Damian Franco
dfranco24@unm.edu
101789677
CS-558 001

Homework 5

5.1 An evaluator for the untyped lambda-calculus

a) For the first question, we were asked to implement the terms of untyped lambda calculus in Haskell. This data type would have to account for the three main terms that are in UTLC which are a variable, abstraction, and an application which is shown in the Term definitions below, alongside how I implemented it in Haskell. I took the definitions word for word and then placed them into Haskell for this part. Nothing too crazy or special behind the implementation here. A string data type is used here to represent variables which adds an ease of viewing the terms when printing to the console. Then I added some test terms that I will be using for the rest of the functions I write in the further parts of these problems which I will show below as well. These test terms are also the terms to be in part (f) that were asked to show how we implemented them in Haskell.

My Haskell TERM definition with the UTLC definition:

```haskell
-- Problem 5.1 (a)
data Term = Var String
          | Abs String Term
          | App Term Term
          deriving Show
```

Terms, t	::=	x	*variable*
	|	$\lambda x.t$	*abstraction*
	|	$t\ t$	*application*

Test Terms Initialized:

```haskell
testData1 = Abs "a" (Abs "b" (Var "a"))
testData2 = App (Abs "z" (App (Var "z") (Var "z"))) (Abs "q" (Var "q"))
testData3 = Abs "x" (Abs "y" (Var "y"))
testData4 = Abs "d" (Var "d")
```

b) Part b asks us to define a Haskell function that implements the capture-avoiding substitution operation within UTLC. This substitution operation will take a term as an input and then substitute a term within the term selected with a different term. Usually this is used to do alpha-renamings and will not change how the term is defined, rather it will change the names of the variables within the terms. With that, my substitution function in Haskell takes three inputs. The first is the main term that we want to be substituting into, the second is a string of the term that we would like to substitute for and the last is the term we would like to substitute into the term. This function returns a Term type variable. The function tests for each case for a Term type meaning it first checks if its a variable, abstraction of an application. If it is variable, then we know we want to first check if the current term is equal to the one that the user would like to change. If it is then it will change that term to the new term or it will not change that term if it is not the string we would like to change. The same is done for abstraction types, but this will also need to change the string name in the abstraction and also call substitutes recursively to account for the rest of the term. Lastly, the application type will simply just call the function recursively since we cannot change application string names. Below you will see my definition of the function in Haskell, as well as some tests that I did to test its functionality.

My Haskell SUBSTITUTE definition:

```
-- Problem 5.1 (b)
var2String :: Term -> String
var2String (Var x) = x

subst :: Term -> String -> Term -> Term
subst (Var s) x t = if x == s
                    then t
                    else Var s
subst (Abs t1 t2) x t = if x == t1
                        then (Abs (var2String t) (subst t2 x t))
                        else (Abs t1 (subst t2 x t))
subst (App t1 t2) x t = App (subst t2 x t) (subst t1 x t)
```

Test Terms Initialized and Outcome:

```
substTest1 = subst testData1 "a" (Var "x")
substTest2 = subst testData2 "z" (Var "a")
substTest3 = subst testData3 "y" (Var "w")
```

```
*Main> testData1
Abs "a" (Abs "b" (Var "a"))
*Main> substTest1
Abs "x" (Abs "b" (Var "x"))
*Main> testData2
App (Abs "z" (App (Var "z") (Var "z"))) (Abs "q" (Var "q"))
*Main> substTest2
App (Abs "q" (Var "q")) (Abs "a" (App (Var "a") (Var "a")))
*Main> testData3
Abs "x" (Abs "y" (Var "y"))
*Main> substTest3
Abs "x" (Abs "w" (Var "w"))
```

c) The next function we were asked to implement was the isValue function and this one was very straightforward. Since we know that the only value in UTLC is an abstraction value, then I simply would just return false if the input was anything else but an abstraction value and otherwise true. This was very simple to implement in Haskell using multiple guards with the cases for each term and just returning true or false accordingly. Below you will find images of my definition of the function, as well as some tests.

My Haskell ISVALUE definition with the UTLC definition:

```
-- Problem 5.1 (c)
isValue :: Term -> Bool
isValue (Var s) = False
isValue (Abs t1 t2) = True
isValue (App t1 t2) = False
```

Values, v ::= $\lambda x.t$ *abstraction value*

Test Terms Initialized and Outcome:

```
isValueTest1 = isValue testData1
isValueTest2 = isValue testData2
isValueTest3 = isValue testData3
isValueTest4 = isValue testData4
```

```
*Main> testData1
Abs "a" (Abs "b" (Var "a"))
*Main> isValueTest1
True
*Main> testData2
App (Abs "z" (App (Var "z") (Var "z"))) (Abs "q" (Var "q"))
*Main> isValueTest2
False
*Main> testData3
Abs "x" (Abs "y" (Var "y"))
*Main> isValueTest3
True
*Main> testData4
Abs "d" (Var "d")
*Main> isValueTest4
True
```

d) We were asked to define a single step reduction for our UTLC terms that when called will reduce a term a single time. We know about how reductions work through the lectures and how we have specifically three rules that we must implement to reduce a term. The rules/axioms that we have are the E-App1, E-App2 and E-AbsApp. Reductions are only applied when an application term is available within the term being passed through. This means that a variable and an abstraction being passed through will return two results. We know that we are unable to reduce and are “stuck” if the term being passed through is a variable. We also know that abstractions are values as listed in my last question write up so there is nothing to reduce to when we have a value. I first accounted for these two cases within my definition of eval1. Each of these return Nothing, which is a part of the Maybe data type. The return value here will be Maybe Term because not all terms can be reduced so there will be times where we have to account for those by either passing nothing or a Term data type using the Just keyword. This then leads me to the implementation of the axioms that we know. We know that each axiom is an application type of Term and each has a specific value that will hint towards which axiom we will use. For example, if we know that the first argument for App is a value, using my isValue Haskell function, then we will use the E-App1 axiom which is implemented here which states that the second and only term needs to be evaluated. If we know both arguments are terms within the App arguments, then E-App2 is used and the first term is evaluated while the second term is not. Lastly, if we know we have value in the first argument of App, then we will use the E-AbsApp axiom which calls the substitution method on the second term argument in app and replace it with the second term. This successfully handles the axioms. On the next page you will see how I implemented the eval1 function as well as some tests to check the functionality.

My Haskell EVAL1 definition:

```haskell
-- Problem 5.1 (d)
eval1 :: Term -> Maybe Term
eval1 (Var s) = Nothing -- Stuck at Variable
eval1 (Abs t1 t2) = Nothing -- Found a Value
eval1 (App (Abs t1 t2) s) | isValue s = Just (subst t2 t1 s) -- E-AbsApp
eval1 (App s t2) | isValue s = let Just evalT = eval1 t2 -- E-App1
                               in Just (App s evalT)
eval1 (App t1 t2) = let Just evalT = eval1 t1 -- E-App2
                    in Just (App evalT t2)
```

Test Terms Initialized and Outcome:

```haskell
testData5 = App (Abs "x" (Var "x")) (Abs "z" (App (Abs "x" (Var "x")) (Var "z")))

eval1Test1 = eval1 testData1
eval1Test2 = eval1 testData2
eval1Test3 = eval1 testData3
eval1Test4 = eval1 testData4
eval1Test5 = eval1 testData5
```

```
*Main> testData1
Abs "a" (Abs "b" (Var "a"))
*Main> eval1Test1
Nothing
*Main> testData2
App (Abs "z" (App (Var "z") (Var "z"))) (Abs "q" (Var "q"))
*Main> eval1Test2
Just (App (Abs "q" (Var "q")) (Abs "q" (Var "q")))
*Main> testData3
Abs "x" (Abs "y" (Var "y"))
*Main> eval1Test3
Nothing
*Main> testData4
Abs "d" (Var "d")
*Main> eval1Test4
Nothing
*Main> testData5
App (Abs "x" (Var "x")) (Abs "z" (App (Abs "x" (Var "x")) (Var "z")))
*Main> eval1Test5
Just (Abs "z" (App (Abs "x" (Var "x")) (Var "z")))
```

e) Once we have defined Eval1, we know that we can use it recursively on a term to reduce as many times. The function that we were asked to implement here, Eval, is to serve that purpose. Output here will not be a single term here, rather it will be a list of Maybe Terms. This list will show all the reduction steps taken and how far of a reduction was achieved for each term. The implementation here would just evaluate 1 and then recursively call the inner terms on eval to continue evaluating the term. Each reduced term will then be added to a list and appended to one another. There is one bug in my implementation which I will talk about in part (g) of this homework, but regardless, this seems to be working for the most part. Below you will find the function and some tests for it.

My Haskell EVAL definition:

```
-- Problem 5.1 (e)
eval :: Term -> [Maybe Term]
eval (Var s) = []
eval (Abs t1 t2) = []
eval (App t1 t2) = [eval1 (App t1 t2)] ++ eval t1 ++ eval t2
```

Test Terms Initialized and Outcome:

```
testData6 = App (Abs "x" (Var "x")) (App (Abs "x" (Var "x")) (Abs "z" (App (Abs "x" (Var "x")) (Var "z"))))

evalTest1 = eval testData5
evalTest2 = eval testData6
```

```
*Main> testData5
App (Abs "x" (Var "x")) (Abs "z" (App (Abs "x" (Var "x")) (Var "z")))
*Main> evalTest1
[Just (Abs "z" (App (Abs "x" (Var "x")) (Var "z")))]
*Main> testData6
App (Abs "x" (Var "x")) (App (Abs "x" (Var "x")) (Abs "z" (App (Abs "x" (Var "x")) (Var "z"))))
*Main> evalTest2
[Just (App (Abs "x" (Var "x")) (Abs "z" (App (Abs "x" (Var "x")) (Var "z")))),Just (Abs "z" (App (Abs
"x" (Var "x")) (Var "z")))]
```

f) After all Haskell functions have been implemented, we were now asked to represent some terms within Haskell using our data type. The terms are listed below, alongside the implementation of how I represented them using my Term data type.

My Haskell terms definition:

(i) $\lambda a.\, \lambda b.\, a$

(ii) $(\lambda z.\, z\, z)\ (\lambda q.\, q)$

(iii) $\lambda x.\, \lambda y.\, y$

```
-- Probem 5.1 (f)
testDatai = Abs "a" (Abs "b" (Var "a"))
testDataii = App (Abs "z" (App (Var "z") (Var "z"))) (Abs "q" (Var "q"))
testDataiii = Abs "x" (Abs "y" (Var "y"))
```

g) For the final part, we were asked to reduce a term that is given to us. This is where I experienced some issues. I believe I know where the issues lie and wish I started the homework earlier to smooth out these bugs. When I put my term in the Eval1 function, I get a non-exhaustive pattern error. I believe it is due to me missing some important cases for some terms within my Eval1 function. I noticed that it will actually reduce till a term with App has an argument of term and value which I do not have a test case for within my eval1 function. I know where the issue lies and even after I turn in this homework, I will make sure to fix this bug because it is something fairly fixable. I did attempt to fix this last minute by adding some more cases to catch this value, but no luck. You will see it reflected in my Haskell file. My output and the set up of my test data is shown below.

My Haskell terms definition:

```
-- Problem 5.1 (g)
testData7 = App (App (Abs "a" (Abs "b" (Var "a"))) (App (App (Abs "z" (Var "z")) (Var "z")) (Abs "q" (Var "q")))) (Abs "x" (Abs "y" (Var "y")))
evalTest3 = eval testData7
```

```
*Main> testData7
App (App (Abs "a" (Abs "b" (Var "a"))) (App (App (Abs "z" (Var "z")) (Var "z")) (Abs "q" (Var "q"))))
(Abs "x" (Abs "y" (Var "y")))
*Main> evalTest3
[Just (App (App (Abs "a" (Abs "b" (Var "a"))) (App (App (Abs "z" (Var "z")) *** Exception: HW5_DamianF
ranco.hs:51:36-56: Non-exhaustive patterns in Just evalT
```