CS51 Final Project Writeup

Gardenia Liu

April 2024

For my CS51 final project, I extended MiniML to include a few additional features. MiniML already allowed for the substitution and dynamic models for evaluation expressions, so I added a lexical model as well. Furthermore, I added the float type and corresponding binary operations. I will explain these extensions in detail and how I implemented and tested them.

1 Lexical Scoping

For dynamically scoped environments, a function's evaluation depends on when the function is called. However, for lexically scoped environments, the function's evaluation depends on when the function is defined. For example, let's take a look at the line below:

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

In a dynamically scoped environment, the above would evaluate to 3. This is because the line x = 2 is applied to the function. Thus, when we apply f to 1, we get 2 + 1 = 3. In the lexically scoped environment (which is what OCaml uses), it would evaluate to 2. This is because the first line with x = 1 is what the definition of f relies on. In other words, when we apply f to 1, we get 1 + 1 = 2. By creating a closure, we can save the environment where the function is defined, and use that environment whenever the function is called.

To do this, I added a Lex model to my model type so that I could add match statements for lexical models within my evaluation helper function. The only parts of this function I changed were Fun, App, and Letrec.

```
type model =
  | Sub
  | Dyn
  | Lex
```

Fun: For lexical models, instead of just evaluating a function expression to itself, I closed the expression with the environment when the function is defined using the close function:

```
(* the close function *)
let close (exp : expr) (env : env) : value = Closure (exp, env)

(* the modified match statement *)
 | Fun _ -> if m = Lex then close exp env else Val exp
```

Letrec: For the recursive Letrec portion of the lexical evaluation, I followed the steps in the readme. I assigned Unassigned to x first so that evaluation of the definition didn't depend on x's value. Then, I updated x to be the evaluated value and evaluated the body subexpression.

```
| Lex ->
    let x = ref (Val Unassigned)
    in let env_x = extend env v x
    in let v_D = eval_helper Lex e1 env_x
    in x := v_D; eval_helper Lex e2 env_x)
```

App: For the application portion of the lexical evaluation, I matched the closure and evaluated the function with the closure's environment.

```
| Closure (Fun (v, e), env_old) ->
    let val_q = ev e2
    in let ext = extend env_old v (ref val_q)
    in eval_helper m e ext
```

2 Floats

After adding lexical environment scoping, I added floats to MiniML. I made changes within expr.ml and expr.mli by adding an additional match case to the expr type:

```
type expr =
  | Var of varied                        (* variables *)
  | Num of int                           (* integers *)
  | Float of float                       (* floats *)
  ...
```

Then, I had to change the miniml_lex.mll and miniml_parse.mly files to allow MiniML to recognize Floats. In miniml_lex.mll, I added the following code to create a regular expression and tokenized them.

```
let float = digit+ ('.' digit*)? (['e' 'E'] ['+' '-']? digit+)?

rule token = parse
  | float as fnum
        {
          let num = float_of_string fnum in
          FLOAT num
        }
```

After parsing, I also changed my free_vars and subst methods to account for Floats.

3 Other Binary Operators

When implementing Floats in MiniML, I had to decide whether or not mixing between types would be allowed. To make MiniML similar to OCaml, I chose to make it strongly typed. I decided to differentiate between binary operators for floats and nums. To do so, I introduced three new types of binops which only operate on floats.

```
type binop =
  | Plus
  | Minus
  | Times
  | FPlus   (* new *)
  | FMinus  (* new *)
  | FTimes  (* new *)
  ...
```

FPlus, FMinus, and FTimes work the same as Plus, Minus, and Times, but they only allow for these operations between floats. Therefore, within my binopeval helper function, I had to account for these cases:

```
  | FPlus, Float x1, Float x2 -> Float (x1 +. x2)
  | FMinus, Float x1, Float x2 -> Float (x1 -. x2)
  | FTimes, Float x1, Float x2 -> Float (x1 *. x2)
```

If anything had intertype operations, I raised an EvalError. I also added another binary GreaterThan operator, which was handled similarly to the LessThan and Equals operators.

4 Testing

After implementing these extensions, I also thoroughly tested my code within test.ml. One particular case I paid extra attention to was the evaluation differences between lexical and dynamic models. With lexical and dynamic models, the evaluations differ with a test case of this structure, which was introduced in part 1 of this writeup:

```
(* let x = 5 in let f = fun y -> x + y in let x = 10 in f 0 *)
let expr11 =  Let("x", Num(5), Let("f", Fun("y", Binop(Plus, Var("x"),
              Var("y"))), Let("x", Num(10), App(Var("f"), Num(0))))) ;;
```

Thus, I made sure that both models were handling this case correctly, with dynamic evaluation returning 10 and lexical returning 5. Furthermore, I also tested my other extensions, such as float binary operators.

```
let expr12 = Unop (Negate, Float(-1.)) ;;
let expr13 = Binop (FPlus, Float 1., Float 2.) ;;
let expr14 = Binop (FMinus, Float 3., Float 2.) ;;
let expr15 = Binop (FTimes, Float 2., Float 3.) ;;
let expr16 = Binop (Equals, Float 2., Float 2.) ;;
let expr17 = Binop (LessThan, Float 2., Float 3.) ;;
let expr18 = Binop (GreaterThan, Float 2., Float 3.) ;;
```

5 Conclusion & Next Steps

Given the time limit on the project, there were many extensions I was unable to complete. If I had more time, I would have added syntactic sugar and allowed for MiniML to recognize curried functions by changing how parsing is implemented. Thank you for reading my writeup!