Final Project: Writeup, Phillip Yu

Part 8: Lexical Scoping

Description

For my extension, I implemented the recommended extension: lexical scoping within an environment model.

Normally, because of the nature of the environment model – wherein a variable's value gets "replaced" each time it is defined – such a model is dynamically scoped. This means that a function depend on the values of its variables when it is called, not when it is defined (as in lexical scoping). Most current languages utilize lexical scoping.

In Part 7, I implemented a dynamically scoped environment model (see "eval_d").In Part 8, I extend this model to instead utilize lexical scoping (see "eval_l").

Implementation

The key difference in the implementations of these two models are in the match cases for function application and functions, as detailed below:

- Eval_d: In eval_d, functions are evaluated to themselves, since the "current" environment when the function is declared does not need to be packaged. See

example of such a match case:

```
| Fun (x, y) -> Env.Val (Fun (x, y))
```

Similarly, in function application in eval_d, the first argument to App(func, arg) should always evaluate to a function of type (Fun x, y). In the code below, x is then mapped to arg in env, **the environment that exists when this function is applied**.

```
| App (func, arg) ->
  (match eval_d func env with
  | Env.Val (Fun (x, y)) -> eval_d y (Env.extend env x (ref (eval_d arg env)))
  | _ -> raise (EvalError "non-function cannot be applied"))
```

- Eval_l: In eval_l, functions are evaluated to a closure of themselves **and** the environment that exists when said function is defined.

```
| Fun (x, y) -> Env.Closure (Fun (x, y), env)
```

In this manner, the lexical environment of the function can be stored. Thus, later when this function is applied, it can be done in the lexical environment rather than the dynamic one, as in:

```
| App (func, arg) ->
  (match eval_l func env with
  | Env.Closure (Fun (x, y), env') -> eval_l y (Env.extend env' x (ref (eval_l arg env
  | _ -> raise (EvalError "non-function cannot be applied"))
```

Notably, here the value x is extended in the environment env', not env, which means function application has become lexically scoped!

Examples

Consider the following code:

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

The evaluation of this depends on whether or not the interpreter is lexically or dynamically scoped; if lexical, this returns 4, while if dynamic, this returns 5.

As expected, when evaluated using eval_d, 5 is returned; when evaluated using eval_l, 4 is returned.

A final note

Because eval_d and eval_l share so much code in common other than the match cases for functions and function application, I extracted the code out into a separate function called eval_dl, which takes an additional argument n (the name of the function that it is being applied to).

If n is passed in as “d”, then eval_dl evaluates dynamically; if n is passed in as “l”, then eval_dl evaluates lexically.

```
let rec eval_dl (exp: expr) (env: Env.env) (n: string) : Env.value =
  match exp with
    # ... more code here ...
    | App (func, arg) ->
      (match n, eval_dl func env n with
      | "d", Env.Val (Fun (x, y)) -> eval_dl y (Env.extend env x (ref (eval_dl arg env
      | "l", Env.Closure (Fun (x, y), env') -> eval_dl y (Env.extend env' x (ref (eval
      | _ -> raise (EvalError "non-function cannot be applied"))
    | Fun (x, y) ->
        if n = "d"
        then Env.Val (Fun (x, y))
        else Env.Closure (Fun (x, y), env)
```