Module code: CS4201
Matriculation number: 200017941

CS4201 P1 – Lambda Calculus

Overview

The objective of this practical is to implement a lambda calculus interpreter for simple lambda expressions. This involves type-checking to ensure that the rules of the type-system are being followed. Additionally, these expressions are required to be evaluated, such that they are reduced to a normal form using the rules of Alpha and Beta reduction. My implementation addresses these baseline requirements as I have written the code for Check.hs, Eval.hs and Quote.hs. As a result, it can type check and evaluate basic lambda expressions which includes variables, lambda abstractions and function applications. There are some limitations with the baseline implementation which will be addressed in the 'Evaluation and Conclusion' section. The parser has largely been untouched and so it hasn't been extended to implement any of the additional functionalities.

Design and Implementation

Check.hs:

The code for type-checking was designed so that it precisely follows the data structure of inferable and checkable terms as defined within the AST module. Here ITerm specifies the structure of annotations, bound variable, free variables, and function applications, and CTerm specifies the structure of lambda abstractions and their parameters. So, each function within Check is responsible for dealing with a specific type of term which has a specific data structure, and this is broadly divided into either being inferable or checkable.

There are three main aspects to my implementation. Firstly, it can check that global variables have valid types and that the type of these types is also valid. Secondly, it ensures that the type of global variables within function applications are themselves valid global variables. Lastly, it checks that the type of definition of lambda abstractions are defined with the valid type of types.

Global variables can either be annotated or have no annotation. Given a global variable that is annotated, this will contain the value assigned to the variable as well as the type of that variable in the format 'z :: n', where z is the value and n is the type. This is a checkable type and is handled by 'iType ii g (Ann cterm t)' since it is the designated function to handle annotations. The given type information is extracted from the annotation through pattern-matching and is passed to 'cType ii g (Inf iterm)' which handles type information for variables. This function extracts the name of the type and then calls 'iType ii g (Free name)' which goes through the list of variables already defined within the context variable. The given type name has the format 'Global string' since this is how global variables are stored within the context variable and so is used as the key to search through the context which contains a list of key-value pairs. If there is a match, the type information stored as the value is returned. This is then compared with the type information provided in the annotation to see if they match, ensuring that it is a valid type. On the other hand, when it comes to global variables with no annotation, the type can be inferred. The given global variable is directly passed to 'iType ii g (Free name)' which checks that the value assigned to the variable is within the context variable.

Lambda abstractions need to be annotated with a function type and therefore it can only be a checkable term. Since it is annotated, it goes through 'iType ii g (Ann cterm t)' which then calls 'cType ii g (Lam cterm)'. This handles lambda abstractions by extracting only the parameters and the type information which is then passed to 'cType ii g cterm (Fun ty1 ty2)'. This function handles checkable terms that have a functional type. Here, both ty1 and ty2 are checked to see whether they are a variable type or a function type and are then passed to the appropriate function to perform the type check. Note that here recursion is possible where if one of them is a function type then the same function is called again. This is important as it ensures that every possible function type is handled. Once, the variable types have been extracted they are then passed to 'cKind g (TFree x) Star' since this is the designated function to type check types themselves.

Function applications are basically global variables and lambda abstractions put together, therefore the implementation for this is trivial as it must simply reuse the functions that have already been defined. This is handled by 'iType ii g (iterm :: cterm)'. First, the annotated lambda abstraction is extracted and type-checked and then the global variables that the abstraction is being applied to are also checked. Once again recursion is involved here where if the iterm within the function application data structure is an application itself, then iType is called again and this time it goes to the same function instead of the one for annotated terms.

Eval.hs:

After type-checking, the interpreter must evaluate the expressions so that they are reduced to their normal forms. Normal forms have the data structure defined in the AST module by Value and the implementation for the Eval module follows this structure. Normal forms can either be variables, lambda abstractions or function applications, and each case is handled by the designated function. The overall design and structure of the code is almost identical to the Check module because the same parsed expressions which are passed to Check.hs in the REPL module are also passed to Eval.hs. However, instead of handling type information, the values of the terms are handled since this is the objective of evaluators. Evaluation of inferable and checkable terms are separated in the same way as the Check module.

Annotated global variables are passed to 'iEval (Ann cterm t) d' which extracts the cterm variable and immediately passes it to 'cEval cterm d'. This function extracts the iterm variable and passes it to 'iEval (Free name) d' which searches for the value of the global variable within NameEnv Value. This is the environment variable and contains a list of variable names and their values. If the value for the variable is already stored in the environment this value is extracted and is the normal form. If the variable cannot be found in the list, then the provided value is the normal form. Note here that the output for this must follow the data structure defined by Value. For a global variable without any annotation, it simply goes through the 'iEval (Free name) d' function where the normal form is evaluated in the same way. In both these cases, the normal form is the value stored in the variable which has the form VNeutral Nfree x, and the value retrieved is converted so that it follows this format.

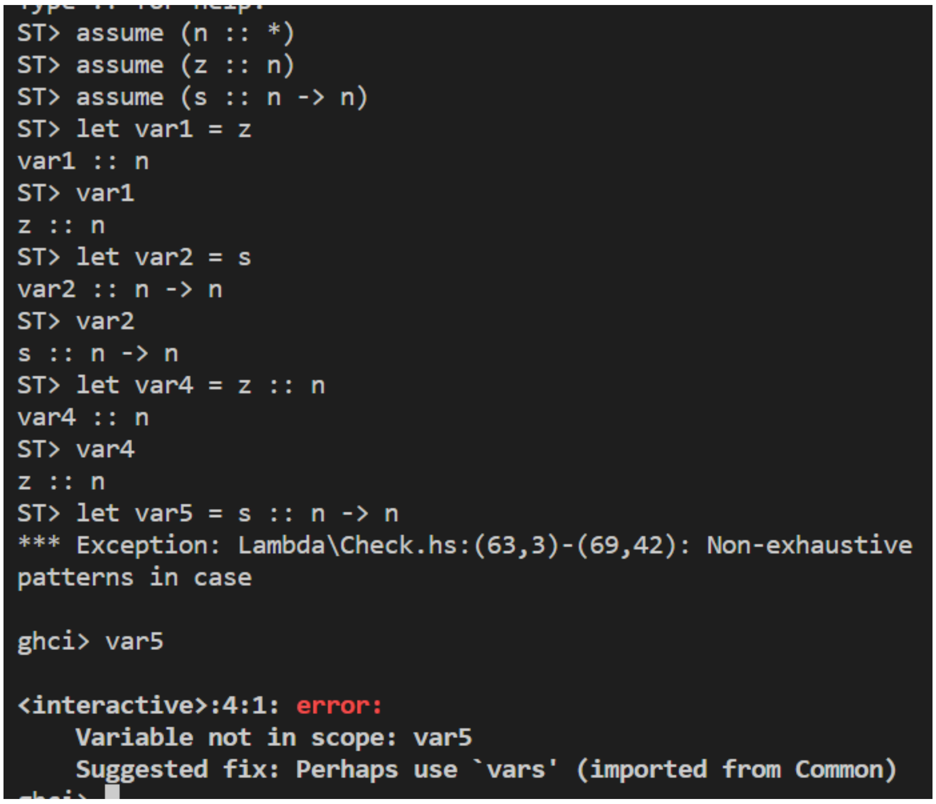
For Lambda abstractions, this time the 'iEval (Ann cterm t) d' calls 'cEval (Lam cterm)' which is responsible for evaluating lambda abstractions. This function uses the VLam constructor, again to follow the format given by Value, and creates a closure '(\arg -> cEval cterm (fst d, arg : snd d))', which captures the behavior of the lambda abstraction. In this lambda function, 'arg' is the argument that would be passed to the lambda abstraction. The argument is stored in 'snd d' which represents Env and is a key component for performing beta reduction later. Given an argument, the lambda function calls 'cEval (Inf iterm) d' with the cterm being the bounded variables within the body of the lambda abstraction. Following this a call is made to 'iEval (Bound index) d' which is responsible for handling these bounded variables. Here, 'index' is the De Brujin index and is a way to represent the parameters without explicitly stating them. It represents the order in which the substitution should be done where an index of 0 takes the highest priority. Take for example, the abstraction Lam (Inf (Bound 1)). 'iEval (Bound index) d', receives Bound 1 as the iterm. When this abstraction is applied to a variable y, Env contains the value of this variable which is also passed along with 'Bound 1'. The function then simply goes through Env and takes the value stored in the position specified by the 'index' variable which in this case is 1. This is how beta reduction is performed as 'Bound 1' is then substituted by the value of 'y'. Note that this evaluation only happens after an argument is provided. If there are no arguments provided, then the normal form of the lambda abstraction is the lambda abstraction itself and no reduction occurs. Therefore, the provided lambda abstraction is simply returned as the evaluated result, which is the closure representing the lambda abstraction.

Function applications are handled by 'Eval (iterm :@: cterm) d'. First the iterm variable is checked to see whether it is a lambda abstraction or another function application. If it is a lambda abstraction (VLam f), then it cEval cterm d' is called to evaluate the arguments that the lambda abstraction is being applied on. This goes through the same process as stated above for global variables since these arguments have a global scope. This evaluation results in the value of the argument being obtained and then 'f' is applied to the argument. Here, 'f' represents the closure which represents the behavior of the lambda abstraction (of performing beta reduction). The normal for in this case is the lambda abstraction being applied to the argument through beta reduction. On the other hand, if iterm is another function application (VNeutral n), it constructs a new neutral term which represents the application of the neutral term 'n' to the argument 'cEval cterm d'. This makes use of 'NApp' which is the constructor to represent the application.

Testing and Debugging

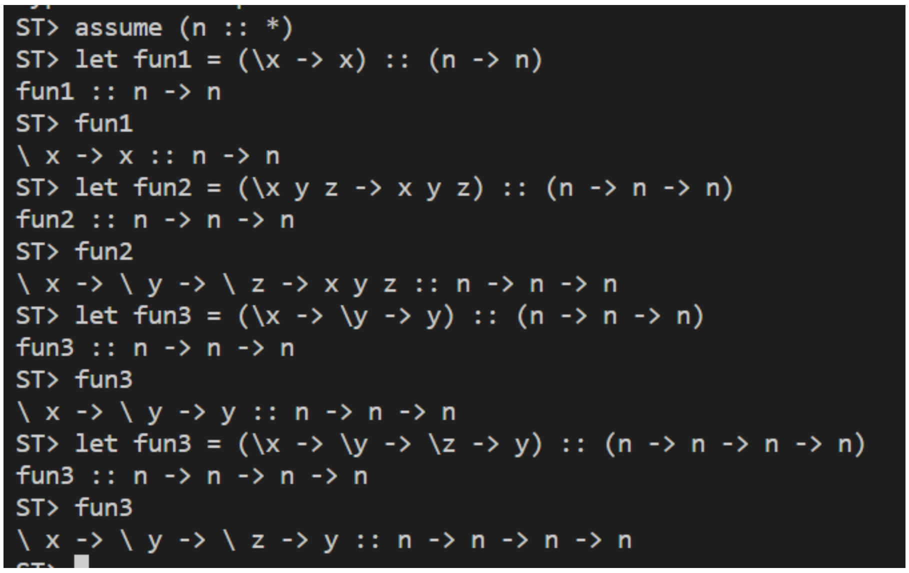
During the development process, I made sure to test my code after every incremental update I made to relevant modules. By doing this I was able to pinpoint exactly which part of the code was causing the error. Initially, the tests mainly involved checking to see that the code was able to be compiled without any errors. Later, after having enough of the functionality implemented, I ran tests to make sure that these functionalities work. I did this by creating a bunch of text files where each is responsible for testing a particular feature. Each of these files have expressions that are relevant to the feature the file is intended to test. The results of the testing are shown below:

- Testing that the interpreter can handle variables with and without annotations.



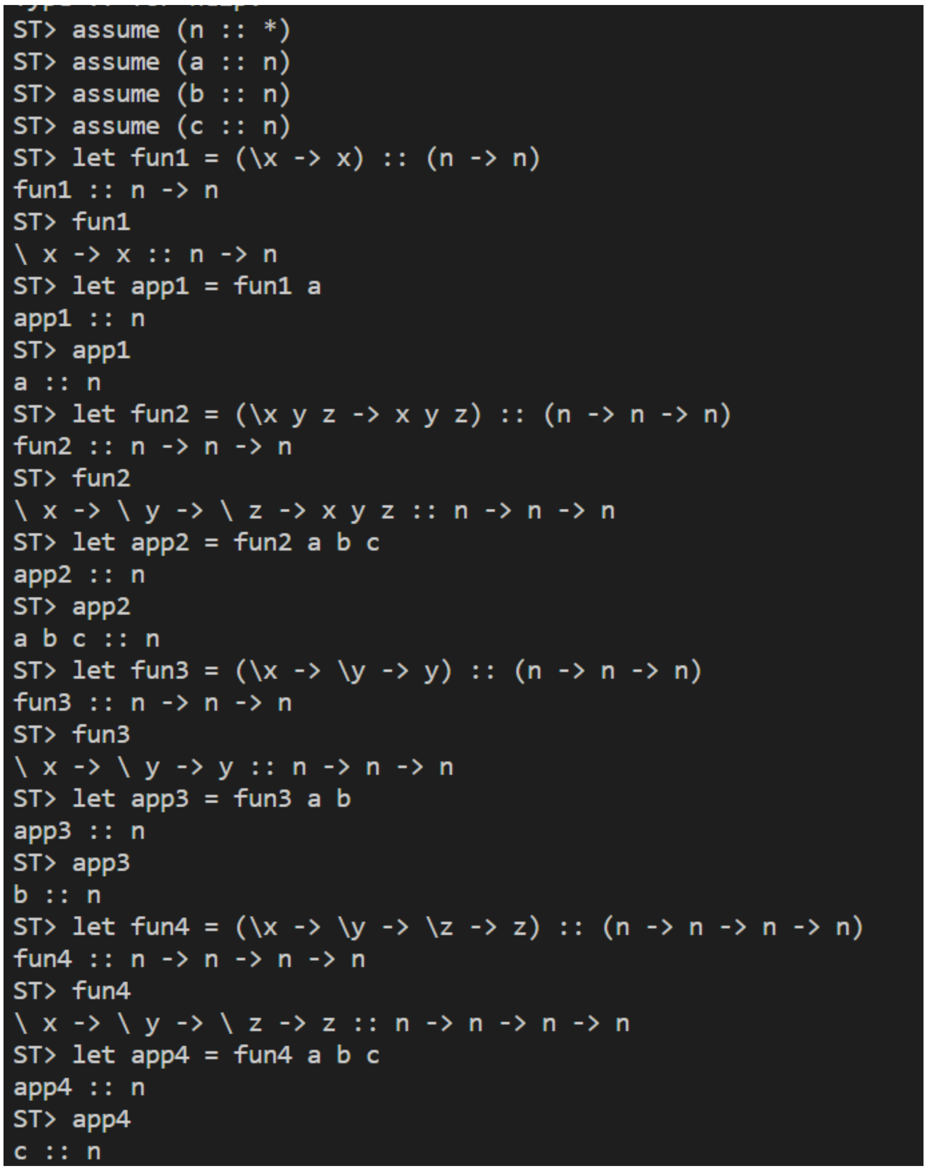
```
ST> assume (n :: *)
ST> assume (z :: n)
ST> assume (s :: n -> n)
ST> let var1 = z
var1 :: n
ST> var1
z :: n
ST> let var2 = s
var2 :: n -> n
ST> var2
s :: n -> n
ST> let var4 = z :: n
var4 :: n
ST> var4
z :: n
ST> let var5 = s :: n -> n
*** Exception: Lambda\Check.hs:(63,3)-(69,42): Non-exhaustive
patterns in case

ghci> var5

<interactive>:4:1: error:
    Variable not in scope: var5
    Suggested fix: Perhaps use `vars' (imported from Common)
```

- Testing that the interpreter can handle lambda abstractions.



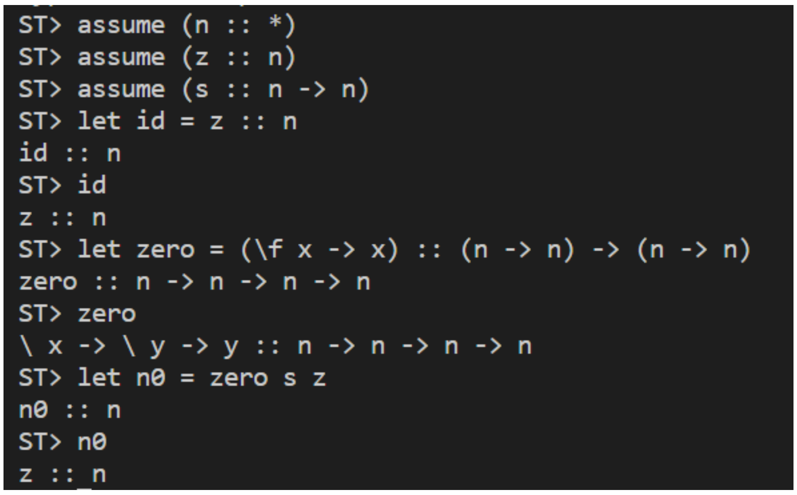
```
ST> assume (n :: *)
ST> let fun1 = (\x -> x) :: (n -> n)
fun1 :: n -> n
ST> fun1
\ x -> x :: n -> n
ST> let fun2 = (\x y z -> x y z) :: (n -> n -> n)
fun2 :: n -> n -> n
ST> fun2
\ x -> \ y -> \ z -> x y z :: n -> n -> n
ST> let fun3 = (\x -> \y -> y) :: (n -> n -> n)
fun3 :: n -> n -> n
ST> fun3
\ x -> \ y -> y :: n -> n -> n
ST> let fun3 = (\x -> \y -> \z -> y) :: (n -> n -> n -> n)
fun3 :: n -> n -> n -> n
ST> fun3
\ x -> \ y -> \ z -> y :: n -> n -> n -> n
```

- Testing that the interpreter can handle function applications.

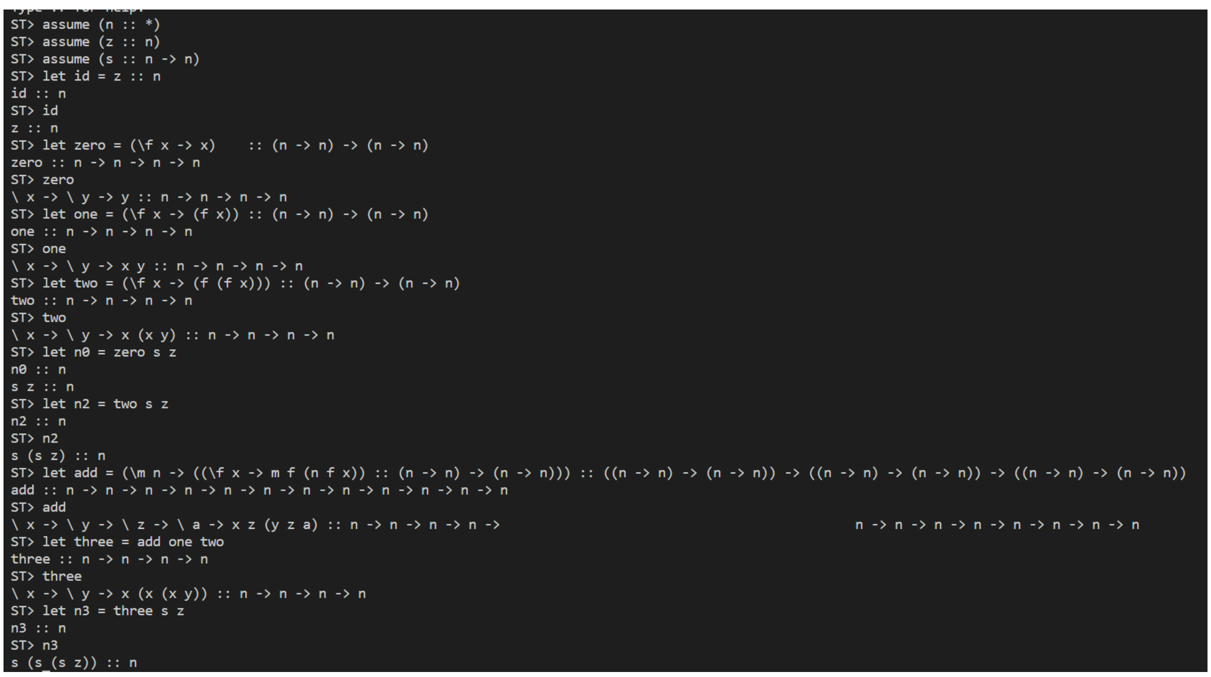


```
ST> assume (n :: *)
ST> assume (a :: n)
ST> assume (b :: n)
ST> assume (c :: n)
ST> let fun1 = (\x -> x) :: (n -> n)
fun1 :: n -> n
ST> fun1
\ x -> x :: n -> n
ST> let app1 = fun1 a
app1 :: n
ST> app1
a :: n
ST> let fun2 = (\x y z -> x y z) :: (n -> n -> n)
fun2 :: n -> n -> n
ST> fun2
\ x -> \ y -> \ z -> x y z :: n -> n -> n
ST> let app2 = fun2 a b c
app2 :: n
ST> app2
a b c :: n
ST> let fun3 = (\x -> \y -> y) :: (n -> n -> n)
fun3 :: n -> n -> n
ST> fun3
\ x -> \ y -> y :: n -> n -> n
ST> let app3 = fun3 a b
app3 :: n
ST> app3
b :: n
ST> let fun4 = (\x -> \y -> \z -> z) :: (n -> n -> n -> n)
fun4 :: n -> n -> n -> n
ST> fun4
\ x -> \ y -> \ z -> z :: n -> n -> n -> n
ST> let app4 = fun4 a b c
app4 :: n
ST> app4
c :: n
```

- Testing that the interpreter can work on the examples given in the specification and produce the expected results.



```
ST> assume (n :: *)
ST> assume (z :: n)
ST> assume (s :: n -> n)
ST> let id = z :: n
id :: n
ST> id
z :: n
ST> let zero = (\f x -> x) :: (n -> n) -> (n -> n)
zero :: n -> n -> n -> n
ST> zero
\ x -> \ y -> y :: n -> n -> n -> n
ST> let n0 = zero s z
n0 :: n
ST> n0
z :: n
```



```
ST> assume (n :: *)
ST> assume (z :: n)
ST> assume (s :: n -> n)
ST> let id = z :: n
id :: n
ST> id
z :: n
ST> let zero = (\f x -> x)    :: (n -> n) -> (n -> n)
zero :: n -> n -> n -> n
ST> zero
\ x -> \ y -> y :: n -> n -> n -> n
ST> let one = (\f x -> (f x)) :: (n -> n) -> (n -> n)
one :: n -> n -> n -> n
ST> one
\ x -> \ y -> x y :: n -> n -> n -> n
ST> let two = (\f x -> (f (f x))) :: (n -> n) -> (n -> n)
two :: n -> n -> n -> n
ST> two
\ x -> \ y -> x (x y) :: n -> n -> n -> n
ST> let n0 = zero s z
n0 :: n
s z :: n
ST> let n2 = two s z
n2 :: n
ST> n2
s (s z) :: n
ST> let add = (\m n -> ((\f x -> m f (n f x)) :: (n -> n) -> (n -> n))) :: ((n -> n) -> (n -> n)) -> ((n -> n) -> (n -> n)) -> ((n -> n) -> (n -> n))
add :: n -> n -> n -> n -> n -> n -> n -> n -> n -> n -> n -> n
ST> add
\ x -> \ y -> \ z -> \ a -> x z (y z a) :: n -> n -> n -> n ->                                             n -> n -> n -> n -> n -> n -> n -> n
ST> let three = add one two
three :: n -> n -> n -> n
ST> three
\ x -> \ y -> x (x (x y)) :: n -> n -> n -> n
ST> let n3 = three s z
n3 :: n
ST> n3
s (s (s z)) :: n
```

Evaluation and Conclusion

The main limitation with the current implementation is with the type-checker. Currently, it does perform type-checking since it makes sure that the types of variables, lambda abstractions and function applications are valid. But what it doesn't do it make sure that the type information accurately describes the term. For example, given a lambda abstraction with a function type. If the type information doesn't match with the parameters of the lambda abstractions, it doesn't report an error as long as the types within the type information are valid types themselves. Apart from that, my implementation seems to work reasonably well with everything else and is therefore able to interpret simple lambda calculus expressions. However, there seems to be specific scenarios where an error is unexpectedly reported. I wasn’t able to fully document this due to time constraints.