a `Maybe` context. You remember from the last assignment that a `Maybe` type is often used when there is a possibility of error. We want to use `Maybe` in the same manner here where the error is an unsolvable Sudoku and we can phrase our usage of the bind function as follows: we want to apply a function that will fail on a value which might already have failed.

Let us code an example. In this example we want to replace multiple items in a list and we also want the result to be valid only if each replacement replaces an element.

Task 1: Implement the function:
`maybeBind :: Maybe a -> (a -> Maybe b) -> Maybe b`
Hint: If the first argument is `Nothing` the function should return `Nothing` otherwise it should return the result of the second parameter function applied on the value inside the first parameters `Maybe` context.

Task 2: Copy the function:

```
tryReplace :: Eq a => a -> a -> [a] -> Maybe [a]
tryReplace _ _ [] = Nothing
tryReplace y y' (x:xs)
  | x == y = Just (y':xs)
  | otherwise = fmap (x:) $ tryReplace y y' xs
```

How does it work? What does the `fmap` function do?

Now we want to successively replace different characters in a list multiple times. First, let us say we have a list:
`[1,2,3]`
We want to, in the following order, replace:

- 1 with 3
- 3 with 2
- 2 with 1

This should in the end return a `Just [1,2,3]`.
A second sequence:

- 1 with 3
- 1 with 2
- 2 with 1

should instead return `Nothing` since there is an invalid replacement.
To implement the first replacement sequence we could use case statements and get:

```
doIt :: Maybe [Int]
doIt = case tryReplace 1 3 [1,2,3] of
  Nothing -> Nothing
  Just list1 -> case tryReplace 3 2 list1 of
    Nothing -> Nothing
    Just list2 -> tryReplace 2 1 list2
```

However, this is neither clear nor generic for use with larger sequences. As you might have guessed, we can instead use the bind function (in infix form here) to get:

```
doIt :: Maybe [Int]
doIt = Just [1,2,3] `maybeBind` tryReplace 1 3  `maybeBind`
     tryReplace 3 2 `maybeBind` tryReplace 2 1
```

This implementation can be much easier expanded into recursion and folding, which is why the bind function is very useful in this case. Try writing the failing sequence and make up two sequences of your own to check if they are valid.

When you are familiar with the `maybeBind` function you can now, if you want, use the real bind function instead,

`(>>=) :: Monad m => m a -> (a -> m b) -> m b}.`

This is a generalised version which works for all data types which are instances of the typeclass Monad. As you know, `Maybe` is an instance of the Monad typeclass which means you can directly use it for the following problems of this assignment. It might be useful to be aware of the flipped version of the bind function as well:

`(=<<) :: Monad m => (a -> m b) -> m a -> m b`

Part 3: The `assign` function

The goal is to implement a function which assigns a value to a square and propagates elimination of that value to all the peers of the square. For each elimination of a value from a square, where we might run into a Sudoku which is unsolvable, we will consider three different special cases:

- After elimination there are no more possible values to insert in this square.
- The elimination does nothing since the value has already been eliminated from this square.
- There is only a single possible value left to put in this square.

These cases will be taken care of in a help function to the `assign` function, called `eliminate`. But first you can implement two other help functions.

Task 1: Implement the functions:
`setValue, eliminateValue :: Int -> String -> Grid -> Grid`.
The first function sets an `Int` value to a square (`String`) in the `Grid` and the `eliminateValue` does the same for elimination.
Hint: Use the functions `mapIf` and `map2` you have prevously implemented.

Task 2: Implement the function:
`eliminate :: Int -> String -> Grid -> Maybe Grid`
which considers the special cases mentioned above to try and eliminate a value from a square. If unsuccessful it should return the `Maybe` value for failure.
Hint: Use guards and the `where` syntax together with some of the previously implemented functions.

Task 3: Implement the function:
`assign :: Int -> String -> Grid -> Maybe Grid`.
Hint: Start by using a help function `assign'` for recursion of the peers in the elimination of a value together with the bind function.

Task 4: Now try to parse a Sudoku with the `parseGrid` function. What remains and how can you find a solution to this remaining `Grid`?

Part 4: Solving the puzzle

The remaining grid, after the parsing, will be solved simply through trying every possible combination and finding the first one that works. As a start we can write a help function
`solveSudoku' :: [String] -> Grid -> Maybe Grid`.
This function has a list of squares as first argument which it can go through recursively. For each recursive step it will in turn try to assign the values which are still possible to insert into the square (found in the `Grid` argument). If

assigning the value is successful continue to the next square, one step deeper in the recursion. Then if none of the possible values to put in the square are valid according to the `assign` function returns up one step and continues with the next possible value for the last square.
Might sound tricky but if bind and `map` are used together with the implemented functions `assign`, `firstJust`, and `lookupList` you can write one beautiful recursion, where the lazy evaluation of Haskell will do the rest.
Now use this function together with the `parseGrid` to finally write the function `solveSudoku :: Grid -> Maybe Grid` and test whether it works as expected.

Part 5: IO

In this part we will put the process of reading in a Sudoku, verifying it, supporting the user in solving it bit by bit, or completely solving it in one go, into one nice function call. The structure might resemble the one you remember from some early lecture:

```
interactionloop = do
        writesomething
        readsomething
        dosomeprocessing
        if finished then saygoodbye
                    else interactionloop
```

where `readsomething` will take in the user command and, depending on it, will dispatch the processing accordingly (possibly using recursion as well to keep the solving process continue).

More help will be available during the lab.