CS51 - Final Project Writeup

Matt Jiang

May 2, 2018

1 Lexical Scope Extension

For my final project extension, I implemented "eval_l", an evaluator that uses environment semantics like "eval_d" but provides closures for functions so that they use the environment they were created in (AKA the lexical environment). This is different from "eval_d", in which the functions use the environment at the time of application (AKA the dynamic environment).

Ocaml is lexically scoped, so "eval_l" handles expressions similarly to the native Ocaml interpreter.

1.1 "eval_d" vs "eval_l

The two are different in how they handle the Fun and App match cases. When "eval_l" evaluates a Fun case, instead of just returning the same Val (Fun expression), it returns a closure of the Fun expression and the env at the time. App in "eval_l" evaluates the function definition in the closure environment, as opposed to the current environment in "eval_d". Left is d, right is l.

```
| Fun (y, e) -> Fun (y, e)
| Let (y, def, body) ->
            exp_of_val (eval_d body (extend env y (ref (eval_d def env))))
| Letrec (f, def, body) -> let unass = ref (Val Unassigned) in
                           let extendenv = extend env f unass in
                           let def' = eval_d def extendenv in
                           unass := def';
                           exp_of_val (eval_d body extendenv)
| Raise -> Raise
| Unassigned -> Unassigned
| App (p, q) ->  match eval_help p with
                 | Fun (y, e) -> exp_of_val
                                 (eval_d e (extend env y (ref (eval_d q env))))
                 | _ -> raise (EvalError "Invalid function application") in
Val (eval_help exp) ;;
```

```
| Fun (y, e) -> close (Fun (y, e)) env
| Let (y, def, body) -> eval_l body (extend env y (ref (eval_l def env)))
| Letrec (f, def, body) -> let unass = ref (Val Unassigned) in
                           let extendenv = extend env f unass in
                           let def' = eval_l def extendenv in
                           unass := def';
                           eval_l body extendenv
| Raise -> Val (Raise)
| Unassigned -> Val (Unassigned)
| App (p, q) -> match eval_l p env with
                | Closure (Fun (y, e), env_closure) ->
                     eval_l e (extend env_closure y (ref (eval_l q env)))
                | _ -> raise (EvalError "Invalid function application") ;;
```

I also had to change how extend works. Initially, it reassigned the reference of the id it was extending; however, this would also change instances inside closures, which was undesirable. I fixed it by making it find elements of the environment whose first element matched the id (so only within the general environment), then deleting those and replacing them with a new element containing the id and the new value ref.

1.2 an example

let x = 1 in let f = fun y -> x + y in let x = 2 in f 3 ;;

d will only evaluate f when it is being applied in the current environment. So when f is applied to 3 in the environment where x = 2, f evaluates to fun y -> 2 + y, and f 3 to 5.

l saves f in a closure the first time it is encountered, along with the environment at that time (i.e. x = 1). When it hits the next line "x = 2", this extends the general environment but does not affect the closure environment. So when f 3 is evaluated, f is evaluated in its closure environment to fun y -> 1 + y, and f 3 to 4.