Logic

Informatics 1 – Introduction to Computation
Functional Programming Tutorial 6

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

Week 7
due 4pm Wednesday 04 November 2020
tutorials on Friday 06 November 2020

Attendance at tutorials is obligatory; please send email to lambrose@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 Please do not publish solutions to these exercises on the internet or elsewhere, to avoid others copying your solutions.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 Warmup for Algebraic Data Types

As a warmup write some functions to act on input of the user-defined type Fruit. In the file `Tutorial6.hs` you will find the following data declaration:

```
data Fruit = Apple String Bool
           | Orange String Int
```

An expression of type `Fruit` is either an `Apple String Bool` or an `Orange String Int`. We use a `String` to indicate the variety of the apple or orange, a `Bool` to describe whether an apple has a worm and an `Int` to count the number of segments in an orange. For example:

```
Apple "Granny Smith" False  -- a Granny Smith apple with no worm
Apple "Braeburn" True       -- a Braeburn apple with a worm
Orange "Sanguinello" 10     -- a Sanguinello orange with 10 segments
```

Exercise 1

Write a function `isBloodOrange :: Fruit -> Bool` which returns `True` for blood oranges and `False` for apples and other oranges. Blood orange varieties are: Tarocco, Moro and Sanguinello. For example:

```
isBloodOrange(Orange "Moro" 12) == True
isBloodOrange(Apple "Granny Smith" True) == False
```

Exercise 2

Write a function `bloodOrangeSegments :: [Fruit] -> Int` which returns the total number of blood orange segments in a list of fruit.

Exercise 3

Write a function `worms :: [Fruit] -> Int` which returns the number of apples that contain worms.

1.2 Well-formed formulas

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

```
data Wff a = V a
         | T
         | F
         | Not (Wff a)
         | Wff a :|: Wff a
         | Wff a :&: Wff a

data Atom = A|B|C|D|P|Q|R|S|W|X|Y|Z
```

The type `Wff a` is a representation of a well-formed formula, that we will note as **wff a** throughout the tutorial. Propositional variables such as P and Q can be represented as `V P` and `V Q`. Furthermore, we have the Boolean constants `T` and `F` for ‘true’ and ‘false’, the unary connective `Not` for negation (not to be confused with the function `not :: Bool -> Bool`), and (infix) binary connectives `:|:` and `:&:` for disjunction ($\vee$) and conjunction ($\wedge$). Another type defined by `Tutorial6.hs` is:

```
type Env a = [(a, Bool)]
```

The type `Env` is used as an 'environment' in which to evaluate a wff, i.e. it is a list of truth assignments for (the atoms of) propositional variables. Using these types, `Tutorial6.hs` defines the following functions:

- `eval :: Eq a => Env a -> Wff a -> Bool` evaluates the given wff in the given environment (assignment of truth values). For example:

  ```
  *Main> eval [(P, True), (Q, False)] (V P :|: V Q)
  True
  ```

- `atoms :: Eq a => Wff a -> [a]` lists the atoms that occur in a wff. Atoms occurring in the result are unique.

  ```
  *Main> atoms (V P :|: (V P :&: V Q))
      [P,Q]
  ```

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

  ```
  *Main> satisfiable (V P :&: Not (V P))
  False
  *Main> satisfiable ((V P :&: Not (V Q)) :&: (V Q :|: V P))
  True
  ```

- `envs :: [a] -> [Env a]` generates a list of all the possible truth assignments for the given set of atoms. Example:

  ```
  *Main> envs [P, Q]
  [[(P,False),(Q,False)],
   [(P,False),(Q,True)],
   [(P,True),(Q,False)],
   [(P,True),(Q,True)]]
  ```

- `showWFF :: Show a => Wff a -> String` converts a wff into a readable string approximating the mathematical notation. For example:

  ```
  *Main> showWff (Not (V P) :&: V Q)
  "~P & Q"
  ```

- `table :: (Eq a, Show a) => Wff a -> IO ()` prints out a truth table.

  ```
  *Main> table ((V P :&: Not (V Q)) :&: (V Q :|: V P))
  P Q | ((P&(~Q))&(Q|P))
  - - | ----------------
  F F |        F
  F T |        F
  T F |        T
  T T |        F
  ```

- `fullTable :: (Eq a, Show a) Wff a -> IO ()` prints out a truth table that includes the evaluation of the subformulas of the given wff. (**Note:** `fullTable` uses the function `subformulas` that you will define in Exercise 8, so it doesn't work just yet.)

```
*Main> fullTable ((V P :&: Not (V Q)) :&: (V Q :|: V P))
```

Exercise 4

Write the following formulas as `Wffs` (call them `wff1` and `wff2`). Then use `satisfiable` to check their satisfiability and `table` to print their truth tables.

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

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

1.3 Tautologies

Exercise 5

(a) A wff is a tautology if it is always true, i.e. in every possible environment. Using `atoms`, `envs` and `eval`, write a function `tautology :: Wff -> Bool` which checks whether the given wff is a tautology. Test it on the examples from Exercise 4 and on their negations.

(b) Create two QuickCheck tests to verify that `tautology` is working correctly. Use the following facts as the basis for your test properties:
For any Wff P,

i. either P is a tautology, or $\neg P$ is satisfiable,

ii. either P is not satisfiable, or $\neg P$ is not a tautology.

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

1.4 Connectives

We will extend the datatype and functions for wffs 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:

```
*Main> table (V P :->: V Q)          *Main> table (V P :<->: V Q)
P Q | (P->Q)                                   P Q | (P<->Q)
- - | ------                                   - - | -------
F F |   T                                      F F |    T
F T |   T                                      F T |    F
T F |   F                                      T F |    F
T T |   T                                      T T |    T
```

Exercise 6

(a) Find the declaration of the datatype `Wff` in `Tutorial6.hs` and extend it with the infix constructors `:->:` and `:<->:`. Also, uncomment the lines `infixr 1 :->:` and `infixr 0 :<->:`.

(b) Find the printer (`showWff`), evaluator `substitute` and `evaluate`, and name-extractor (`atoms`) functions and extend their definitions to cover the new constructors `:->:` and `:<->:`. Test your definitions by printing out the truth tables above.

(c) Define the following wffs as `Wffs` (call them `wff3` and `wff4`). Check their satisfiability and print their truth tables.

i. $((P \rightarrow Q) \wedge (P \wedge (\neg Q)))$

ii. $((P \leftrightarrow Q) \wedge ((P \wedge (\neg Q)) \vee ((\neg P) \wedge Q)))$

1.5 Equivalence

Two wffs are *equivalent* if they always have the same truth values, regardless of the values of their propositional variables. In other words, wffs are equivalent if in any given environment they are either both true or both false.

Exercise 7

Write a function `equivalent :: Wff a -> Wff a -> Bool` that returns `True` just when the two wffs are equivalent in this sense. For example:

```
*Main> equivalent (V P :&: V Q) (Not (Not (V P) :|: Not (V Q)))
True
*Main> equivalent (V P) (V Q)
False
*Main> equivalent (V R :|: Not (V R)) (V Q :|: Not (V Q))
True
```

You can use `atoms` and `envs` to generate all relevant environments, and use `eval` to evaluate the two `Wff`s.

1.6 Subformulas

The *subformulas* of a wff are defined as follows:

- A propositional letter P or a constant **t** or **f** has itself as its only subformula.
- A wff of the form $\neg P$ has as subformulas itself and all the subformulas of P.
- A wff of the form $P \wedge Q$, $P \vee Q$, $P \to Q$, or $P \leftrightarrow Q$ has as subformulas itself and all the subformulas of P and Q.

The function `fullTable :: (Eq a, Show a) => Wff a -> IO ()`, already defined in `Tutorial6.hs`, prints out a truth table for a wff, with a column for each of its non-trivial subformulas.

Exercise 8

(a) Add a definition for the function `subformulas :: Eq a => Wff a -> [Wff a]` that returns all of the (unique) subformulas of a wff. **Hint:** to ensure uniqueness, use the list function `nub`. For example:

```
*Main> map showWff (subformulas wff1)
["((P | Q)&(P & Q))","(P | Q)","P","Q","(P & Q)"]
```

(We need to use `map showWff` here in order to convert each wff into a string; otherwise we could not easily view the results.)

(b) Test out `subformulas` and `fullTable` on each of the `Wff`s you defined earlier [`wff1`..`wff4`].

Exercise 9

(a) Write the following formula $((P \vee Q) \wedge ((\neg P) \wedge (\neg Q)))$ as `wff5`. Then use `satisfiable` to check its satisfiability and `table` to print its truth table.

(b) Test your `tautology` function on the `wwf` and on its negation. If something goes wrong refer to your implementation of Exercise 5.

(c) Write the following formula $((P \to Q) \wedge (P \leftrightarrow Q))$ as `wff6`. Again use `satisfiable` to check its satisfiability and `table` to print its truth table. If something goes wrong refer to your implementation of Exercise 6.

(d) i. Write another version of Exercise's 7 `equivalent`, this time by combining the two arguments into a larger wff and using `tautology` or `satisfiable` to evaluate it.

ii. Write a QuickCheck test property to verify that the two versions of `equivalent` are equivalent.

(e) Test out your `subformulas` and `fullTable` on each of the [`wff5`, `wff6`] you defined earlier. If something goes wrong refer to your implementation of Exercise 8.

2 Optional material

Please note that optional exercices **do** contribute to the final mark. If you don't do the optional work and get the rest mostly right you will get a mark of 3/4. To get a mark of 4/4, you must get almost all of the tutorial right, including the optional questions.

2.1 Normal Forms

In this part of the tutorial we will put wffs into several different normal forms. First, we will deal with negation normal form. As a reminder, a wff is in negation normal form if it consists of just the connectives $\vee$ and $\wedge$, unnegated propositional variables P and negated propositional variables $\neg P$, and the constants $\mathbf{t}$ and $\mathbf{f}$. Thus, negation is only applied to propositional variables, and nothing else.

To transform a wff into negation normal form, you might want to 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 10

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

Exercise 11

Write a function `impElim :: Wff a -> WWF a` that converts implications to normal forms.

Write a function `toNNF :: Wffa -> Wff a` using `impElim` that puts an arbitrary `Wff` into negation normal form. Use the test properties `prop_NNF1` and `prop_NNF2` to verify that your function is correct. **Hint:** don't be alarmed if you need many case distinctions.

2.1.1 Conjunctive Normal Forms

Next, we will turn a wff into conjunctive normal form. This means the wff is a conjunction of clauses, and a clause is a disjunction of (possibly negated) propositional variables, called *atoms*.

You will need to pay special attention to the constants $\mathbf{t}$ and $\mathbf{f}$. The `Wffs` `T` and `F` themselves are considered to be in conjunctive normal form, but otherwise they should not occur in wffs in normal form. They can be eliminated using the following equivalences:

$$
\begin{array}{rcccl}
(P \wedge \mathbf{t}) & \Leftrightarrow & (\mathbf{t} \wedge P) & \Leftrightarrow & P \\
(P \wedge \mathbf{f}) & \Leftrightarrow & (\mathbf{f} \wedge P) & \Leftrightarrow & \mathbf{f} \\
(P \vee \mathbf{t}) & \Leftrightarrow & (\mathbf{t} \vee P) & \Leftrightarrow & \mathbf{t} \\
(P \vee \mathbf{f}) & \Leftrightarrow & (\mathbf{f} \vee P) & \Leftrightarrow & P
\end{array}
$$

Exercise 12

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

Exercise 13

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

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

Think of how the constants **t** and **f** can be represented as lists of lists.

Hint: a wff in conjunctive normal form is true when *all* its clauses are true. A clause is true if *any* of its atoms is true.

Write a function `listsToCNF` to translate a list of lists of `Wff`s (which you may assume to be variables or negated variables) to a `Wff` in conjunctive normal form.

Exercise 14

Write a function `listsFromCNF` to write a wff in conjunctive normal form as a list of lists.

Exercise 15

Finally, we will convert an arbitrary `Wff` to a list of lists. You can use the following distributive law (check it first using your previous code):

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

Or, in a more generalized 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 `toCNFList` that turns a `Wff` into a list of lists of propositional variables and their negations, representing the wff in conjunctive normal form. The output of `toCNFList` may contain empty clauses as long as `toCNF` produces a wff without `T` nor `F` as strict subformulas. Check that the result of `toCNF` is actually in conjuctive normal form (`prop_CNF1`) and equivalent to its input (`prop_CNF2`).

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