Functional Programming 1 — 1DL330

Assignment 4

Lab: Thursday, 28 September

Submission Deadline: 18:00, Tuesday, 3 October, 2017

This is an **individual** assignment. You must construct your own solution. Keep any discussions with other students at the level of abstract solution methods; do *not* share your code. If you think there is a risk that such a discussion leads to very similar solutions, then report this to the instructor in advance, e.g., by clearly stating any collaboration in your submission. If you have any questions about this assignment, please contact our lab assistant, albert.yang@it.uu.se.

Instructions

- Start by reading the General Lab Instructions (on the Student Portal).
- Name the file containing your solutions `lab4.sml`. The file that you submit must contain valid SML code, so the answers to some of the questions need to be placed in comments: *(* ... *)*.
- Remember to follow our Coding Convention (also on the Student Portal), and provide specifications for all functions that you write.
- Make sure that your solution passes the tests in `lab4_test.sml` before you submit. A submission that does not pass these tests will get grade K.

1 A Signature for Valuations

A valuation in propositional logic maps named variables (i.e., strings) to Boolean values. Define a signature VALUATION for valuations. This signature shall provide a type T of valuations, and various operations on them. Among the operations provided there shall be a way to

1. obtain an empty valuation (empty: T),
2. add a variable with a given Boolean value (set : T -> string -> bool -> T),
3. find the Boolean value associated with a variable (value_of : T-> string -> bool),
4. compute a list of all variables in a valuation (variables : T -> string list),
5. and print the contents of a valuation (print : T -> unit).

2 A Structure for Valuations

1. Define a matching structure Valuation :> VALUATION that implements (at least) the operations from Exercise 1. (You may use any suitable data structure that has been presented in class, or invent one yourself.)

   To print the contents of a valuation, you can use the following code:

```
fun print valuation =
  (
    List.app
      (fn name => TextIO.print (name ^ " = " ^
         Bool.toString (value_of valuation name) ^ "\n"))
      (variables valuation);
    TextIO.print "\n"
  )
```

   (See here for an explanation of List.app.)

2. For all operations provided by your Valuation structure, state their (worst-case) time complexity. (The size of a valuation is given by the number of variables that the valuation maps to a value.)

3 A Functor for Propositional Logic

1. Consider the following data type of propositional formulas:

```
datatype formula = True
                 | False
                 | Var of string
                 | Not of formula
                 | And of formula * formula
                 | Or of formula * formula
```

   Define a functor Semantics that, given any structure V that implements valuations (i.e., any structure that matches your VALUATION signature from Exercise 1), returns a structure that matches the following signature:

```
sig
  val truth_value: V.T -> formula -> bool
  val is_taut: formula -> bool
end
```

   Here, V.T is the type of valuations.

   The functions truth_value and is_taut are specified as follows.

   truth_value: Given a valuation and a formula, the function truth_value returns the truth value of the formula under the given valuation. Examples:

   - Given any valuation v, truth_value v maps
     - True to true,
     - False to false.

- Given any valuation v that maps "x" to true and "y" to false, truth_value v maps
  - Var "x" to true,
  - Not (Var "x") to false,
  - And (Var "x", Var "y") to false.

If a variable is mentioned in the formula but not defined in the valuation, truth_value cannot compute a meaningful result. You must decide how to handle this situation.

is_taut: Given a formula, the function is_taut returns true if and only if the formula is a tautology. A formula is a tautology if its truth value is true under all valuations (that define all variables mentioned in the formula). Examples:

- For the formula Var "x", we have to consider two valuations: the valuation that maps "x" to true, and the valuation that maps "x" to false. Under the latter valuation, the truth value of Var "x" is false. Therefore, Var "x" is not a tautology, i.e., is_taut (Var "x") must return false.
- For the formula Or (Var "x", Not (Var "x")), we again have to consider these two valuations: the valuation that maps "x" to true, and the valuation that maps "x" to false. Under either valuation, the truth value of Or (Var "x", Not (Var "x")) is true. Therefore, Or (Var "x", Not (Var "x")) is a tautology, i.e., is_taut (Or (Var "x", Not (Var "x"))) must return true.
- For the formula Or (Var "x", Var "y"), we have to consider four valuations: all possible combinations of mapping "x" and "y" to true or false. In particular, under the valuation that maps both "x" and "y" to false, the truth value of Or (Var "x", Var "y") is false. Therefore, Or (Var "x", Var "y") is not a tautology, i.e., is_taut (Or (Var "x", Var "y")) must return false.

Hint: First, make sure you understand the definition of tautology! To find out whether a formula is a tautology, you can simply check its truth value under a number of valuations. First, write a helper function that collects all variables that occur in a formula. Next, think about which valuations you need to consider to work out if the formula is a tautology, and how these valuations can be generated recursively. (In general, if there are n distinct variables occurring in a formula, then there are 2^n valuations to consider.) Use additional helper functions as necessary.

2. Apply your Semantics functor to your Valuation structure from Exercise 2. Test the resulting truth_value and is_taut functions on at least five different combinations of valuations and formulas. Include the test code in the file that you submit.

4 Simplification of Propositional Formulas

1. Write a function simp: formula -> formula that takes a formula and simplifies it. There are many strategies for simplifying a logic formula. You should at least apply the following rules:
   - Or (True, x) simplifies to True
   - Or (False, x) simplifies to x

- Or (x, True) simplifies to True
- Or (x, False) simplifies to x
- And (True, x) simplifies to x
- And (False, x) simplifies to False
- And (x, True) simplifies to x
- And (x, False) simplifies to False
- Not (Not x) simplifies to x
- Not True simplifies to False
- Not False simplifies to True

In these rules, x refers to any formula. Your simplifier should be able to apply the rules to sub-formulas. Note that application of one simplification rule might allow other simplifications to be applied later. Your simplifier should take advantage of this. For example,

```
Not (And (And (False, Var "x"), Var "y"))
```

can be simplied to

```
Not (And (False, Var "y"))
```

which can in turn be simplified to

```
Not False
```

and further to

```
True
```

Include at least five test cases for your simp function in the file that you submit.

2. Of course, your simp function should terminate for any input formula. Give a variant (or several variants, one for each recursive function used in your simplifier) that shows termination.

Testing

Use the file `lab4_test.sml` (from the Student Portal) to test your solution. See the instructions for Assignment 1 for further information on running the tests.

Some tests might fail because they assume other datatype definitions or function signatures than what you are using. If necessary, *change your solutions* to make all tests compile and run successfully.

Good luck!