Exercise 04 solution

Task 1: Deferred substitutions and closures

Consider the interpreters for FLAE.

1. In the interpreter with environments we introduced closures. What is a closure? What do we need them for ?

   Answer: A closure is a function definition combined with the environment at function-definition time. We need closures for implementing static scoping in a language with first-class/higher-order functions: The closure "remembers" exactly the bindings for free identifiers used within the function definition that was valid when the function was defined.

2. Why did we not need closures in the interpreter with substitution ?

   Answer: The interpreter with substitution immediately replaces all bindings in function definitions within let-bodies. Hence, any binding is "automatically remembered", since it is directly inlined into the function definition.

3. Why did we not need closures in the *F1LAE* interpreter with environments ?

   Answer: In the F1LAE interpreter, functions were not first-class functions and hence could not be defined within a program expression. Here, functions could only be pre-defined in an external environment given to the interpreter for evaluating a program expression. Therefore, when implementing static scoping, it was simply not possible to bind free identifiers in function definitions (neither when using substitution, nor when using environments!). Hence, there was no need for closures. (And when implementing dynamic scoping, the environment would always dynamically contain any binding that we need during evaluation, just as in the dynamically scoped interpreter for FLAE, where you don't need closures anyway.)

Task 2: Lambda Calculus

In the lecture, we introduced first-class functions. To have a minimal *Turing Complete* language, we only need 'Id', 'Fun' and 'App'. This language is called as LAMBDA CALCULUS. Refer to the interpreter *FEInterp.scala.*

The grammar of Lambda Calculus is given by

$$e ::= x \mid \lambda x.e \mid e\ e$$

1. Convert following expressions from our language into lambda expressions

    1. `App(Fun('x,'x),'z)`

        Answer: $(\lambda x.x)\ z$

    2. `App(Fun('x,App(Fun('y,App('x,'y)),'x)),Fun('z,'p))`

        Answer: $(\lambda x.((\lambda y.(x\ y))\ x))\ (\lambda z.p)$

    3. `App(App(Fun('x,App(Fun('y,App(Fun('x,Fun('y, 'x)),'z)),'z)), Fun('p,'p)), 'q)`

        Answer: $(\lambda x.(\lambda y.((\lambda x.\lambda y.x)\ z)\ z))(\lambda p.p)\ q$

2. What are *values* ? and how do we represent them in the Lambda Calculus ?

    Answer: Values are the instances of a particular type of data. They can be considered as the members of a particular set(s), where the set represents a data type. E.g. TRUE and FALSE are the values of type *Boolean*, ... -1, 0, 1, 2 ... are the values of type *Integer* etc. In the Lambda Calculus, we represent values using functions (lambda abstractions).

3. What are the interesting steps of the interpreter that evaluates the expressions of the lambda calculus ?

    Answer: It can be observed that the interpretation of an *App* term is the only case that makes any progress in the interpretation process. We reduce an *App* term using the beta reduction. The interpreter also incorporates some additional checks e.g. *FunExp* in an App term should be a valid function, check for unbound *Id* terms etc.
    (note: we did not consider these checks while reducing a lambda expression by hand.)

    Beta reduction is used to further reduce and evaluate the lambda expressions. Rule for the beta reduction is given by

$$(\lambda x.e_1)\ e_2 \rightarrow e_1[x/e_2]$$

1. Use beta reduction to reduce the following expressions by hand.

    1. $\lambda x.(\lambda y.y\ x)\ z$

        Answer: z

    2. $(\lambda f.\lambda g.((f\ g)(g\ f)))\ (\lambda h.(h\ h))\ (\lambda x.x)$

        Answer: $\lambda h.(h\ h)$

    3. $((((\lambda x.(\lambda y.(\lambda f.((x\ f)(y\ f)))))\ (\lambda g.(\lambda h.(h\ g))))(\lambda h.z))\ p)$

        Answer: $(z\ p)$

2. Use scala interpreter to reduce the following expressions.

   1. $(\lambda x.x)\ (\lambda x.x)$

      Answer: `App(Fun('x,'x), Fun('x,'x))`

      and reduces to : `Fun('x,Id('x))`

   2. $(\lambda x.x)\ (\lambda x.x)\ y$

      Answer: `App(App(Fun('x,'x), Fun('x,'x)), 'y)`

      and reduces to : error (unbound Id 'y)

   3. $(\lambda x.\lambda y.x)\ (\lambda x.x)\ (\lambda f.\lambda g.\lambda h.(f\ g\ h))\ (\lambda x.\lambda y.y)$

      Answer:
      ```
      App(App(App(Fun('x, Fun('y, 'x)), Fun('x, 'x)), Fun('f,
      Fun('g, Fun('h, App(App('f, 'g), 'h))))), Fun('x, Fun('y,
      'y)))
      ```

      and reduces to: `Fun('x, Fun('y, Id('y))`

Task 3: Data representation in Lambda Calculus

How can different data be represented in Lambda Calculus ? E.g. numbers, booleans and different other data structures. Data needs to be represented in the form of functions.

Represent following data / data structures using lambda expressions and using our FEInterp language.

Answer: We represent values using functions (lambda abstractions).

Value	Lambda Expression	FEInterp Expression
TRUE	$(\lambda xy.x)$	`Fun('x, Fun('y, 'x))`
FALSE	$(\lambda xy.y)$	`Fun('x, Fun('y, 'y))`
num 0	$(\lambda fn.n)$	`Fun('f, Fun('n, 'n))`
num 1	$(\lambda fn.fn)$	`Fun('f, Fun('n, App('f, 'n)))`
If-then-else	$(\lambda cte.cte)$	`Fun('c, Fun('t, Fun('e, App(App('c, 't), 'e))))`