Logic
Informatics 1 – Functional Programming: Tutorial 6

Heijltjes, Wadler

Due: The tutorial of week 8 (13/14 Nov.)
Reading assignment: Chapters 12–14 (pp. 210–279)

Please attempt the entire worksheet in advance of the tutorial, and bring with you all work, including (if a computer is involved) printouts of code and test results. Tutorials cannot function properly unless you do the work in advance.

You may work with others, but you must understand the work; you can't phone a friend during the exam.

Assessment is formative, meaning that marks from coursework do not contribute to the final mark. But coursework is not optional. If you do not do the coursework you are unlikely to pass the exams.

Attendance at tutorials is obligatory; please let your tutor know if you cannot attend.

Logic

In this tutorial we will implement propositional logic in Haskell. Download the file `tutorial6.hs` from the website. It defines the following types:

```
type Name = String
data Prop = Var Name
          | F
          | T
          | Not Prop
          | Prop :|: Prop
          | Prop :&: Prop
```

The type `Prop` is a representation of propositional formulas. Propositional variables such as P and Q can be represented as `Var "P"` and `Var "Q"`. Furthermore, we have the Boolean constants `T` and `F` for 'true' and 'false', unary predicate `Not` for negation (not to be confused with the function `not :: Bool -> Bool`), and (infix) binary predicates `:|:` and `:&:` for disjunction ($\vee$) and conjunction (&). Another type defined by `tutorial.hs` is:

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

The type `Env` is used as an 'environment' in which to evaluate a proposition: it is a list of truth assignments for (the names of) propositional variables. Using these types, `tutorial6.hs` defines the following functions:

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

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

- `eval :: Env -> Prop -> Bool` evaluates the given proposition in the given environment (assignment of truth values). For example:

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

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

```
*Main> showProp (Not (Var "P") :&: Var "Q")
"((~P)&Q)"
```

- `names :: Prop -> Names` returns all the variable names used in a proposition. Example:

```
*Main> names (Not (Var "P") :&: Var "Q")
["P", "Q"]
```

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

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

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

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

- `fullTable :: Prop -> IO ()` prints out a truth table that includes the evaluation of the subformulas of the given proposition. (**Note:** `fullTable` uses the function `subformulas` that you will define in Exercise (5), so it doesn't work just yet.)

```
*Main> fullTable ((Var "P" :&: Not (Var "Q")) :&: (Var "Q" :|: Var "P"))
P Q | ((P&(~Q))&(Q|P)) (P&(~Q)) (~Q) (Q|P)
- - | ---------------- -------- ---- -----
F F |        F             F      T     F
F T |        F             F      F     T
T F |        T             T      T     T
T T |        F             F      F     T
```

Exercises

1. Write the following formulas as `Props`. Then use `satisfiable` to check their satisfiability and `table` to print their truth tables.

   (a) $((P \vee Q) \mathbin{\&} (P \mathbin{\&} Q))$

   (b) $((P \vee Q) \mathbin{\&} ((\neg P) \mathbin{\&} (\neg Q)))$

   (c) $((P \mathbin{\&} (Q \vee R)) \mathbin{\&} (((\neg P) \vee (\neg Q)) \mathbin{\&} ((\neg P) \vee (\neg R))))$

2. (a) Using `names`, `envs` and `eval`, write a function `tautology :: Prop -> Bool` which checks whether the given proposition is a tautology. Test it on the examples from Exercise (1) 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 property 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 `Bool`s (`not`) from that for `Props` (`Not`).

3. We will extend the datatype and functions for propositions in `Prop.hs` to handle the connectives $\rightarrow$ and $\leftrightarrow$.

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

   (b) Find the printer (`showProp`), evaluator (`eval`), and name-extractor (`names`) functions and extend their definitions to cover the new constructors `:->:` and `:<->:`.

   (c) Check the satisfiability of the following formulas and print their truth tables.

   i. $((P \rightarrow Q) \mathbin{\&} (P \leftrightarrow Q))$

   ii. $((P \rightarrow Q) \mathbin{\&} (P \mathbin{\&} (\neg Q)))$

   iii. $((P \leftrightarrow Q) \mathbin{\&} ((P \mathbin{\&} (\neg Q)) \vee ((\neg P) \mathbin{\&} Q)))$

   (d) Find the declaration that starts with:

   ```
   instance Arbitrary Prop where
   ```

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

   ```
   -- , liftM2 (:->:) subform subform
   -- , liftM2 (:<->:) subform' subform'
   ```

   Now try your test properties from Exercise (2b) again. (**Note:** if you use `verboseCheck`, you will see that QuickCheck uses your `showProp` function to display the propositions.)

4. Two formulas are *equivalent* if they always have the same truth values, regardless of the values of their propositional variables.

   (a) Using `names`, `envs`, and `eval`, write a function `equivalent :: Prop -> Prop -> Bool` that returns `True` just when the two propositions are equivalent in this sense. For example:

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

(b) Write another version of equivalent, this time by combining the two arguments into a larger proposition and using `tautology` or `satisfiable` to evaluate it.

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

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

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

The function `fullTable :: Prop -> IO ()`, already defined in `Table.hs`, prints out a truth table for a formula, with a column for each of its non-trivial subformulas.

Exercises

5. Add a definition for the function `subformulas :: Prop -> [Prop]` that returns all of the subformulas of a formula. For example:

```
*Main> map showProp (subformulas p2)
["((P|Q)&((~P)&(~Q)))","(P|Q)","P","Q","((~P)&(~Q))","(~P)","(~Q)"]
```

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

Test out `subformulas` and `fullTable` on each of the formulas in questions 1 and 3.

Normal Forms (Optional)

In this part of the tutorial we will put propositional formula into several different normal forms. First, we will deal with negation normal form. As a reminder, a formula is in negation normal form if it consists of just the connectives $\vee$ and &, positive propositional letters P and negated ones $\neg P$, and the constants **t** and **f**. Thus, negation is only applied to propositional letters, and nothing else.

To transform a formula into negation normal form, you might want to use the following equivalences:

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

Next, we will turn a formula into conjunctive normal form. This means the formula is a conjunction of clauses, and a clause is a disjunction of (possibly negated) propositional variables. You can use the following distributive law:

$$P \vee (Q \,\&\, R) \quad \Leftrightarrow \quad (P \vee Q) \,\&\, (P \vee R)$$

Or, in a more generalized version:

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

$$\Updownarrow$$

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

A common way of writing formulas in conjunctive normal form is as a list of lists. Thus:

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

In `tutorial6.hs` you will find functions `listsToCNF` and `listsFromCNF` for converting between the two formats, as well as a property `isCNF` that tells you whether a `Prop` is already in conjunctive normal form.

Exercises

6. Complete the function `toNNF` that transforms an arbitrary `Prop` into negation normal form. Use the test properties `prop_NNF1` and `prop_NNF2` to verify that your function is correct.
7. Complete one of the functions `toCNF` and `toCNFL` so that it transforms a formula to conjunctive normal form or a list of lists, respectively. Use the test properties `prop_CNF` and `prop_CNFL` to check your answer.

Note: transforming to conjunctive normal form is computationally expensive; especially formulas with many bi-implications ($\leftrightarrow$). Make sure you use `verboseCheck` when testing, so you can see how long QuickCheck takes to verify a single case.

When trying these exercises, you will need to pay special attention to the constants **t** and **f**. Remember that you can eliminate them using the following equivalences:

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

It is a useful exercise to see how these equivalences apply to a list of lists.