CS51 Final Project

Laura Ungar

April 26, 2016

For my final project extension, I decided to implement eval_l. It seemed like a natural choice, since OCaml itself is lexically scoped. This would have been much easier to do if I had not chosen to use a helper function with return type expression when writing my eval_d method. This choice made sense for eval_d, since eval_d never deals with Closures. Repeatedly wrapping expressions in Env.Val then pattern matching to get rid of the Env.Val would have made no sense in eval_d, and so I did not retroactively change my eval_d method. However, for eval_l, a helper function returning an expression would not work, since one has to return a closure for the Function case then use that closure in the function application case. This design difference is why I did not attempt to merge the two methods as was suggested in the writeup.

I started by copy-and-pasting my code from eval_d, then modifying it to not require a helper function. The main cases that needed to be changed to implement lexical scoping were

```
| Fun (v, exp1) -> exp
| App (exp1, exp2) -> (match eval_d_h exp1 env with
    | Fun (v, f) ->  eval_d_h f
      (Env.extend env v (ref (Env.Val (eval_d_h exp2 env))))
    | _ -> raise (EvalError ("invalid function application")))
```

from eval_d becoming

```
| Fun (v, exp1) -> Env.Closure (exp, env)
| App (exp1, exp2) -> (match eval_l exp1 env with
    | Env.Closure (Fun (v, f), env2) -> eval_l f
      (Env.extend env2 v (ref (eval_l exp2 env)))
    | _ -> raise (EvalError ("invalid function application")))
```

The eval_l version extends env2 (the environment passed up from the function) rather than env (the current environemnt) when evaluating function application. This is the critical difference that allows the program to "remember" the scope of the function to make miniml lexically scoped rather than dynamically scoped.

For reasons unknown to me and several TFs, who said my code for let rec in eval_d looked fine for both functions, I also had to change my let rec case slightly. The version copied from eval_d, which was transcribed into OCaml

from the lecture slides and works perfectly for eval_d, throws an EvalError in eval_l because at some point it tries to directly evaluate Unassigned. The code in eval_l is more methodical about setting up the recursive pointers and does not throw this error.

I use the test function test_scope () to demonstrate the differences between eval_l and eval_d. test_scope () runs on two cases involving variable shadowing where eval_l and eval_d should return different results.