CS51: Final Project Writeup

Maggie Rufaro Mano

Wednesday 4 May 2021

I extended my final project for CS51 in a number of ways that are listed below:

1. Adding a Lexically Scoped Evaluator
2. Adding an additional atomic type, `float`, with and corresponding literals and operators
3. Adding additional Binary operators: **Divide** and **GreaterThan**

1. LEXICALLY SCOPED EVALUATOR

In order to implement a lexically scoped environment I modified the dynamic evaluator, `eval_d`. In section 21.4.2 of the textbook, an example is offered that results in different solutions when using `eval_d` in comparison to a lexically scoped evaluator. The example is:

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

When using `eval_d`, since the evaluation is dynamic, we would use the most recent definition of x which would be `x = 2`. This would result in the answer of 5. However, in a lexically scoped environment, we would use the x that is defined initially which would be `x = 1`. This would result in the answer of 4.

Another example we can look at is:

```
 let x = 5 in
let f = fun y -> x + y in
let x = 7 in
f x ;;
```

In a dynamic environment this function evaluates to 16 because the most recent definition of x is `x = 7`. This means at the time the function is called `x = 7` and the function would evaluate to **7 + 7** which is 14. In a lexical environment, a closure preserves the definition as: `{x -> 5} fun y -> x + y`, thus always evaluating the function as: `fun y -> 5 + y`. Since 7 is then passed in as an argument to the function, the evaluation would result in **5 + 7** which is 12.

So how would we modify `eval_d` to create a lexically scoped evaluator, `eval_l`?

The only aspects that needed to change from `eval_d` were:

- **Fun**
  - I modified the evaluation of a function, so that it returns a closure containing the function itself as well as the current environment. So instead of returning an `Env.Val`, it would return an `Env.Closure` containg a tuple of both the expression **and** the environment.
- **App**
  - This stream of conscoiusness is continued in this modification. When evaluating the first expression we return a `Env.Closure` that can only store a Fun. Then we extend the environment and evaluate accordingly by mapping the variable with the evaluation of what's passed to the function.

Does it work?

Let's test out the example using both `eval_d` and `eval_l`.

`eval_d`:

```
 <== let x = 1 in
let f = fun y -> x + y in
let x = 2 in f 3 ;;
==> 5
```

`eval_l`:

```
 <== let x = 1 in
let f = fun y -> x + y in
let x = 2 in f 3 ;;
==> 4
```

In order to abstract away commonalities between `eval_d` and `eval_l`, I implemented a helper evaluator: `eval_dl_helper`. The function has type:

```
Expr.expr -> Env.env -> (Expr.expr -> Env.env -> Env.value) -> Env.value
```

We can see that unlike the other evaluators, `eval_dl_helper` takes an evaluator as an argument as well. This is because for the cases that are different for `eval_d` and `eval_l`, we call the `eval` argument so that the appropriate result is given in the different circumstances.

2. FLOATS

In addiion to adding a lexically scoped evaluator, I added an additional atomic type, `float`. In order to do this, I had to:

- Add **Float** type to *expr.mli*
- Make relevant additions to *expr.ml* with **Float**. Modify the functions in *expr.ml* to suit the Float case.
- Modify *evaluate.ml* so that the evaluators evaluate the **Float** case as well.
- Modify miniml_parse.mly
  - I added an appropriate float token:

  ```
  %token <float> FLOAT
  ```

  as well as a grammar definition:

  ```
  | FLOAT                 { Float $1 }
  ```

- Modify miniml_lex.mll
  - I added a lexing rule of:

  ```
  | digit+ '.' digit* as fnum
     { let fl = float_of_string fnum in
       FLOAT fl
     }
  ```

  This is very similar to the lexing rule for `int`, just modified with the appropriate "." for floats.

At first I added additional unary and binary operators that corresponded to the `float` type, but I realized that it wasn't necessary. Instead I just made the unary and binary operators accessible to the `float` type. This makes it easier for users as they wouldn't have to use operators such as: `+.`, `~-.`, `*.`, etc, when evaluating with floats.

3. DIVIDE AND GREATERTHAN

I additionally added **Divide** and **GreaterThan** binary operators to binop to extend my MiniMl Project. **Divide** divides the first expression given by the second expression. **GreaterThan** returns **Bool** true if the first expression given is bigger than the second expression and **Bool** false if the opposite is true. These additions provide a more extensive and a wider variety for users. In order to do this, I had to:

- Add **Divide** and **GreaterThan** binary operators to *expr.mli*
- Make relevant additions to *expr.ml* with **Divide** and **GreaterThan**. Modify the functions in *expr.ml* to suit these extra binops.
- Modify *evaluate.ml* so that the binop helper evaluator evaluates the **Divide** and **GreaterThan** cases as well.
- Modify miniml_parse.mly
  - I added the appropriate divide and greater than tokens as follows:

  ```
   %token TIMES DIVIDE
  %token LESSTHAN GREATERTHAN EQUALS
  ```

  as well as the appropriate associations:

  ```
   %left LESSTHAN GREATERTHAN EQUALS
  ...
  %left TIMES DIVIDE
  ```

  as well as the appropriate grammar definitions:

  ```
   | exp DIVIDE exp        { Binop(Divide, $1, $3) }
  | exp GREATERTHAN exp   { Binop(GreaterThan, $1, $3) }
  ```

- Modify miniml_lex.mll
  - I added the following to the sym_table that is a referance for parsing:

```
 (">", GREATERTHAN);
("/", DIVIDE);
```

These various additions make my Miniml project extended and more user-friendly.