Haskell Assignment 1

Philip Rajani Lassen (vgh804)
Nicolas Ringsmose Larsen (vgn209)

September 2018

1 evalExpr

For the resubmission, we worked on using monadic style more, and finishing up `evalExpr`. We ended up with a working implementation for nearly all cases. The most troublesome cases, aside from the list comprehensions that we did not implement properly (more on this in the assessment section), was when we were given an array of expressions. This is done in expressions of type

`Array [Expr] and Call FunName [Expr]`

The simplest way for us to evalute a list of expressions seemed to be to map `evalExpr` over the list. However, this returns a `[SubsM Value]`. For both our original submission and our resubmission, we spent a lot of time figuring out how to take each `SubsM Value` and append the inner values on to each other.

Enter our helper function `concatSubsM`:

```
concatSubsM :: [SubsM Value] -> SubsM [Value]
```

The function works by folding the `liftM2` function (with the const function inside) over the `[SubsM Value]` list. Thus creating a list of the values inside each `SubsM`, and returning them inside a single `SubsM [Value]`.

For the array comprehensions, we had some issues with the finer points. Again, these will be addressed in the assessment section, this section will go over our design choices instead.
For `ACFor` we used a strategy similar to the `Array` and `Call` case earlier. This time, we mapped a helper function over the list contained in the `Expr` part of `ACFor Ident Expr ArrayCompr`. The helper function esentially binds the `Ident` to a value (each value in the list) and then evaluates the `ArrayCompr` with this binding.
For `ACIf` we simply evaluated the expression and moved on to evaluating the following array comprehension if the first evaluation returned true. However, if it returned false, we decided to return an `UndefinedVal`, even though we really just wanted to return nothing. This would make for a slight problem, as we will see later.

2 Monad Laws

2.1 Left Identity

We need to show that

```
do {x' <- return x
    f x'
    }
is the same as
do {f x}
In our Monad the result of x' in x' <- return x is indeed x. Thus
f x' = f x
Thus we have shown left Identity
```

2.2 Right Identity

We need to show that m >>= return = m
This follows from the fact that

```
m >>= return = SubsM $ \(c1,c2) -> case m of SubsM x ->
                   let var = x (c1,c2) in
                       case var of
                       Right (a, env) -> case (return a) of
                           SubsM y -> y (env,c2)
                           Left er -> Left er
```

And it follows from the definition of *return a* that m ¿¿ return reduces to m in the above function

3 Process

In order to implement the code we spent a large portion of the time reading about Monads. We hoped that this would help speed our ability to produce the code, however we still lacked some understanding. When we wrote the code we had difficulties getting our implementations to pass the type-checker. This was due to our lack of understanding of why functions and types were chosen to be what they were.

4 Assessment

Our implementation seems to work for most cases, and we feel like we have gotten a better grasp on monads for the resubmission. However, there are still three issues with the list comprehensions.
The first one happened if an `ACFor` had an `ACIf` inside. Since our implementation of `ACIf` returns an `UndefinedVal` if the test is false, the resulting array

would contain such values instead of just throwing them away.
We were able to put a bandaid on this problem by filtering out `UndefinedVal` in the `ACFor` case. However, this means that there can never be an UndefinedVal in the resulting array. We are not sure if this matters for list comprehensions, since we already have the `Array(e)` function.

The second problem is that we do not know how to "reset" the context after evaluating the list comprehensions. Thus, all bindings made inside a list comprehensions persists outside it as well.

Lastly, for nested `ACFor` expressions, we return an `ArrayVal [ArrayVal [Value]]`, which was not the intention of the nested for loops. However, we could not come up with a good way of handling this problem.

Overall, we feel more comfortable using monads, but confusions still arise for these special cases.

5 Testing

We were able to pass all but two of the twenty nine tests of the online TA (27/29 were passed). We discussed why our implementation was unable to pass these two tests earlier. For our own testing we used the Tasty Testing framework, and built our Test Suite in Test.hs. We tried to cover some edge cases. We wrote 8 tests. In some of the cases a test may be comparing a list to another list, which means that there are more than 8 tests. The tests can be run by running the command

```
stack test
```

The results should look something like

```
Progress 1/2: subscript-interpreter-0.0.0Tests
  Boolean Tests:   OK
  Number Tests:    OK
  String Tests:    OK
  Undefined Tests: OK
  Array Tests:     OK
  Comma Tests:     OK
  Functions calls: OK
  Other Tests:     OK

All 8 tests passed
```