Implementing MiniML – CS51 Final Project Writeup

Lan Zhang

May 2019

Overview

In this project, I implemented MiniML, a small subset of an OCaml-like language, consisting of only a fraction of the constructs and types associated with OCaml, and does not support type inference. I implemented three different MiniML interpreters that evaluate expressions using different semantics; the first MiniML interpreter uses the substitution model, the second uses the dynamic scoped environment model, and the third, an added extension, implements the lexical scoped environment model.

Extensions

I. Lexical Scoped Environment Model

As part of my extension of the MiniML language, I implemented an interpreter that utilizes lexically scoped environment semantics. The main difference between lexically scoped environment semantics and dynamically scoped environment semantics is that the values of the variables are determined by the lexical structure of the program; unlike in dynamic scoping where functions are evaluated using the environment existent at the time of application, in lexical scoping functions are evaluated in the environment created at the time of function definition.

The two main differences between dynamic environment semantics and lexical environment semantics lies in the evaluation of a function and the evaluation of a function application. So, in accordance with the final project instructions, I first made a copy of my `eval_d` function and modified the code so that the evaluation of a function returned a closure containing the function itself and the environment existent at the time of its definition.

```
| Fun (_v, _e) -> Closure (exp, env)
```

I also modified the code so that the evaluation of a function application evaluated the body of the function in the environment from the corresponding closure.

```
| App (e1, e2) -> match eval_l e1 env with
                  | Closure (Fun (v, e), env1) -> let v_e2 = eval_l e2 env in
                                                  let new_env =
                                                    extend env1 v (ref v_e2) in
                                                  eval_l e new_env
                  | _ -> raise (EvalError "Invalid Application")
```

Upon making these two modifications, I recognized that because the function and application match cases for an expression were the only differences between the `eval_d` and `eval_l` functions, I created the function `eval_both` that employs either dynamic or lexical semantics evaluation depending on the boolean value of the input `dyn_eval`. If the value of this boolean flag is true, then evaluation associated with dynamic environment semantics is employed, and if the value of this boolean flag is false, then evaluation associated with lexical environment semantics is employed.

The value of the boolean flag `dyn_eval` is critical in the function and application match cases for the inputted expression argument of `eval_both`. For the function match case, if `dyn_eval` is set to true, the function would evaluate to the expression itself as per dynamic environment semantics rules, and if `dyn_eval` is set to false, the function would evaluate to a closure of the expression and the current environment at the time of function definition, as per lexical environment semantics rules.

With the application match case, I abstracted the code associated with this match case from `eval_d` and `eval_l` into two separate helper functions, `app_d` and `app_l`. If the boolean flag `dyn_eval` is set to true, `app_d` is called

and dynamic environment semantics evaluation rules are then employed, the function evaluated in the environment existent at the time of application. On the other hand, if `dyn_eval` is set to false, `app_l` is called and lexical environment semantics evaluation rules are employed, the function evaluated in the environment existent in the corresponding closure that was created at the time of function definition.

To test my lexical scoped environment model extension, I used two expressions that would evaluate to different values under dynamic environment semantics vs. lexical environment semantics rules:

The first expression was provided in the project description:

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

Under dynamic environment semantics, the above expression evaluates to 5, as `x = 2` at the time of function application. On the other hand, under lexical environment semantics, the expression evalues to 4, as `x = 1` at the time of function definition.

Similarly for the other test case,

```
let x = 5 in
let f = fun y -> 2 * x * y in
let x = 3 in
f 4 ;;
```

the expression evaluates to 24 under dynamic environment semantics, and 40 under lexical environment semantics.

II. Additional Operators

As another portion of my extension, I added both the division and greater than operators for the existing atomic types of integers and bools. To do so, I first modified relevant functions in `expr.ml` and `evaluation.ml`.

Because the division and greater than operators are both binary operators, I first expanded the binop type to include `Divide` and `GreaterThan`, making corresponding changes to the `expr.mli` file as well to modify the signature of the `Expr` module. I then modified the `expr_to_abstract_string`

and `expr_to_concrete_string` functions to return appropriate string representations of these additional operators.

Turning to the `evaluation.ml` file, I modified the helper function `binopeval`, providing evaluation rules for the division operator and the greater than operator applied to two integers, raising an `Eval_Error` exception if applied to arguments of other types. For the `Divide` operator, because division by zero is undefined, I also raised an exception if the second argument (`arg2`) in `Binop (Divide, arg1, arg2)` is the integer zero.

I then extended the provided MiniML parser in the file the `miniml_parse.mly` to parse symbols associated with these new operators. I expanded the token that previously had the constructor `TIMES` to include the constructor corresponding to division as well (`DIVIDE`), and similarly expanded the token that had the constructors `LESSTHAN EQUALS` to contain the constructor corresponding to the `GREATERTHAN` operator:

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

I also expanded the grammer in `miniml_parse.mly` to include functionality for the division and greater than operators:

```
%%
...
expnoapp: ...
        | exp DIVIDE exp        { Binop(Divide, $1, $3) }
        | exp GREATERTHAN exp   { Binop(GreaterThan, $1, $3) }
        ...
;

%%
```

Next, I modified the `miniml_lex.mll` file to add the division symbol `/` and greater than symbol `>` to the symbol hashtable and associated these symbols with the named constructors I had just defined in `miniml_parse.mly`:

```
  let sym_table =
    create_hashtable 8 [
                       ...
                       (">", GREATERTHAN);
```

```
    ("/", DIVIDE);
]
```

I then made one final edit to this lexical analyzer for MiniML, adding the symbols `>` and `/` to the defined symbol character set.

```
let sym = ['(' ')'] | (['+' '-' '*' '.' '=' '~' ';' '<' '>' '/']+)
```

Upon making these changes, I ran `./miniml.byte` and was able to evaluate expressions containing the division and greater than operators in the MiniML REPL:

```
#./miniml.byte
Entering ./miniml.byte...
<== 3 > 4 ;;
==> false
<== 3 / 4 ;;
==> 0
<== 6 / 2 ;;
==> 3
```

III. Floats

Another extension I implemented consisted of adding floats to the MiniML language. To do so, I first modified `expr.ml`, expanding the `unop` type definition to include a new unary operator, negation of floats, naming it `Negate_f`. I also expanded the `binop` type definition to include the corresponding binary operators on floats that my MiniML implementation at the time supported for integers, naming them `Plus_f` for addition of floats, `Minus_f` for subtraction of floats, `Times_f` for multiplication of floats, and `Divide_f` for division of floats:

```
type unop =
  ...
  | Negate_f
;;

type binop =
  ...
  | Plus_f
  | Minus_f
```

```
  | Times_f
  | Divide_f
;;
```

Next, I expanded the type definition of `expr` to include `Float of float`. Upon doing so, I had to update the match cases of many other functions that pattern matched a variable of type `expr`.

In `expr.ml`, the functions that I altered to support the new float type were `free_vars`, `subst`, `exp_to_concrete_string`, and `exp_to_abstract_string`. For `free_vars`, because a float is a constant and therefore has no free variables, I returned an empty set for the float match case. Similarly for `subst`, because a float is not a variable, I simply returned the original float in the float match case. For `exp_to_concrete_string`, I simply called the function `float_of_string` from the Pervasives module and applied it to the actual float (`f` in `Float(f)`, extracted through pattern-matching). Finally, for `exp_to_abstract_string`, I returned essentially the same thing as I did for the float match case in `exp_to_concrete_string`, but wrapped the result with the string `"Float( )"`.

In `evaluation.ml`, I first altered the helper functions `unopeval` and `binopeval` to support the new float operations I had defined earlier in `expr.ml`. I defined the evaluation of float operations to be analagous to the existing int operations I had already implemented, instead using OCaml float operators (`~-., +., -., *., /.`). Then, I altered `eval_s` and `eval_both` to support the `Float (_)` pattern-match case for expressions. In all evaluation rules for substitution, lexical environment, and dynamic environment semantics, a float simply evaluates to itself, so I implemented these minor changes in `eval_s` and `eval_both`.

Finally, I extended the provided MiniML parser to parse floats. I first altered the `miniml_parse.mly` file, extending the existing token declarations to include support for float operations. I added the constructors `NEG_F, PLUS_F`, `MINUS_F`, and `TIMES_F` to the token declarations containing their integer operator counterparts. I also added a token declaration for floats, creating a token with a constructor that has the attributes of type float:

```
...
%token NEG NEG_F
%token PLUS PLUS_F MINUS MINUS_F
```

```
%token TIMES TIMES_F DIVIDE DIVIDE_F
...
%token <float> FLOAT
```

I also expanded the grammer in `miniml_parse.mly` to include functionality for these five new float-specific operators:

```
%%
...
expnoapp:
        ...
        | exp PLUS_F exp        { Binop(Plus_f, $1, $3)
        | exp MINUS_F exp       { Binop(Minus_f, $1, $3) }
        | exp TIMES_F exp       { Binop(Times_f, $1, $3) }
        | exp DIVIDE_F exp      { Binop(Divide_f, $1, $3) }
        | NEG_F exp             { Unop(Negate_f, $2) }
;

%%
```

Next, I modified the `miniml_lex.mll` file to add the symbols associated with these five new float-specific operators to the existing symbol hashtable, and associated these symbols with the named constructors I had just defined in `miniml_parse.mly`.

```
  let sym_table =
    create_hashtable 8 [
                       ...
                       ("~-.", NEG_F);
                       ("+.", PLUS_F);
                       ("-.", MINUS_F);
                       ("*.", TIMES_F);
                       ("/.", DIVIDE_F);
                     ]
}
```

I then referenced the everyday OCaml resource cited at the end of the writeup to make the final changes to the lexical analyzer for MiniML to parse floats and the corresponding operations of this added type.

I first defined the float character set:

```
let digit = ['0'-'9']
let frac = "." digit*
```

```
let float = digit* frac?
...
```

I then modified the token rule to support floats:

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

Upon making these changes, I ran `./miniml.byte` and was able to evaluate expressions containing floats in the MiniML REPL:

```
#./miniml.byte
Entering ./miniml.byte...
<== 3. > 4. ;;
==> false
<== 3. / 4. ;;
==> 0.75
<== 3. +. 4. ;;
==> 7. ;;
<== let rec f = fun x -> if x = 0. then 1. else x *. f (x -. 1.) in f 4. ;;
==> 24.
```

Resources

Minsky, Yaron, et al. "Chapter 16. Parsing with OCamllex and Menhir." Chapter 16. Parsing with OCamllex and Menhir / Real World OCaml, v1.realworldocaml.org/v1/en/html/parsing-with-ocamllex-and-menhir.html.