CL exercise for Tutorial 7

Introduction

Objectives

In this tutorial, you will:

- become familiar with the DPLL algorithm;
- complete a Sudoku solver.

Tasks

Exercise 1 is mandatory. Exercise 2 is optional. All the rest are for your entertainment.

Submit

a file `sudoku.hs` with your answers.

Deadline

16:00 Tuesday 8 November

Reminder

Good Scholarly Practice

Please remember the good scholarly practice requirements of the University regarding work for credit.

You can find guidance at the School page

https://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct.

This also has links to the relevant University pages. Please do not publish solutions to these exercises on the internet or elsewhere, to avoid others copying your solutions.

Exercise 1 –mandatory–-marked–

Read Chapter 19 (*Checking Satisfiability*) from the textbook, then thoroughly examine the file `sudoku.hs`, which follows closely the descriptions of DPLL (the optimized version) and Sudoku from the book.

In the Haskell file, the function `sudoku` enumerates a list of eight constraints. Some of them are already defined. You need to complete the remaining ones with appropriate CNF expressions:

- `columnsComplete`: each column contains all digits;
- `squaresComplete`: each 3×3 square contains all digits;
- `columnsNoRepetition`: each column contains no repeated digit;
- `squaresNoRepetition`: each 3×3 square contains no repeated digit.

You may find it helpful to look at 14 of the lecture slides.

If you are using Windows, you may get ***`hPutChar: invalid character`*** *errors when you use the printing functions. There are two solutions:*

- *If you are using an up-to-date Haskell, start it with*
  `ghci +RTS --io-manager=native`
- *If that doesn't work, type*
  `chcp.com 65001`
  in the command shell before starting ghci.
- *If that doesn't work, ask on Piazza.*

Solution to Exercise 1

```
columnsComplete :: Form (Int,Int,Int)
columnsComplete = And [ Or [ P (i, j, n) | i <- [1..9]]
                        | j <- [1..9], n <- [1..9] ]

squaresComplete :: Form (Int,Int,Int)
squaresComplete = And [ Or [ P (3*p+q, 3*r+s, n)
                             | q <- [1..3], s <- [1..3]]
                        | p <- [0..2], r <- [0..2], n <- [1..9]]

columnsNoRepetition :: Form (Int,Int,Int)
columnsNoRepetition = And [ Or [ N (i, j, n), N (i', j, n) ]
                            | j <- [1..9], n <- [1..9],
                              i <- [1..9], i' <- [1..(i-1)] ]

squaresNoRepetition :: Form (Int,Int,Int)
squaresNoRepetition =
    And [ Or [ N (3*p+q, 3*r+s, n), N (3*p+q', 3*r+s', n) ]
          | p <- [0..2], r <- [0..2], n <- [1..9],
            q <- [1..3], s <- [1..3], q' <- [1..q], s' <- [1..3],
            q' < q || s' < s ]
```

Exercise 2 –optional—marked–

Your answers to this question should be typed in the comment block at the end of `sudoku.hs`

Some of the constraints in the definition of `sudoku` are redundant.

Which of the constraints give a minimal and complete description of the Sudoku game?

Can we improve the efficiency of the solver by removing some of the redundant constraints?

Compare the efficiency of different sets of constraints using Haskell.

Hint: To time the computation of solutions precisely, you could use the function `getCPUTime` from `System.CPUTime`. You can read about it here.

Solution to Exercise 2

The constraints `noneFilledTwice`, `rowsComplete`, `columnsComplete`, and `squaresComplete` give us a complete specification of Sudoku.

Adding the other constraints improves in fact the efficiency of the solver by restricting the search space explored by the DPLL algorithm.

You can read more about this in the textbook in chap 19.

Exercise 3 –optional—not marked–

Sudoku is usually defined for nine 3×3 squares using the digits 1–9.

Discuss, either before or during the tutorial, your colleagues what you would need to change in the code to make it work for sixteen 4×4 squares using the hexadecimal digits 1–9, A–F.

Hint: Think first of what should change for squares of size 2×2.

Exercise 4 –optional—not marked–

The unit clause optimisation is implemented by sorting the clauses by increasing length (the `prioritise` function in the code). Since we only want to ensure that pure literals come to the front of the list we could use a simpler sorting function to do just this.

We could also change the implementation of `<<` to ensure that clauses of length 1 (or of lengths 1 and 2) come first in its result.

Experiment with these optimisations.

Exercise 5 –optional—not marked–

The webpage http://magictour.free.fr/top95 gives a list of 95 hard sudoku problems.

- Each line is a string of length 81 listing the 81 squares.
- There are 9 characters for each Sudoku row. Each character is a digit or a period . representing a blank square.

Write a Haskell function to convert such a line into a Form representing the problem. Test your implementation on these problems.

Exercise 6 –optional—not marked–

Killer Sudoku is a variant of the Sudoku puzzle with groups of squares called *cages*. The digits in each cage should add to a given sum.

Modify the Haskell implementation of Sudoku by adding CNF expressions for the cage sum constraints in Killer Sudoku.