Problem 4

Adding the \textit{fix} operator to the \(\lambda\)-calculus, we get expressions of the form:

\[ e \rightarrow x \mid \lambda(x.e) \mid ap(e; e) \mid fix(e) \]

We want to define \textit{fix} in such a way that

\[ \text{fix}(\lambda(f.b)) \downarrow b[\text{fix}(\lambda f.b)/f] \]

But we can see that this is just the same as \(ap(\lambda(f.b), fix(\lambda f.b))\). By the definition of \(ap\), \(fix(\lambda f.b)\) will be substituted for \(f\) in \(b\), so this will evaluate exactly as specified by the evaluation rule. Perhaps we can define in general

\[ \text{fix}(t) = ap(t; \text{fix}(t)) \]

. In the case of variables \(x\), \(fix(x)\) will diverge, which is appropriate since it doesn’t make sense to take the fixpoint of just a variable. Also, with this definition of \textit{fix}, \(fix(ap(t_1, t_2))\) will be \(ap(ap(t_1, t_2), fix(ap(t_1, t_2)))\). So, if \(ap(t_1, t_2)\) evaluates to a \(\lambda\) abstraction, then this definition will satisfy the evaluation rule, and if it evaluates to a variable, it will diverge. However, we can see that rule will diverge with a call-by-value evaluation strategy, but since this is just abstract \(\lambda\)-calculus, we can say that if there is a sequence of \(\beta\)-reductions that converge to a value, then with this definition of \textit{fix}, we have defined a superset of the \(\lambda\)-calculus such that \textit{fix} obeys the evaluation rule given above.

Problem 5

First, I will give a \(\lambda\)-expression representing \textit{add} assuming that there is recursion, and then I will transform it into a solution that uses the fixpoint combinator and the notation from class. I am assuming that we have the expression \(id = \lambda x.x\).

First I will define a \(\lambda\)-expression that takes in an integer \(a\) and returns a new function that takes in an integer \(b\) and returns \(a + b\).

\[ \text{add} = \lambda a.\text{case}(a; \text{id}; (\lambda t.\lambda b.S((\text{add } t)b))) \]

It is clear that this expression expresses the correct idea. If \(a\) is 0, it simply returns the identity function, since \(0 + b = b\). If \(a > 0\), it returns a function that takes in an integer \(b\), applies \(a - 1\), to it, and then returns the successor of that.

This expression actually does not converge because \textit{add} is unbound in the body, we must use the \textit{fix} expression so that we are able to use recursion.
\[ add = \text{fix}(\lambda f.\lambda a.\text{case}(a; \text{id}; (\lambda t.\lambda b.S((ft)b)))) \]

This add function is actually a lambda expression that is essentially equivalent to \( \lambda a.\lambda b.a + b \) since add is an expression that takes in an integer \( a \) and returns a function that takes in another integer \( b \) and returns \( a + b \). So, we can define

\[ \text{add}_\eta = \lambda m.\lambda n.(\text{add } m)n \]

But since \( \text{add}_\eta \) is just an \( \eta \)-expanded version of add, it is clear to see that add is the addition expression we were trying to find.

In the notation used in the course notes

\[ \text{add} = \text{fix}(\lambda f.\lambda a.\text{case}(a; \text{id}; (\lambda t.\lambda b.S(\text{ap}(\text{ap}(f; t); b)))))) \]

Now, to define the multiplication expression, I will assume that we have an expression \( Z = \lambda x.0 \). This is a function that ignores its argument and returns 0.

Following in the same vein as add, we can define mul as

\[ \text{mul} = \text{fix}(\lambda f.\lambda a.\text{case}(a; Z; (\lambda t.\lambda b.\text{add } b((ft)b)))) \]

This is similar to add. The expression takes in an integer, if the integer is 0 then it returns a function that ignores its input and returns 0 \((0 \cdot b = 0)\), and if the integer is greater than 0, it returns a function that takes in another integer, multiplies it by \( a - 1 \), and adds \( b \) to it. Thus, this is an expression that takes in 2 integers, and returns their product.

Exponentiation is a little bit trickier than mul since we want to do the same thing as in mul except flip the arguments. Since the special case is when the second argument is 0. So, we will do much the same thing as in mul, except we will flip the order of the arguments.

\[ \text{pow}’ = \text{fix}(\lambda f.\lambda a.\text{case}(a; (\lambda x.S(0)); (\lambda t.\lambda b.\text{mul } b((ft)b)))) \]

The problem with \( \text{pow}’ \) is that \( \text{pow}’ a b = b^a \). We can fix this by defining

\[ \text{pow} = \lambda a.\lambda b.\text{pow}’ b a \]

Now we have a working exponentiation function.

Just for fun, here is the Haskell code I used to check whether the expressions I defined above actually perform correctly.

```
alcase 0 a b = a
alcase x a b = b (pred x)
fix f = f (fix f)

add :: Integer -> Integer -> Integer
add = fix (\f -> \a -> alcase a id (\t -> \b -> succ ((f t) b)))

mul :: Integer -> Integer -> Integer
mul = fix (\f -> \a -> alcase a (const 0) (\t -> \b -> (add b) ((f t) b)))

pow’ :: Integer -> Integer -> Integer
pow’ = fix (\f -> \a -> alcase a (const 1) (\t -> \b -> (mul b) ((f t) b)))

pow = flip pow’ — flip f = \x y -> f y x
```