The University of Hong Kong

COMP3258: Functional Programming

Assignment 3

Deadline: 23:59, Dec 4, 2017 (HKT)

Notice:

1. Please use the provided template file `A3.hs`. Rename it as `A3_XXX.hs`, with XXX replaced by your UID. In the case that Moodle rejects your .hs file, please submit it as A3_XXX.zip, which includes only one file A3_XXX.hs.

2. Please follow all the type signatures strictly.

1 A Tour of Haskell as a Theorem Prover

LiquidHaskell (LH) refines Haskell's types with logic predicates that let you enforce critical properties at compile time. One of its interesting applications, is to do equational reasoning.

The benefit of using tools such as LH for writing proofs, compared to a paper-and-pencil-style proof, is that the proof can be machine checked and if it passes the checker, it is guaranteed to be correct.

1.1 Installation

Recommended: The online editor is recommended for this assignment. It can generate a permanent link (Permalink), with which you can store your progress. Pressing `Check` will check your proofs. If you see `Safe`, it means your proofs are correct.

For those of you who are adventurous enough and want to do the assignment offline, you can install LH by following the instructions.

1.2 Introduction

In `A3.hs`, we begins with two imports:

```
import Language.Haskell.Liquid.ProofCombinators
import Prelude hiding (length, (++), reverse, foldr, filter,
  Maybe(..), return, (>>=))
```

We import `Language.Haskell.Liquid.ProofCombinators`, a (Liquid) Haskell library that defines and manipulates logical proofs. We also need to hide some datatypes and functions from `Prelude` because we are going to define our own versions.

The following two lines

```
{-@ LIQUID "--exact-data-cons" @-}
{-@ LIQUID "--higherorder" @-}
```

are two annotations that enable Liquid Haskell to extract definitions of datatypes, so that they can later be used in logical propositions, and reason about higher-order functions. Annotations inside `{-@ @-}` are processed by Liquid Haskell. You don't need to understand the details for the purpose of this assignment.

We define lists as:

```
data List a = Nil | Cons a (List a)
```

In the template, we also give definitions for utility functions `length`, `++`, `reverse`, `foldr`. Apart from annotations inside `{-@ @-}`, they are all ordinary Haskell functions.

1.3 Example: Proving a lemma about list length

To give you a taste of how to write proofs in Haskell, let's try to prove the following lemma:

```
length (xs ++ ys) == length xs + length ys
```

Before moving on, you should convince yourself that the above lemma is true for any two lists `xs` and `ys`.

We begin with the following type signature:

```
{-@ lengthConcat :: xs:List a -> ys:List a
                 -> {length (xs ++ ys) == length xs + length ys} @-}
lengthConcat :: List a -> List a -> Proof
```

The Haskell type `List a -> List a -> Proof` reads that `lengthConcat` takes two lists, and produces a proof. In the annotation inside `{-@ @-}`, the Haskell type is then refined to tell precisely what lemma we are going to prove. During the proof, we will justify each step by giving annotations. LH does not process Haskell normal annotations. But this is **required** for this assignment.

We proceed by induction on the first list:

Base Case

The first case is when the first list is empty:

```
{- Induction on xs -}
lengthConcat Nil ys
```

The left-hand side of the equation, namely, `length (xs ++ ys)`, now becomes:

```
lengthConcat Nil ys
  {- Base case: xs = [] -}
  = length (Nil ++ ys)
```

where `xs` is substituted by `Nil`.

Next we know that `Nil ++ ys == ys` by the definition of `++`. We can express this kind of equivalence reasoning by the combinator `==.`. Therefore we can simplify the above equation as follows:

```
lengthConcat Nil ys
  {- Base case: xs = [] -}
  = length (Nil ++ ys)
  {- Definition of ++ -}
  ==. length ys
```

Liquid Haskell also knows quite a lot about linear arithmetic, so we can add a `0` to it:

```
lengthConcat Nil ys
  {- Base case: xs = [] -}
  = length (Nil ++ ys)
  {- Definition of ++ -}
  ==. length ys
  {- plus left identity -}
  ==. 0 + length ys
```

Next we have `length Nil == 0` by the definition of `length`:

```
lengthConcat Nil ys
  {- Base case: xs = [] -}
  = length (Nil ++ ys)
  {- Definition of ++ -}
  ==. length ys
  {- plus left identity -}
  ==. 0 + length ys
  {- Definition of length -}
  ==. length Nil + length ys
```

Now we arrive exactly at the right-hand side of the equation, namely `length xs ++ length ys`. We can conclude this proof with a `***QED`.

```
lengthConcat Nil ys
  {- Base case: xs = [] -}
  = length (Nil ++ ys)
  {- Definition of ++ -}
  ==. length ys
  {- plus left identity -}
  ==. 0 + length ys
  {- Definition of length -}
  ==. length Nil + length ys
  {- xs = [] -}
  *** QED
```

Inductive Case

The inductive case is when the first list has at least one element:

```
lengthConcat (Cons x xs) ys
  {- Inductive case: xs = x:xs -}
  = length (Cons x xs ++ ys)
```

According to the definition of `++`, we know that `Cons x xs ++ ys == Cons x (xs ++ ys)`:

```
lengthConcat (Cons x xs) ys
  {- Inductive case: xs = x:xs -}
  = length (Cons x xs ++ ys)
  {- Definition of ++ -}
  ==. length (Cons x (xs ++ ys))
```

By the definition of `length`, we have:

```
lengthConcat (Cons x xs) ys
  {- Inductive case: xs = x:xs -}
  = length (Cons x xs ++ ys)
  {- Definition of ++ -}
  ==. length (Cons x (xs ++ ys))
  {- Definition of length -}
  ==. 1 + length (xs ++ ys)
```

At this point, you should pause and think about how to proceed. As with any inductive proofs, we have the induction hypothesis, but we haven't used it yet! Now you may be tempted to write something like:

```
lengthConcat (Cons x xs) ys
  {- Inductive case: xs = x:xs -}
```

```
  = length (Cons x xs ++ ys)
  {- Definition of ++ -}
  ==. length (Cons x (xs ++ ys))
  {- Definition of length -}
  ==. 1 + length (xs ++ ys)
  {- Induction Hypothesis -}
  ==. 1 + length xs + length ys
```

which is (morally) correct by the induction hypothesis. However if you copy and paste the above code into the online editor, it will fail to pass the checker. This is because the combinator `==.` is only capable to do some basic reasoning, e.g., arithmetic reasoning. It will not magically search for other lemmas or generate induction hypothesis. This is where we humans need to intervene by providing some hints.

Since we are doing induction on the first list `Cons x xs`, we know that, by the induction hypothesis, `length (xs ++ ys) == length xs + length ys` should hold for any `ys`. We can tell this to Liquid Haskell via the notation `?`, as follows:

```
lengthConcat (Cons x xs) ys
  {- Inductive case: xs = x:xs -}
  = length (Cons x xs ++ ys)
  {- Definition of ++ -}
  ==. length (Cons x (xs ++ ys))
  {- Definition of length -}
  ==. 1 + length (xs ++ ys)
  {- Induction Hypothesis -}
  ==. 1 + length xs + length ys                    ? lengthConcat xs ys
```

The induction hypothesis is represented by a recursive call of `lengthConcat` on `xs` and `ys`, which is exactly the hypothesis we need to justify the aforementioned problematic step. Liquid Haskell will valid if this is the correct justification for the reasoning.

Next according to the length definition, we know:

```
lengthConcat (Cons x xs) ys
  {- Inductive case: xs = x:xs -}
  = length (Cons x xs ++ ys)
  {- Definition of ++ -}
  ==. length (Cons x (xs ++ ys))
  {- Definition of length -}
  ==. 1 + length (xs ++ ys)
  {- Induction Hypothesis -}
  ==. 1 + length xs + length ys                    ? lengthConcat xs ys
  {- Definition of length -}
  ==. length (Cons x xs) + length ys
```

Now we arrive at exactly the right-hand side of the equation. Therefore we can finish the proof by `*** QED`:

```
lengthConcat (Cons x xs) ys
  {- Inductive case: xs = x:xs -}
  = length (Cons x xs ++ ys)
  {- Definition of ++ -}
  ==. length (Cons x (xs ++ ys))
  {- Definition of length -}
  ==. 1 + length (xs ++ ys)
  {- Induction Hypothesis -}
  ==. 1 + length xs + length ys                    ? lengthConcat xs ys
  {- Definition of length -}
  ==. length (Cons x xs) + length ys
  {- xs = x : xs -}
  *** QED
```

You can copy and paste the code up to this point into the online editor and run `Check`. It should give you output `Safe`.

Note:

- You can only check your proof after you arrive at `*** QED`, otherwise you will get a type mismatch error.
- You can check your proof case by case, by using `undefined` for unfinished cases first.
- If you skip important steps or write a not-so-obvious step without giving justification to Liquid Haskell (via the `?` notation), your proof will most likely be rejected (as in the case when we didn't provide the inductive hypothesis). Unfortunately, Liquid Haskell is unable to give very useful error messages. In the situations like this, please check your proof more carefully and make sure you write every step detailed enough to convince Liquid Haskell.
- **In this assignment, annotations are required for every step.**
- To simplify the tasks, all type signatures are provided.

2 More on List

Problem 1. (20 pts.) Using `lengthConcat`, prove that the length of a list is equal to that of itself being reversed.

```
length (reverse l) == length l
```

The type signature is given as follows:

```
{-@ lengthReverse :: xs:List a
                  -> {length (reverse xs) == length xs} @-}
lengthReverse :: List a -> Proof
lengthReverse l = undefined
```

Hint: You can use `lengthConcat` via the `?` notation.

Problem 2. (20 pts.) Prove that `(++)` and `reverse` on lists satisfy the following property.

```
reverse (xs ++ ys) == reverse ys ++ reverse xs
```

The type signature is given as follows:

```
{-@ reverseConcat :: xs:List a -> ys:List a
                  -> {reverse (xs ++ ys) == reverse ys ++ reverse xs} @-}
reverseConcat :: List a -> List a -> Proof
reverseConcat l = undefined
```

You may or may not need to prove following utility lemmas:

```
{-@ concatNil :: xs:List a -> {xs == xs ++ Nil} @-}
concatNil :: List a -> Proof

{-@ concatAssoc :: xs:List a -> ys:List a -> zs:List a
                -> {xs ++ ys ++ zs == xs ++ (ys ++ zs)} @-}
concatAssoc :: List a -> List a -> List a -> Proof
```

Problem 3. (20 pts.) For functions `g` and `h`, and a value `w`, if `f` and `v` satisfy the following:

```
v          == h w
f x (h y) == h (g x y)
```

then the following equation holds:

```
h . foldr g w == foldr f v
```

This is the so-called fusion property of `foldr`. Try to prove that by induction on lists.

In `A3.hs`, to simplify the representation, `v`, `w`, `h`, `g`, `f` are defined, and two assumptions are provided by `p1` and `p2`. The type signature is:

```
{-@ fusion    :: xs:List a -> {h (foldr g w xs) == foldr f v xs} @-}
fusion :: List a -> Proof
```

Hint: Use the assumptions `p1` and `p2` via `?`. Do not forget the arguments for `p2`.

3 Monad Laws

We learned from the lecture that `Maybe` is a Monad:

```
data Maybe a = Nothing | Just a
```

And its Monad instance definition:

```
instance Monad Maybe where
  return          = Just
  (Just x) >>= k = k x
  Nothing >>= _   = Nothing
```

However, Haskell will not check whether this is a proper Monad, in the sense that it should satisfy the following Monad laws:

```
return a >>= k          == k a
m >>= return            == m
m >>= (\x -> k x >>= h) == (m >>= k) >>= h
```

In order to show that Maybe is indeed a valid monad, we are going to prove the monad laws for Maybe. The definitions of `return` and `>>=` are given as:

```
{-@ reflect return  @-}
return :: a -> Maybe a
return a = Just a

{-@ infix   >>= @-}
{-@ reflect >>= @-}
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
(>>=) Nothing _ = Nothing
(>>=) (Just a) f = f a
```

Problem 4. (10 pts.) Prove that the `Maybe` Monad satisfies the left identity law:

```
{-@ left_identity :: v:a -> f:(a -> Maybe b)
                  -> {return v >>= f == f v} @-}
left_identity :: a -> (a -> Maybe b) -> Proof
```

Problem 5. (10 pts.) Prove that the `Maybe` Monad satisfies the right identity law:

```
{-@ right_identity :: m:(Maybe a)
                   -> {m >>= return == m} @-}
right_identity :: (Maybe a) -> Proof
```

Problem 6. (15 pts.) Prove that `Maybe` Monad satisfies the associativity law:

```
{-@ associativity :: m:(Maybe a) -> k:(a -> Maybe b) -> h:(b -> Maybe c)
                  -> {m >>= (\x:a -> k x >>= h) == (m >>= k) >>= h} @-}
associativity :: Maybe a -> (a -> Maybe b) -> (b -> Maybe c) -> Proof
```

You may or may not need the following helper lemma, which essentially states that

```
f x == (\a -> f a) x
```

```
{-@ beta_application :: bd:b -> f:(a -> {bd':b | bd' == bd}) -> x:a
                     -> {f x == bd } @-}
beta_application :: b -> (a -> b) -> a -> Proof
beta_application bd f x
  = f x
  ==. bd
  *** QED
```

Code style and submission (5 pts.)

All functions should be implemented in a single Haskell file, named as A3_XXX.hs, with XXX replaced by your UID. Your code should be well-written (e.g. proper indentation, names, and type annotations) and documented. Please submit your solution on Moodle before the deadline. In the case that Moodle rejects your .hs file, please submit it as A3_XXX.zip, which includes only one file A3_XXX.hs. Please at least make sure your file can compile!

Plagiarism

Please do this assignment on your own; if, for a small part of an exercise, you use something from the Internet or were advised by your classmate, please mark and attribute the source in a comment. Do not use publicly accessible code sharing websites for your assignment to avoid being suspected of plagiarism.