Minh Le

Abstraction and Design in Computation

4 May 2022

Final Project Writeup

I implemented two extensions for my final project. The first extension was implementing the Lexicallly-scoped environment model evaluator, which followed many of the same rules as the Dynamically-scoped environment model evaluator. I then abstracted all of the similar code in the three different evaluators into one function, and utilized that abstraction function to implement the three different evaluators very simply.

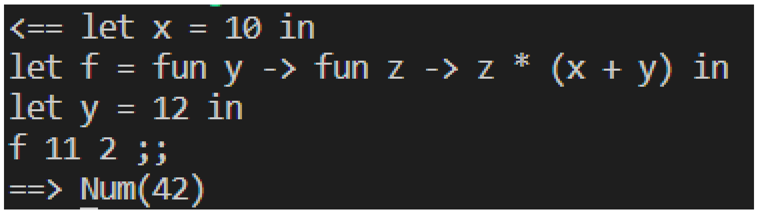
To start, I created three different helper functions to help implement the evaluators: the first function takes in a value and returns it as an expression; the second function takes a binary operation and two expressions and returns the expression where the operation was applied to both of the expressions; the third function takes an expression and evaluates it as a unary operation if possible.

The Lexically-scoped evaluator evaluates very similarly compared to the Dynamically-scoped evaluator, except in the cases that the expression is a Letrec, Fun, or App. For Letrec, if we say the expression looked like “let rec x = D in B,” I implemented the lexical Letrec evaluation using the 5-step process described in the textbook, where I first extend the current environment to contain a binding between x and Unassigned. I then evaluate expression D with this updated environment and assigned it to a variable v_D. I then changed the environment to create a binding between x and v_D and evaluated expression B in this new environment to see what is returned. For Fun, if we say the expression looks like “fun x -> B,” I implemented the lexical Fun evaluation by returning a closure containing both the current environment and the entire function expression. For App, if we say the expression abstractly looks like “P Q,” I implemented the lexical App evaluation by first evaluating expression P in the environment. I then match this evaluation with two cases: either a Closure containing a Fun and an environment or anything else, in which case I raise an error. In the match case of the Closure containing a Fun and an environment, I first evaluate expression Q (we call this v_Q) and update the current environment to contain a binding between x and v_Q. I then evaluate the body of the function in this updated environment to see what is returned.

In terms of abstracting all of the similar code in the three different evaluators into one function, in the cases of Num and Bool expression types, I simply returned the expression as a value and for Raise and Unassigned, I raised an EvalException, as these are common between the Substitution model evaluator, Dynamically-scoped evaluator, and Lexically-scoped evaluator. For Unop, Binop, and Conditional, the evaluations all stayed the same in all three evaluators as well. For Var, Let, Letrec, and App, I abstracted out code that may have been similar between the three different evaluators. I also created an algebraic data type to represent the type of evaluator it may be and included a variable of this data type in the abstraction function to separate out code that was not common between the three different evaluators.

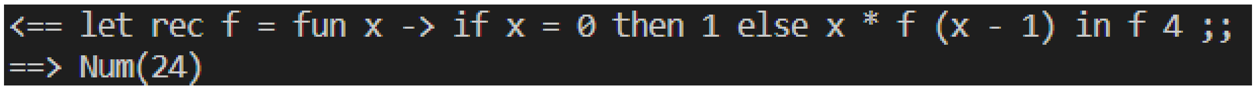
Below I have attached some images of the lexical evaluator working (Seen in the pdf):

Let expression with a nested Fun (function) and App (function application)



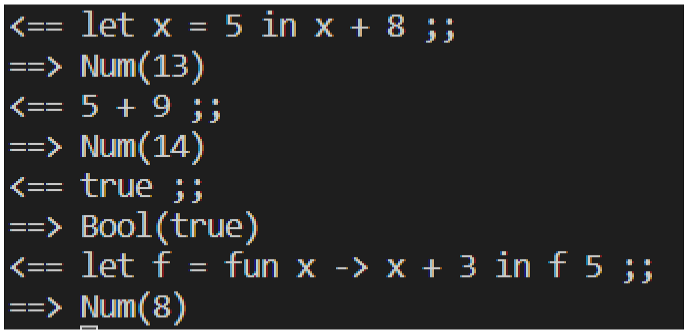
```
<== let x = 10 in
let f = fun y -> fun z -> z * (x + y) in
let y = 12 in
f 11 2 ;;
==> Num(42)
```

Letrec expression with a nested Fun (function)



```
<== let rec f = fun x -> if x = 0 then 1 else x * f (x - 1) in f 4 ;;
==> Num(24)
```

Expressions such as Binop and Bool (similar to Substitution and Dynamic)



```
<== let x = 5 in x + 8 ;;
==> Num(13)
<== 5 + 9 ;;
==> Num(14)
<== true ;;
==> Bool(true)
<== let f = fun x -> x + 3 in f 5 ;;
==> Num(8)
```