Final Project CS51

Nicole Chen

May 2022

1 Introduction

As part of this project, I implemented a couple of extensions. First, I added the lexical semantics environment to evaluate expressions. Second, I augmented the range of acceptable inputs to the parser and evaluator such that mutable state and imperative programming were now represented. Lastly, I included floats in the acceptable inputs to the parser as well as division.

2 Lexical Environment

To begin, I modified the scope of environmental semantics to include not just dynamical but also lexical. This is showcased in the `evaluation.ml` file in the the `eval_l` code. In the written code, I specifically used `and` statements to try to abstract the code as `eval_s`, `eval_d` and `eval_l` have similar implementations for `Num`, `Bool`, `Unop`, `Binop`, `Conditional`, `Unit`, `Raise`, and `Unassigned`. However, where `eval_l` mostly diverges is in function and function application. Rather than returning the function, in a lexical environment, evaluation should return a closure wrapping up the function in the current environment. This way, during function application, the right environment will be applied to the function. We see this in the match statement for functions, whereby rather than repeating code from the dynamic environment (`Val exp`), the lexical environment returns a closure from the `Env module` such that the match statement returns `Env.close exp env`. Of course, during function application then, rather than matching the evaluated function as a value (as the dynamic environment does), we match it for a closure and pattern match/evaluate within the closure's environment to get the correct expression.

Lastly, unlike dynamic environments which have the same implementation for `let` and `letrec`, the lexical environment must utilize the `Unassigned` expression to evaluate `letrec` statements, following the rule below in Figure 1.

Thus, translating such a rule in code, we write.

```
let temp = ref (Env.Val(Unassigned)) in
let env_x = Env.extend env var temp in
temp := (eval_l e1 env_x); eval_l e2 env_x
```

Figure 1: from http://book.cs51.io/pdfs/abstraction-19-environments.pdf

$$
\begin{array}{l}
E, S \vdash \texttt{let rec } x \texttt{ = } D \texttt{ in } B \Downarrow \\
\qquad \left| \begin{array}{l} E\{x \mapsto l\}, S\{l \mapsto \texttt{unassigned}\} \vdash D[x \mapsto !x] \Downarrow v_D, S' \\ E\{x \mapsto l\}, S'\{l \mapsto v_D\} \vdash B[x \mapsto !x] \Downarrow v_B, S'' \end{array} \right. \\
\Downarrow v_B, S'' \\
\qquad\qquad\qquad (R_{letrec})
\end{array}
$$

The variable temp references an Unassigned value such that when we extend the environment to include temp, we can reassign it to v_D which is the evaluated value of D (definition in the let statement). v_D is represented as `(eval_l e1 env_x)`. Next, we output the evaluated value of B in the reassigned environment through the statement `eval_l e2 env_x` .

3 Mutable State

The next extension that I implemented involved references. Specifically, I implemented evaluation rules for the assignment operator `:=`, the reference operator `ref`, and the dereference operator `!`. Note: I did not implement code for the sequence operator `;` as it was not specified in the suggested extensions. To develop these extensions, I first incorporated these symbols into the lexical analyzer, adding to the keyword and symbol table in the `miniml_lex.mll` file code:

```
("ref", REFERENCE)
(":=", ASSIGN);
("!", DEREFERENCE)
```

This code was added so that the lexical analyzer would recognize these different operators when typed into the `repl`. This was accompanied by additions to the parser. Ultimately, I implemented the dereference and reference operators as instances of `unop` and the assignment operator as an instance of `binop` for assignment priority reasons as well as to simplify the code.

Because mutable states can only be represented in the lexical environment, any attempt to utilize references in the dynamic environment semantics or substitution semantics results in an evaluation error. This is represented by the `eval_h` code

```
 | Deref, Unop(Ref, expr), 3 -> expr
 | Ref, expr, 3 -> Unop(Ref, expr)


 | Assign, Unop(Ref, Num p), Num q, 3 -> Unit
 | Assign, Unop(Ref, Bool p), Bool q, 3 -> Unit
```

whereby the 3 represents the evaluation type (1 being substitution semantics, 2 being semantics in the dynamic environment, and 3 being semantics in the lexical environment). Thus, only expressions that are being evaluated in the lexical environment can use mutable operators.

Within the `eval_l` function, we see that there is one specific area that has changed: the matching of the `let` expression. As the extension in the textbook emphasized "Since the environment is already mutable, you can even implement this extension without implementing stores and modifying the type of the eval function," I decided to implement mutable types without changing the eval_l typing and without adding stores. As such, the let construct needed to be changed. Note: I understand in the textbook that the let construct is never changed, but because the textbook utilizes stores, that is possible. In the instance that the sequence operator (;) is not used and the fact that stores also aren't, it is necessary to alter the `let` expression.

More specifically, the code goes that:

```
| Let (var, e1, e2) -> (match e1 with
    | Binop (Assign, p, q) -> let temp = ref (eval_l p env) in
    (match !temp, p with
        | Val(Unop(Ref, _)), Var v -> let env_x = Env.extend env v
                                    temp in temp := eval_l
                                    (Unop (Ref, q)) env_x;
                                    eval_l e2 (Env.extend env_x var
                                    (ref (Env.Val(Unit))))
        | Val(Unop(Ref, _)), _ -> Val(Unit)
        | _ -> raise (EvalError "Binop"))
    | _ -> eval_l e2 (Env.extend env var (ref (eval_l e1 env))))
```

Walking through my logic here, for assignment, only when the expression on the left is a reference variable will the assignment actually store something in that variable. As such, we pattern match p with expression `Var v`, and then pattern match `temp`, which represents the evaluated value of p in an empty environment, to `Val(Unop(Ref, _)`. This is to ensure that the assigned expression is first a reference and second a variable. If it is not a variable, the assignment returns a `Unit`. If it is not a reference, the assignment returns an error.

If p is both a reference and a variable, we extend the value of evaluated p (represented as `temp` in the code) into the environment `env_x` and then evaluate q as a reference in that environment, extending `env_x` to the newly evaluated reference value of q. Lastly, we utilize `env_x` to evaluate the second expression within the `let` statement, but because the assignment operator returns a `Unit`, the extended environment matches the let variable `var` to a `Unit` value. The tricky part here was making sure that I didn't assign the let variable `var` to the reassigned value of the reference `Var v`.

The main reason why I implemented the assignment match case in the `let` match case was because since there is no sequence operator (;), the only way assignment can be utilized and tested is through let statements such that

```
let y = ref 3 in let z = (y := 5) in !y;;
```

which evaluates to `Num(5)`. If we were to return z instead such that

```
let y = ref 3 in let z = (y := 5) in z;;
```

we would get `Unit`. Note: the `Unit` expression is only really applicable to references in my implementation. I did not add any extensions towards function application with Units.

4 Floats and Division

For my last extension, I implemented Floats and Divisions. Similar to how we implemented integers within the lexical analyzer and parser, I added floats to the lexical analyzer through utilizing ocaml's `float_of_string` function as shown below and adding the respective token and precedence to the parser whereby `TIMES` and `DIVIDE` had the same precedence.

```
Lexical Analyzer:

| digit+"."digit+ as ifloat
        { let float = float_of_string ifloat in
          FLOAT float
        }

Parser:

%token <float> FLOAT

%left TIMES DIVIDE

| FLOAT                 { Float $1 }
```

It must be noted though that to increase readability and simplicity of the code, rather than adding different float operators such as `PLUSDOT` and `TIMESDOT` (represented as +. and *.), floats can be computed with integer operators such (+, -, *, /) and negated with integer operators such as `-`. As such, in the evaluation functions (specifically in `eval_h` because all evaluations have the same implementation for floats), the binary and unary operators remain the same for floats and nums (integers); however, each must be matched to the same type, as shown below:

```
| Unop (u, e) -> (match u, eval\_h e env eval\_type, eval\_type with
                   | Negate, Num num, \_ -> Num(~-num)
                   | Negate, Float f, \_ -> Float (~-.f)
                   | Deref, Unop(Ref, expr), 3 -> expr
```

```
                          | Ref, expr, 3 -> Unop(Ref, expr)
                          | \_ -> raise (EvalError "Unop"))

  | Binop (b, e1, e2) -> (match b, (eval\_h e1 env eval\_type),
                                    (eval\_h e2 env eval\_type), eval\_type with
                                | Equals, Num p, Num q, \_ -> Bool (p = q)
                                | Equals, Bool p, Bool q, \_ -> Bool (p = q)
                                | LessThan, Num p, Num q, \_ -> Bool (p < q)
                                | LessThan, Bool p, Bool q, \_ -> Bool (p < q)
                                | Plus, Num p, Num q, \_ -> Num (p + q)
                                | Plus, Float p, Float q, \_ -> Float (p +. q)
                                | Minus, Num p, Num q, \_ -> Num (p - q)
                                | Minus, Float p, Float q, \_ -> Float (p -. q)
                                | Times, Num p, Num q, \_ -> Num (p * q)
                                | Times, Float p, Float q, \_ -> Float (p *. q)
                                | Divide, Num p, Num q, \_ -> Num (p / q)
                                | Divide, Float p, Float q, \_ -> Float (p /. q)
                                | Assign, Unop(Ref, Float \_), Float \_, 3 -> Unit
                                | Assign, Unop(Ref, Num \_), Num \_, 3 -> Unit
                                | Assign, Unop(Ref, Bool \_), Bool \_, 3 -> Unit
                                | \_ -> raise (EvalError "Binop"))
```

It also must be noted that because of the way I specified in the parser and lexical analyzer, `12.` is not a valid substitute for `12.0`. Lastly, I implemented the divide operator for both floats and nums (integers). Adding the divide operator simply involved adding to the lexical analyzer and parser to recognize the divide symbol and then augmenting the evaluation function (in the `eval_h` function) so that division between two nums and two floats gave `p / q` or `p /.q`.

5 Conclusion

All in all, this project was extremely fun and I really enjoyed the exploratory aspect of it!