TUTORIAL 7 · [COMPUTATION AND LOGIC]

OBJECTIVES

In this tutorial, you will:

- learn more about quantifiers and relations;
- become familiar with the DPLL algorithm;
- complete a Sudoku solver.

TASKS

Exercises 1–4 and 6 are mandatory. Exercise 7 is optional.
Exercises 5, 8–11 require no submission.

SUBMIT a file `cl-tutorial-7` (image or pdf) with your answers that do not require programming, and the files `all-you-need.hs` and `sudoku.hs` with your Haskell code.

DEADLINE Saturday, 7th of November, 4 PM UK time

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

> *http://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 Please do not publish solutions to these exercises on the internet or elsewhere, to avoid others copying your solutions.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 | BEFORE TUTORIAL SESSION

Watch the Week 7 videos and read Chapter 16 (*Relations and Quantifiers*) of the textbook.

Recall the universe of persons from Chapter 16. The file `cl-tutorial-7.hs` contains a Haskell representation of this universe.

$$\begin{tikzcd}
\text{Angela} \arrow[r] & \text{Ben} \arrow[dl, bend left] \arrow[dr] & \\
\text{Claudia} \arrow[u] \arrow[ur, bend left] \arrow[d] & & \text{Gavin} \arrow[d] \\
\text{Emilia} \arrow[rr] & & \text{Fangkai} \arrow[dl] \\
\text{Iain} \arrow[u] & \text{Diana} \arrow[l] \arrow[r, bend left] & \text{Hao} \arrow[l, bend left] \arrow[u]
\end{tikzcd}$$

EXERCISE 1 (CONT.)

Express the following Haskell representations of natural-language statements as propositions of one of the forms $\nvDash p$ or $\vDash p$ (for a suitable predicate p).

1. "Angela loves somebody"

```
some people (Angela `loves`)
```

2. "Everybody loves Ben"

```
every people (`loves` Ben)
```

3. "Somebody loves themself"

```
some people (\x -> x `loves` x)
```

SOLUTION TO EXERCISE 1

1. “Angela loves somebody”

   ⊭ ¬ `(Angela `loves`)`

2. “Everybody loves Ben”

   ⊨ `(`loves` Ben)`

3. “Somebody loves themself”

   ⊭ ¬ `( \x -> x `loves` x)`

EXERCISE 2

MANDATORY | BEFORE TUTORIAL SESSION

Translate the following Haskell code into English:

1.
```haskell
some people (\x -> every people (x `loves`))
```

2.
```haskell
every people (\x -> some people (`loves` x))
```

3.
```haskell
some people (\x -> every people (neg (`loves` x)))
```

4.
```haskell
some people (\x -> every people (not . (`loves` x)))
```

SOLUTION TO EXERCISE 2

1. There is somebody who loves everybody.

```
some people (\x -> every people (x `loves`))
```

2. For every person, there is some person who loves them.

```
every people (\x -> some people (`loves` x))
```

SOLUTION TO EXERCISE 2 (CONT.)

3. There is somebody who is not loved by anyone.

```
some people (\x -> every people (neg (`loves` x)))
```

4. There is somebody who is not loved by anyone.

```
some people (\x -> every people (not . (`loves` x)))
```

EXERCISE 3

MANDATORY | BEFORE TUTORIAL SESSION

For each of the statements below, first turn it into a formal mathematical statement (using the ∀ and ∃ symbols), then translate the statement into Haskell code and determine whether it is true.

1. Somebody loves someone who loves them.

$$\begin{tikzcd} x \arrow[r, bend left] & y \arrow[l, bend left] \end{tikzcd}$$

2. Everybody (x), loves some y, who loves some z, who loves them (x).

$$\begin{tikzcd} x \arrow[r] & y \arrow[r] & z \arrow[ll, bend left] \end{tikzcd}$$

SOLUTION TO EXERCISE 3

1. Somebody loves someone who loves them.

   $\exists x \in \texttt{people} . \exists y \in \texttt{people} . x \texttt{ `loves` } y \land y \texttt{ `loves` } x$

```
some people (\x -> some people (\y ->
    (x `loves` y) && (y `loves` x)))
```

2. Everybody (x), loves some y, who loves some z, who loves them (x).

   $\forall x \in \texttt{people} . \exists y \in \texttt{people} . \exists z \in \texttt{people}$
   $\quad . x \texttt{ `loves` } y \land y \texttt{ `loves` } z \land z \texttt{ `loves` } x$

```
every people (\x -> some people (\y -> some people (\z ->
    (x `loves` y) && (y `loves` z) && (z `loves` x))))
```

EXERCISE 4

Define a Haskell version of the ∃! quantifier (“there exists a unique”) and give an example of its use in our universe of persons.

In order to use equality in the definition, you will need to use the type

```
existsUnique :: Eq u => [u] -> Predicate u -> Bool
```

Hint: Use the fact that $\exists! x. P(x)$ is equivalent to:

$$\exists x.(P(x) \wedge \forall y.(P(y) \rightarrow x = y))$$

SOLUTION TO EXERCISE 4

```
existsUnique :: Eq u => [u] -> Predicate u -> Bool

existsUnique xs p =
    some xs (\x -> p x && every xs (\y ->
        if p y then x==y else True))
```

EXERCISE 5

OPTIONAL | DURING TUTORIAL SESSION

The first line of the song "Everybody Loves Somebody" is

"Everybody loves somebody *sometime*"

This statement is ambiguous. Discuss with your colleagues the different possible readings.

Pick a reading and express it in Haskell, after explaining how the universe of people would need to be enriched with additional information.

EXERCISE 6

MANDATORY | BEFORE TUTORIAL SESSION

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.

SOLUTION TO EXERCISE 6

```
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]]
```

SOLUTION TO EXERCISE 6 (CONT.)

```
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 7

OPTIONAL | BEFORE TUTORIAL SESSION

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 7

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 at pages 186–187.

EXERCISE 8

OPTIONAL | DURING TUTORIAL SESSION

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

Discuss with 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 9

NON-ASSESSED | NO SUBMISSION REQUIRED

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 10

NON-ASSESSED | NO SUBMISSION REQUIRED

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 11

NON-ASSESSED | NO SUBMISSION REQUIRED

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.