

Logic

Informatics 1 – Introduction to Computation

Functional Programming Tutorial 5

Week 6 (22-26 Oct.)

Each tutorial involves a mixture of individual homework as a preparation step, followed by work and discussion done by groups of five to six students during the tutorial meeting. You are being encouraged to work together and help each other throughout the whole tutorial. A number of tutors will also be available to give help or advice. If at any point your group needs assistance please raise your hand to draw our attention.

Please go through the entire worksheet in advance of the tutorial; answer the questions when needed or take some personal notes. Make sure that you complete the *Answer Sheet* of the second section, which in turn corresponds to various tasks throughout the tutorial. Don't forget to bring your answers along with any personal notes — personal notes may include printouts of code and test results.

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 send email to Rob.Armitage@ed.ac.uk if you cannot join your assigned tutorial.

1 Homework: Implementing propositional logic in Haskell

1.1 Well-formed formulas

In this tutorial we will mainly implement propositional logic in Haskell. In the file `tutorial15.hs` you will find the following type and data declarations:

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

type Names = [Name]
```

The type `Wff` is a representation of a well-formed formula, that we will note as **wff** throughout the tutorial. 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’, 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 `tutorial15.hs` is:

```
type Env = [(Name, 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 names of) propositional variables. Using these types, `tutorial15.hs` defines the following functions:

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

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

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

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

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

- `names :: Wff -> Names` returns all the variable names used in a wff. 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 :: Wff -> 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 :: Wff -> 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 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
```

Exercise 1

- 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.
 - $((P \vee Q) \wedge (P \wedge Q))$
 - $((P \wedge (Q \vee R)) \wedge (((\neg P) \vee (\neg Q)) \wedge ((\neg P) \vee (\neg R))))$
- Fill out the *Answer Sheet* in Section 2 of the tutorial with the body of the aforementioned functions `[1ai, 1aii]`.

1.2 Tautologies

Exercise 2

- A wff is a tautology if it is always true, i.e. in every possible environment. Using `names`, `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 1](#) and on their negations.
- 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 ,
 - either P is a tautology, or $\neg P$ is satisfiable,
 - either P is not satisfiable, or $\neg P$ is not a tautology.**Note:** be careful to distinguish the negation for Booleans (`not`) from that for Wffs (`Not`).
- Fill out the *Answer Sheet* in Section 2 of the tutorial with the body of the aforementioned functions `[2a, 2b]`.

1.3 Connectives

We will extend the datatype and functions for wffs in `tutorial5.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:

<pre>*Main> table (Var "P" :->: Var "Q") P Q (P->Q) - - ----- F F T F T T T F F T T T</pre>	<pre>*Main> table (Var "P" :<->: Var "Q") P Q (P<->Q) - - ----- F F T F T F T F F T T T</pre>
--	--

Exercise 3

- Find the declaration of the datatype `Wff` in `tutorial5.hs` and extend it with the infix constructors `:->` and `:<->`.
- Find the printer (`showWff`), evaluator (`eval`), and name-extractor (`names`) functions and extend their definitions to cover the new constructors `:->` and `:<->`. Test your definitions by printing out the truth tables above.
- Define the following wffs as `Wffs` (call them `wff3` and `wff4`). Check their satisfiability and print their truth tables.
 - $((P \rightarrow Q) \wedge (P \wedge (\neg Q)))$
 - $((P \leftrightarrow Q) \wedge ((P \wedge (\neg Q)) \vee ((\neg P) \wedge Q)))$
- Below the ‘exercises’ section of `tutorial5.hs`, in the section called ‘for QuickCheck’, you can find a declaration that starts with:

```
instance Arbitrary Wff where
```

This tells QuickCheck how to generate arbitrary `Wffs` 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 2](#) again.

- Fill out the *Answer Sheet* in Section 2 of the tutorial with the body of the aforementioned functions [3c].

1.4 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 4

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

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

You can use `names` and `envs` to generate all relevant environments, and use `eval` to evaluate the two `Wffs`.

- (b) Fill out the *Answer Sheet* in Section 2 of the tutorial with the body of the aforementioned function [4a].

1.5 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 $P \wedge Q$, $P \vee Q$, $P \rightarrow Q$, or $P \leftrightarrow Q$ has as subformulas itself and all the subformulas of P and Q .

The function `fullTable :: Wff -> IO ()`, already defined in `tutorial5.hs`, prints out a truth table for a `wff`, with a column for each of its non-trivial subformulas.

Exercise 5

- (a) Add a definition for the function `subformulas :: Wff -> [Wff]` that returns all of the subformulas of a `wff`. 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.)

Test out `subformulas` and `fullTable` on each of the `Wffs` you defined earlier [`wff1..wff4`].

- (b) Fill out the *Answer Sheet* in Section 2 of the tutorial with the body of the aforementioned function [5a].

2 Answer Sheet

Please fill in your name as another member of your group will check your answers.

Name:

Exercise	
1ai: wff1	
1aii: wff2	
2a: tautology	
2b: prop_taut1	
2b: prop_taut2	
3ci: wff3	
3cii: wff4	
4a: equivalent	
5a: subformulas	

3 Tutorial Activities

3.1 Warmup for Algebraic Data Types

As a warmup write some functions to act on input of the user-defined type `Fruit`. In the file `tutorial5.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
```

Work with the person sitting next to you to fill out the table below with the body of the following functions that correspond to Exercises [6..8] below.

6: isBloodOrange	
7: bloodOrangeSegments	
8: worms	

Exercise 6

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 7

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

Exercise 8

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

3.2 Test your Logic

The following set of exercises is similar to the one that you have implemented as *Homework* in Section 1. Please switch pairs to fill out the table below with the body of the following functions that correspond to Exercise [9..10] below; when needed refer to your answers for Exercises [1ai..5a].

9a: wff5	
9c: wff6	
9di: equivalent'	
9dii: prop_equivalent	

Exercise 9

- 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.
- Test your `tautology` function on the `wff` and on its negations. If something goes wrong refer to your implementation of [Exercise 2](#).
- Write the following formula $((P \rightarrow Q) \wedge (P \leftrightarrow Q))$ 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 3](#).
- Write another version of [Exercise's 4](#) `equivalent`, this time by combining the two arguments into a larger wff and using `tautology` or `satisfiable` to evaluate it.
 - Write a QuickCheck test property to verify that the two versions of `equivalent` are equivalent.
- 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 5](#).

4 Optional material

Please note that you may **NOT** have time to complete this last part during the tutorial; you can do it at home afterwards if you want.

4.1 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 **t** and **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:

$$\begin{aligned}\neg(P \wedge Q) &\Leftrightarrow (\neg P) \vee (\neg Q) \\ \neg(P \vee Q) &\Leftrightarrow (\neg P) \wedge (\neg Q) \\ (P \rightarrow Q) &\Leftrightarrow (\neg P) \vee Q \\ (P \leftrightarrow Q) &\Leftrightarrow (P \rightarrow Q) \wedge (Q \rightarrow P) \\ \neg(\neg P) &\Leftrightarrow P\end{aligned}$$

Exercise 10

Write a function `isNNF` to test whether a `Wff` is in negation normal form.

Exercise 11

Write a function `toNNF` 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.

4.1.1 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 **t** and **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:

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

Exercise 12

Write a function `isCNF` to test if a `Wff` is in conjunctive normal form.

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

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

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.

Exercise 13

Write a function `listsToCNF` to translate a list of lists of `Wffs` (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):

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

Or, in a more generalized version:

$$\begin{aligned} & (P_1 \wedge P_2 \wedge \dots \wedge P_m) \vee (Q_1 \wedge Q_2 \wedge \dots \wedge Q_n) \\ & \Updownarrow \\ & (P_1 \vee Q_1) \wedge (P_1 \vee Q_2) \wedge (P_1 \vee Q_3) \wedge \dots \wedge (P_1 \vee Q_n) \wedge \\ & (P_2 \vee Q_1) \wedge (P_2 \vee Q_2) \wedge (P_2 \vee Q_3) \wedge \dots \wedge (P_2 \vee Q_n) \wedge \\ & \vdots \\ & (P_m \vee Q_1) \wedge (P_m \vee Q_2) \wedge (P_m \vee Q_3) \wedge \dots \wedge (P_m \vee Q_n) \end{aligned}$$

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.

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 property `wff_CNF` with `QuickCheck`.