Assignment #4: Lambda Calculus & Semi-automatic Program Verification

You are allowed to work on this assignment in pairs.

Hand in Requirements:

Download a4-materials.tar.gz from the course website. Following the directions below and fill in the appropriate missing elements. You are to send a4-materials.tar.gz back to the TA by email by the deadline. It should contain the following modifications:

- Lambda/a4-1.hs -- containing your solutions to part I of the assignment
- a4-2.txt -- containing your hand Hoare proofs to part II of the assignment
- Verifier -- containing your solution to part III

Part I: Lambda calculus Implementations.

Inside the folder Lambda, there are 2 implementations of the lambda calculus from lecture. One is a first-order representation of the lambda calculus and the other is the higher-order representation. Read through the code to make sure you understand it. Implement the following functions inside the file a4-1.hs. Use the QuickCheck code at the bottom of the file to help you test your solution, though that testing is not complete.

- intToLamF :: Int -> F.Lam
- intToLamH :: Int -> H.Lam
- lamFToInt :: F.Lam -> Int
- lamHToInt :: H.Lam -> Int

Part II: Hoare proofs.

Consider the following two programs and their pre- and post-conditions. In the file a4-2.txt, state the loop invariant(s) you to prove each program correct and give a full Hoare logic proof of correctness. In your Hoare proofs, when you use the rule of consequence, you do not have to say why some formula P implies some other formula Q. You just have to be sure the implication is correct. In other words, I do not want to see full proofs of P => Q, I just want to see you use the Hoare rules correctly.

In the following triple 2^n is 2 to the exponent n. Also, we use x ++ y to concatenate string x to string y. We use length (x) to denote the length of string x. You can use any facts about arithmetic or strings that you want (again, without stating them explicitly). For example:

- 2^0 = 1
- 2^1 = 2
- 2^m * 2^n = 2^(m+n)
- x = 2^m ==> 2*x = 2^(m+1)
- length(x) + length(x) = 2* length(x)

a)

{ n >= 0 & i = 0 & v = 1 }
while (i <= n) { i = i + 1; v = 2*v }
{ v = 2^n}

b)

{ n = 2^k & k >= 0 & x = “” }
while (n >1) { n = n / 2; x = “-“ ++ x; }
{ length(x) = k }

Part III: Automating Hoare Logic

The goal of this part of the assignment is develop a semi-automatic tool for program verification using Hoare Logic and to use the tool to verify several small imperative programs. To begin, take the following steps.

Obtain the presburger arithmetic package by invoking cabal in your standard shell (not in ghci). At the prompt, type:

cabal install presburger

To ensure the package is working and accessible load the file PTest.hs into ghci and invoke main. It should compile properly and you should get a message. If you have any problems, please let the TA know ASAP.

Next, examine the other Haskell files involved in this assignment. These files include the following. I have highlighted with the word missing! where there is something for you to do.

- State.hs: defines the type Var for program variables and implements imperative program states
- Imp.hs:
  - defines the syntax for Expressions, Boolean Expressions, Commands and Programs
  - defines a Num type class for Expressions so you can build expressions using +, *, 0, …
  - defines an evaluator for Programs
  - defines well-formedness for Programs -- this is essentially a scope-checker for imperative programs. It verifies that the program only uses the declared variables. (missing!)
- Prover.hs:
  - defines the function prove that will determine the validity of a Boolean Expression (missing!)
  - this file imports Data.Integer.Presburger, which is a theorem prover for logical formulae that include linear arithmetic. The function prove will be defined by converting a BExp into a Data.Integer.Presburger.Formula and then calling the function Data.Integer.Presburger.check to determine the validity of the formula.
- Hoare.hs:
  - defines wp, a function to calculate the weakest precondition of a command, given a post-condition (missing the case for the assignment rule!)
  - defines verify, which will use wp to verify that a given Hoare triple is valid
- a4-3.hs:
  - defines a number of small programs and their post-conditions. Some of the programs are missing! pre-conditions and loop invariants and you need to fill those in in such a way that you create valid Hoare triples that can be proven using your verifier.

For each module further instructions follow.

Imp.hs

The main task in this module is to write the functions good and goodState.

```
good :: Program -> Bool
goodState :: State -> Declarations -> Bool
```

The function good should check that the program is well-formed. Programs look like this:

Prog ds c

where ds is a series of declarations (ie: a list of variables) and c is a command. A program is well-formed if its command only uses variables that appear in its declaration list ds. The function good should return true if a program is well-formed and false otherwise. If you are looking for stylistic tips concerning how to write a function like good in Haskell, examine the function eval. Notice in particular that eval calls evalC, an evaluator for commands and evalC in turn calls evalB and evalE. Implementing a collection of related functions, with one function per type, is a common technique in Haskell.

You might find the function contains useful in your implementation of good. It tests to see whether an item v appears in a list of vs. It's definition appears below. If you don't know what the function any does, you might want to look it up using http://www.haskell.org/hoogle/

```
contains :: (Eq a) => [a] -> a -> Bool
contains vs v = any (\x -> x == v) vs
```

The function goodState should check that the State contains a value for every variable that appears in the list of Declarations.

Prover.hs

The goal of this module is to implement the function

prove :: BExp -> [Var] -> Bool

prove b ds should return True if b is True regardless of what values the variables in ds take on. It should return False if there exists any assignment of values to variables that render b False. Since building a theorem prover is a lot of work, we will use the Data.Integer.Presburger library (called "P" from now on) to do most of it. However, we still need to convert the BExp b into a value f with type P.Formula. If we do that, we can call the function check defined by the library P:

```
P.check f      -- returns True if f is valid
```

Take a look at the web page for the P library:

http://hackage.haskell.org/package/presburger

And especially look at the HOAS sublibrary:

http://hackage.haskell.org/packages/archive/presburger/0.4/doc/html/Data-Integer-Presburger-HOAS.html

At the top of the latter page, under the subtitle "Constructors," you will see a list of data constructors and operators you can use to create Presburger Formulae. For example, if we used an (unqualified) import at the top of our file, we could create formulae as follows:

```
import Data.Integer.Presburger

myTrue = TRUE
myFalse = FALSE
myConjunction = myFalse :/\: myTrue
myDisjunction = myFalse :\/: myConjunction
```

If we use a qualified import, as in Prover.hs, we write the above as follows:

```
import qualified Data.Integer.Presburger as P

myTrue = P.TRUE
myFalse = P.FALSE
myConjunction = myFalse P.:/\: myTrue
myDisjunction = myFalse P.:\/: myConjunction
```

We use qualified imports to avoid name clashes with our own BExp operators. While most of the translation from BExp to P.Formula is straightforward (BExp conjunction is translated to P.Formula conjunction), there is one tricky issue: variables. BExp uses "X id" where id is a String as a variable expression. Also, BExp expressions are *open*, meaning they contain free variables. However, P.Formula are closed and they use *Higher-Order Abstract Syntax*. A BExp b such as:

```
b = X "x" :=: X "y" :==> X "y" :=: X "x"
```

Is translated in to the following (or an equivalent Haskell expression) :

```
P.Forall (\x1 ->
  P.Forall (\y1 ->
    x1 P.:=: y1 P.:=>: x1 P.:=: y1))
```

More generally, a BExp with at most k free variables is translated in to a P.Formula with k universally quantified variables x1, x2, x3, … (universally quantified means "Forall") out front and with the BExp expressions of the form X "x" replaced by the appropriate variables x1, x2, x3.

Hoare.hs

In this module, your job is to fill in the remaining parts of the function wp.

wp :: BExp -> Comm -> BExp

wp is a function that generates Boolean conditions by working backwards. More specifically,

wp post c

generates a BExp which serves as the weakest pre-condition for the command c given that post is the BExp that must be satisfied when the program terminates. Most of the cases for wp have been filled in for you. You must fill in the case for assignment. The weakest precondition for an assignment statement involves substitution, which you must implement:

wp post (x := e) = post[e/x]

a4-3.hs

This module contains a number of small programs, together with post-conditions. Some of the Hoare triples are missing pre-conditions and some are missing loop invariants. In such cases, supply a loop invariant or a precondition that will allow you to prove the triple correct. Read through the file to identify the missing invariants you must supply. Note: when supplying a precondition, you should not supply "FALSE" or any formula equivalent to false. You should supply a formula that is strictly weaker than that.

Play.hs

You do not need to modify or even use this module if you do not want to. However, it is often useful to have a module that you can use to "play around with" various definitions in ghci and create test programs and such. I used Play.hs when creating a solution to this problem set.