Logic
Informatics 1 – Introduction to Computation
Functional Programming Tutorial 6

Cooper, Heijltjes, Lehtinen, Melkonian, Sannella, Vlassi-Pandi, Wadler, Yallop

due 4pm Tuesday 1 November 2022
tutorials on Thursday 3 and Friday 4 November 2022

You will not receive credit for your coursework unless you attend the corresponding tutorial session. Please email kendal.reid@ed.ac.uk if you cannot join your assigned tutorial.

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 Please do not publish solutions to these exercises on the internet or elsewhere, to avoid others copying your solutions.

1 Implementing propositional logic in Haskell

1.1 Well-formed formulas

In this tutorial we will implement propositional logic in Haskell. In the file `Tutorial6.hs` you will find the following type and data declarations:

```
type Name = String

data Prop = Var Name
          | F
          | T
          | Not Prop
          | Prop :||: Prop
          | Prop :&&: Prop
          deriving (Eq)
```

The type `Prop` is a representation of a proposition, where variables names are of type `String`.

Propositional variables such as P and Q can be represented by `Var "P"` and `Var "Q"` of type `Prop`. Furthermore, we have the Boolean constants `F` and `T` for false (0) and true (1), corresponding to, but not the same as, `False, True :: Bool`; unary connective `Not` for negation ($\neg$), corresponding to, but not the same as, `not :: Bool -> Bool`); and infix binary connectives `:||:` and `:&&:` for disjunction ($\vee$) and conjunction ($\wedge$), corresponding to, but not the same as, `(||), (&&) :: Bool -> Bool -> Bool`.

Another type defined by `Tutorial6.hs` is:

```
type Valn = Name -> Bool
```

The type `Valn` is used as an *valuation* in which to evaluate a proposition, i.e. it is a function taking variable names to Boolean values.

Using these types, `Tutorial6.hs` defines the following functions:

- `showProp :: Prop -> String` converts a proposition into a readable string. For example:

  ```
  > showProp (Not (Var "P") :&: Var "Q")
  "((not P) && Q)"
  ```

- `eval :: Valn -> Prop -> Bool` evaluates the given proposition in the given valuation. For example, if we define the valuation

  ```
  vn :: Valn
  vn "P" = True
  vn "Q" = False
  ```

  then

  ```
  > eval vn (Var P :||: Var Q)
  True
  ```

- `names :: Prop -> Names]` where

  ```
  type Names = [Name]
  ```

  lists the names of the variables that occur in a proposition. Variable names occurring in the result are unique.

```
> names (Var "P" :||: (Var "P" :&&: Var "Q"))
["P","Q"]
```

- `table :: Prop -> IO ()` prints out a truth table.

```
> table ((Var "P" :&&: Not (Var "Q")) :&&: (Var "Q" :||: Var "P"))
P Q | ((P && (not Q)) && (Q || P))
- - | ----------------------------
1 1 |              0
0 1 |              0
1 0 |              1
0 0 |              0
```

- `valns :: Names -> [Valn]` generates a list of all the possible valuations for the given set of variable names. The result is a list of functions. Note that this list can't be displayed by Haskell.

- `satisfiable :: Prop -> Bool` checks whether a proposition is satisfiable — that is, whether there is some assignment of truth values to the variables in the proposition that will make the whole proposition true.

```
> satisfiable (Var "P" :&&: Not (Var "P"))
False
> satisfiable ((Var "P" :&&: Not (Var "Q")) :&&: (Var "Q" :||: Var "P"))
True
```

Exercise 1

Write the following formulas as values of type `Prop`; call them `p1`, `p2`, and `p3`.

(a) $(P \vee (\neg P))$

(b) $((P \vee Q) \wedge (P \wedge Q))$

(c) $((P \wedge (Q \vee R)) \wedge ((((\neg P) \vee (\neg Q)) \wedge ((\neg P) \vee (\neg R)))))$

Then use `table` to print their truth tables and `satisfiable` to check their satisfiability; include the results as comments in your submission.

1.2 Tautologies

Exercise 2

(a) A proposition is a tautology if it is always true, i.e. for every possible valuation. Using `names`, `valns` and `eval`, write a function `tautology :: Prop -> Bool` that checks whether the given proposition is a tautology. Test it on the examples from Exercise 1; include the results as comments in your submission.

(b) Create a QuickCheck test to verify that `tautology` is working correctly. Use the following as the basis for your test properties:

A proposition P is a tautology if and only if the proposition $\neg P$ is not satisfiable.

Note: be careful to distinguish the negation for booleans (`not`) from that for propositions (`Not`).

1.3 Connectives

We will extend the datatype and functions for propositions in `Tutorial6.hs` to handle the connectives $\rightarrow$ (implication) and $\leftrightarrow$ (bi-implication, or 'if and only if'). They will be implemented as the constructors `:->:` and `:<->:`. After you have implemented them, the truth tables for both should be as follows:

```
> table (Var "P" :->: Var "Q")
P Q | (P -> Q)
- - | ------
1 1 |    1
0 1 |    1
1 0 |    0
0 0 |    1
```

```
> table (Var "P" :<->: Var "Q")
P Q | (P <-> Q)
- - | -------
1 1 |     1
0 1 |     0
1 0 |     0
0 0 |     1
```

Exercise 3

(a) Find the declaration of the datatype `Prop` in `Tutorial6.hs` and extend it with the infix constructors `:->:` and `:<->:`.

(b) Find the functions `showProp`, `eval`, and `names`, and extend their definitions to cover the new constructors `:->:` and `:<->:`. There should be

```
-- begin
-- end
```

around any lines you add to the code. Test your definitions by printing out the truth tables for the two new connectives and checking that they match those given above; include the results as comments in your submission.

(c) Define the following propositions (call them `p4`, `p5`, and `p6`).

i. $(P \rightarrow Q) \leftrightarrow ((\neg P) \vee Q)$

ii. $(P \rightarrow Q) \wedge (Q \rightarrow P)$

iii. $((P \rightarrow Q) \wedge (Q \rightarrow R)) \wedge (\neg(P \rightarrow R))$

Then use `table` to print their truth tables and `satisfiable` to check their satisfiability, and `tautology` to check whether they are tautologies. Include the results as comments in your submission.

1.4 Arbitrary

Exercise 4

Below the 'exercises' section of `Tutorial6.hs`, in the section called 'for QuickCheck', you can find a declaration that starts with:

```
instance Arbitrary Prop where
```

This tells QuickCheck how to generate arbitrary propositions to conduct its tests. To make QuickCheck use the new constructors, uncomment the two lines in the middle of the definition:

```
-- , liftM2 (:->:) p2 p2
-- , liftM2 (:<->:) p2 p2
```

Now try your test property from Exercise 2 again.

2 Optional Material: Negation Normal Form

Following the Common Marking Scheme, a student with good mastery of the material is expected to get 3/4 points. This section is for demonstrating exceptional mastery of the material. It is optional and worth 1/4 points.

A proposition is in negation normal form if it consists of just the connectives $\vee$ and $\wedge$, propositional variables P and negated propositional variables $\neg P$, and the constants 1 and 0. Thus, negation is only applied to propositional variables, and nothing else.

To transform a proposition into negation normal form, you may use the following equivalences:

$$\begin{array}{rcl} \neg(P \wedge Q) & \Leftrightarrow & (\neg P) \vee (\neg Q) \\ \neg(P \vee Q) & \Leftrightarrow & (\neg P) \wedge (\neg Q) \\ (P \to Q) & \Leftrightarrow & (\neg P) \vee Q \\ (P \leftrightarrow Q) & \Leftrightarrow & (P \to Q) \wedge (Q \to P) \\ \neg(\neg P) & \Leftrightarrow & P \end{array}$$

Exercise 5

Write a function `isNNF :: Prop -> Bool` to test whether a `Prop` is in negation normal form.

Exercise 6

Write a function `impElim :: Prop -> Prop` that eliminates all implications in a proposition using two of the above equations, yielding an equivalent proposition containing no implications.

Exercise 7

Write a function `notElim :: Prop -> Prop` that converts a proposition containing no implications to an equivalent proposition in negation normal form. **Hint:** don't be alarmed if you need around a half dozen case distinctions to deal with negation.

Exercise 8

Write a function `toNNF :: Prop -> Prop` using the two previous functions that converts an arbitrary proposition to negation normal form. Use the test properties `prop_NNF1` and `prop_NNF2` to verify that your function is correct.

3 Challenge: Conjunctive Normal Forms

Challenges are meant to be difficult. You can receive full marks without attempting the challenge.

We consider how to convert a proposition into conjunctive normal form. A proposition is in *conjunctive normal form* if it is a conjunction of clauses, where each *clause* is a disjunction of literals, where each *literal* is either a propositional variable or its negation. The propositions `T` and `F` themselves are considered to be in conjunctive normal form, but otherwise they should not occur in propositions in conjunctive normal form.

Exercise 9

Write a function `isCNF :: Prop -> Bool` to test if a `Prop` is in conjunctive normal form.

Exercise 10

A common way of writing propositions in conjunctive normal form is as a list of lists, where the inner lists represent the clauses. Thus:

$$(((A \vee B) \wedge (C \vee (D \vee E)) \wedge G) \quad \textit{corresponds to} \quad \texttt{[[A,B],[C,D,E],[G]]}$$

A proposition in conjunctive normal form is true when *all* its clauses are true, and clause is true if *any* of its literals is true.

Write a function `fromLists :: [[Prop]] -> Prop` to translate a list of lists of literals (variables or negated variables) to a `Prop` in conjunctive normal form.

Hint: An obvious idea is to use `foldr (:||:) F` to combine a list of literals, and to use `foldr (:&&:) T` to combine a list of clauses. But, for example,

```
foldr (:||:) F [p,q,r] == (p :||: (q :||: (r :||: F)))
```

which is not what we want. A better idea is to use

```
foldr1 (:||:) [p,q,r]  ==  (p :||: (q :||: r))
```

Note that if the list of lists is empty it corresponds to true, whereas if it contains an empty list it corresponds to false, so in the remaining case we have a non-empty list of non-empty lists and it is safe to use `foldr1` instead of `foldr`.

Exercise 11

Next, we will convert a proposition in negation normal form to a list of lists. You can use the following distributive law (check it using your previous code):

$$P \vee (Q \wedge R) \quad \Leftrightarrow \quad (P \vee Q) \wedge (P \vee R)$$

Or, in a more general version:

$$(P_1 \wedge P_2 \wedge \ldots \wedge P_m) \vee (Q_1 \wedge Q_2 \wedge \ldots \wedge Q_n)$$

$$\Updownarrow$$

$$\begin{array}{c}(P_1 \vee Q_1) \wedge (P_1 \vee Q_2) \wedge (P_1 \vee Q_3) \wedge \ldots \wedge (P_1 \vee Q_n) \wedge \\ (P_2 \vee Q_1) \wedge (P_2 \vee Q_2) \wedge (P_2 \vee Q_3) \wedge \ldots \wedge (P_2 \vee Q_n) \wedge \\ \vdots \\ (P_m \vee Q_1) \wedge (P_m \vee Q_2) \wedge (P_m \vee Q_3) \wedge \ldots \wedge (P_m \vee Q_n)\end{array}$$

Write a function `toLists :: Prop -> [[Prop]]` takes a proposition in negation normal form and converts it to a list of lists of literals (propositional variables and their negations), representing the proposition in conjunctive normal form.

Exercise 12

Write a function `toCNF :: Prop -> Prop` using the function to convert to negation normal form and the two previous functions that converts an arbitrary proposition to negation normal form. Use the test properties `prop_CNF1` and `prop_CNF2` to verify that your function is correct.

Hint: Transforming to conjunctive normal form is computationally expensive, especially for propositions with many bi-implications ($\leftrightarrow$). Test your code on small examples first before trying the test properties `prop_CNF1` and `prop_CNF2` with QuickCheck.