CL exercise for Tutorial 8

Introduction

Objectives

In this tutorial, you will:

- learn to apply the Tseytin transformation
- use the arrow rule to count satisfying valuations

Tasks

Exercises 1 and 2 are mandatory. Exercise 3 is optional.

Submit

a file called `cl-tutorial-8` with your answers (image or pdf) and a file `CLTutorial8KillerSudoku.hs` with your answers to exercise 3.

Deadline

12:00 Tuesday 14 November

Reminder

Good Scholarly Practice

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

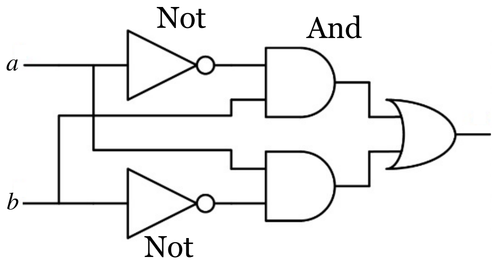
You can find guidance at the School page

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 Please do not publish solutions to these exercises on the internet or elsewhere, to avoid others copying your solutions.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–

Consider the following circuit:



Give an equivalent logical expression.

Apply the Tseytin transformation to give an equisatisfiable CNF expression.

Exercise 2 –mandatory—marked–

Read Chapter 23 (*Counting Satisfying Valuations*) of the textbook.

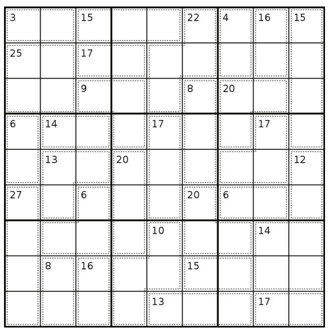
Use the arrow rule to count the number of satisfying assignments for the CNF expression

$$(E \vee F) \wedge (\neg A \vee B) \wedge C$$

Exercise 3 –optional—marked–

The answers to this exercise are quite short, but understanding the question may not be!

Killer Sudoku is a variant of the Sudoku puzzle. Like a standard Sudoku, each column, each row, and each 3×3 square must contain the numbers 1 to 9 exactly once. Killer Sudoku contains shapes marked by a dotted line (as in the image below). All the digits in a shape must add up to the total in the top corner of that shape.



Read the Haskell implementation of Killer Sudoku in the file `CLTutorial8KillerSudoku.hs`.

Complete the implementation by defining the following functions:
`scores :: Int -> Int -> [[Int]]`
that takes two natural numbers, `n` and `m`, and returns the list of all lists `ds` of digits from `[1..9]` such that (1) `length ds == n`, and (2) `sum ds == m`, and

```
mustSumTo ::  Int -> Shape -> Form (Int,Int,Int)
```

that takes an integer `k` and a shape `sh` and produces a `Form` that rejects all patterns of scores whose sum is not `k`.

Hint: For `mustSumTo`, you will want to use the function `deny` and the operator `>>*<` that are defined in the code file and discussed in the book.