DM509 Programming Languages

Fall 2006, 2nd quarter

Project 1

Department of Mathematics and Computer Science
University of Southern Denmark

November 23, 2006

The purpose of this project is to implement in Haskell a function solving SuDoKu puzzles. The project is to be done in groups of two persons.

SuDoKu

A SuDoKu is a number puzzle where the objective is to complete a partially filled 9-by-9 table of numbers. Only the integers from 1 to 9 is to be used, and each number must appear exactly once in each row, in each column, and in each of the nine non-overlapping 3-by-3 sub-tables inside the main table. Normally, a Sudoku is supposed to have only one possible solution/completion. An example of a SuDoKu and its solution are shown below.

-	5	-	-	-	1	6	-	-
3	-	6	-	-	-	-	-	-
-	-	9	3	-	-	2	-	4
-	-	-	-	3	-	1	-	2
-	-	-	8	-	4	-	-	-
8	-	5	-	2	-	-	-	-
6	-	1	-	-	5	3	-	-
-	-	-	-	-	-	9	-	1
-	-	7	2	-	-	-	4	-

4	5	2	9	8	1	6	3	7
3	7	6	4	5	2	8	1	9
1	8	9	3	7	6	2	5	4
7	6	4	5	3	9	1	8	2
2	1	3	8	6	4	7	9	5
8	9	5	1	2	7	4	6	3
6	4	1	7	9	5	3	2	8
5	2	8	6	4	3	9	7	1
9	3	7	2	1	8	5	4	6

Much more info on SuDoKu can be found on the net (a good starting point in Danish is `http://www.daimi.au.dk/~helle/sudoku/`).

2D-Arrays

An obvious representation of a SuDoKu is by using a two-dimensional array. Haskell, however, does not have arrays as a built-in type (for good reason, standard arrays are highly imperative). However, arrays can be simulated rather efficiently using binary search trees.

Search trees store search keys from some ordered universe, possibly with additional information (often termed elements) associated with each search key, and allow searches (and also updates, by using any of number of possible rebalancing schemes, e.g. red-black trees) in time logarithmic in the number of stored keys. In binary search trees, each node has two subtrees (possibly empty), and all keys in the left [right] subtree are smaller [larger] than the key in the node.

The idea of simulating arrays using search trees is to use array indexes as search keys, and use values stored in the array as elements. For a two-dimensional array of height N and width M, the indexes are tuples (i, j), where i is an integer between 0 and $N-1$, and j is an integer between 0 and $M-1$. Note that tuples have an ordering, namely the lexicographical ordering, and hence are usable as search keys.

Arrays are static in size, but variable in contents. Hence the search trees are static in terms of tree shape and the indexes used, but are variable in terms of elements. To ensure efficiency, the trees should of course be as balanced as possible. Best possible balance is achieved for trees where the two subtrees of each node differ by at most one in size. Such trees are said to have perfect balance.

Task 1

Implement in Haskell methods for simulating 2D-arrays on a generic type `a` as array element type. The implementation should be based on binary search trees with perfect balance. Your implementation should contain the following type definitions:

```
type Array2D a = .....your tree type........
type Index2D = (Int,Int)
type Dimension2D = (Int,Int)
```

Your implementation should contain the following function definitions:

```
makeArray2D ::  Dimension2D -> a -> Array2D a
readArray2D ::  Array2D a -> Index2D -> a
updateArray2D ::  Array2D a -> Index2D -> a -> Array2D
```

The value returned by these functions should be the following, respectively:

> A new array with the dimensions given as first argument, and with all entries set to the element given as second argument .
>
> The array entry at the index given as second argument in the array given as first argument.
>
> An array which is identical to the array given as first argument, except that for the index given as second argument, the array entry is the value given as third argument.

Task 2

One idea for solving a SuDoKu is to try to expand all positions of the partially filled 9-by-9 table one by one in all possible ways allowed by the rules (i.e. that each number must appear exactly once in each row, in each column, and in each of the 3-by-3 sub-tables). This is the idea followed here.

Based on the following type for SuDoKus (i.e., partially filled 9-by-9 tables)

```
type Sudoku = Array2D Int
```

implement a function

```
expand ::  Sudoku -> Index2D -> [Sudoku]
```

which is the list of Sudokus obtained by expanding (i.e., inserting a number), in all possible ways allowed by the rules, the Sudoku given as first argument at the index given as second argument. If the given index is already filled, the list just contains the input Sudoku.

Then implement a function

```
solve ::  Sudoku -> [Sudoku]
```

which is the list of all possible solutions of the given Sudoku. In particular, `head (solve s)` is one (normally, the only) solution of a Sudoku `s`.

Input and Output

No facilities for input and output to and from files are required. You should assume that Sudokus to be solved are values in your program of type `[[Int]]`, where each row is an inner list (hence you will need a function converting this into something of type `Sudoku`). As an example, the Sudoku above should be represented as follows:

```
s :: [[Int]]
s =
  [[0, 5, 0,  0, 0, 1,  6, 0, 0],
   [3, 0, 6,  0, 0, 0,  0, 0, 0],
   [0, 0, 9,  3, 0, 0,  2, 0, 4],

   [0, 0, 0,  0, 3, 0,  1, 0, 2],
   [0, 0, 0,  8, 0, 4,  0, 0, 0],
   [8, 0, 5,  0, 2, 0,  0, 0, 0],

   [6, 0, 1,  0, 0, 5,  3, 0, 0],
   [0, 0, 0,  0, 0, 0,  9, 0, 1],
   [0, 0, 7,  2, 0, 0,  0, 4, 0]]
```

Also, you should simply let Hugs display the solution on the screen. Using some form of pretty-printing, based e.g. on Section 2.7.1 in Bird, would be nice, but is not required.

Formalities

A printed report of three to four pages should be handed in. The Haskell code (which must be sufficiently commented) and reasonable test data should be given as appendices. The main aim of the report should be to describe the structure of the final solution. Central Haskell code snippets should be included in the text of the report.

A copy of the Haskell code should be handed in using the `aflever` command on the Imada system: Move to the directory containing your code and issue the command `aflever DM509`. This will copy the contents of the directory to a place accessible by the lecturer. Repeated use of the command is possible (later uses overwrite the contents from earlier uses). In the directory, you must for identification purposes have an ASCII file named `names.txt` containing the names of the group members, with one name per line.

You must hand in the report and the code by

Thursday, December 7, 2006