CS51 Final Project: MiniML

Sid Bharthulwar

May 3rd, 2022

1 Introduction

In this project, I completed an implementation of an OCaml subset language, called MiniML. This language is Turing-complete, which by the Church-Turing thesis implies that any performable computation can be completed within the language. However, there are some constraints on the language due to its simple nature and lack of advanced data types by default (lists, records, arrays, etc). Additionally, because MiniML is in the form of an interpreter for expressions written in the OCaml syntax itself, this implementation can also be considered a metacircular interpreter. MiniML supports basic computation and recursive functions, but does not have type inference, custom data types, classes and objects, or functors. Additionally, I implemented three types of semantics in MiniML: the subtitution model, dynamically scoped environmental model, and lexically scoped environmental model.

2 Extensions

2.1 Lexical Scope Extension

The first major extension that I added to MiniML was the addition of the `eval_l` function, which evaluates expressions with lexically scoped environment semantics. Because OCaml is lexically scoped, this particular evaluation method handles expressions more similarly to OCaml than either `eval_d` or `eval_s`. In order to facilitate both `eval_l` and `eval_d` methods, I created a helper function based on my original `eval_d` code that would remove code redundancy between the two methods.

```
let eval_d_l (exp : expr) (env : Env.env) (isdynam : bool) : Env.value =
  let rec helper (exp : expr) (env : Env.env) : Env.value =
  match exp with
  | Var v -> Env.lookup env v
  | Num _  | Float _ | Bool _ | Raise | Unassigned -> Env.Val exp
  | Unop (xyz, e) -> (match helper e env with
                    | Env.Val e -> Env.Val (evalunop xyz e)
                    | _ -> raise (EvalError " invalid unary operation applied"))
  | Binop (xyz, ab, bc) -> (match helper ab env, helper bc env with
                          | Env.Val ab, Env.Val bc -> Env.Val (evalbinop xyz ab bc)
                          | _, _ -> raise (EvalError " invalid binary operation applied "))
  | Conditional (ab, bc, cd) -> (match helper ab env with
                               | Env.Val (Bool b) -> if b then helper bc env
```

```
                                        else helper cd env
                                | _ -> raise (EvalError " invalid conditional applied "))
  | Fun _ -> if isdynam then Env.Val exp else Env.close exp env
  | Let (v, ab, bc) -> helper (App (Fun (v, bc), ab)) env
  | Letrec (v, ab, bc) ->
                          let a = ref (Env.Val Unassigned) in
                          let new_env = Env.extend env v a in
                          let a_1 = ref (helper ab new_env) in
                          (a := !a_1; helper bc (Env.extend env v a))
  | App (ab, bc) ->
      match helper ab env with
      | Env.Val (Fun (v, e)) ->
        if isdynam then helper e (Env.extend env v (ref (helper bc env)))
        else raise EvalException
      | Closure (Fun (v, e), en) ->
        if not isdynam then helper e (Env.extend en v (ref (helper bc env)))
        else raise EvalException
      | _ -> raise (EvalError " application error ")

  in
  helper exp env ;;
```

This function follows the same approach as my original `eval_d` function, but instead includes a boolean argument that determines whether the evaluation should follow dynamically scoped environment semantics or lexically scoped environment semantics. If it is following dynamically scoped environment semantics, if the expression had a function application, it is evaluated in the most recent environment. Conversely, if it is following lexically scoped environment semantics, if the expression had a function application, it iw evaluated with the environment that it was originally defined in (like OCaml currently works). The difference is clearly visible with an example: `let x = 1 in let f = fun y -> x + y in let x = 2 in f 3 ;;` evaluates to `5` in the dynamically scoped environment, but evaluates to `4` with the lexically scoped environment model. Additionally, when passed into an OCaml interpreter, the result is also `4`, which is consistent with the type of environment method that OCaml uses.

2.2 Floats

2.2.1 Introduction

In order to make MiniML more useful for standard computations, I deemed the inclusion of the float data type as necessary. Just as the integer is defined in the parser, `eval_parse.mly`, I added the line `%token <float> FLOAT` and added the grammar rule in `expnoapp: | FLOAT {Float $1 }`. Within the lexer, I added the following rule for floats:

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

2.2.2 Trigonometric Unary Operators

Similar to how the float syntax was added to MiniML, three unary operators that work on both floats and integers were also added: the trigonometric functions sine, cosine, and tangent. Both the lexer and parser were modified to recognize these tokens as valid syntax, and the evaluation rules were added to `evaluation.ml`. Specifically, the helper function that I created for evaluating unary operators was modified as so:

```
let evalunop (xyz : unop) (e : expr) : expr =
  match xyz, e with
  | Negate, Num x -> Num (~-x)
  | Negate, Float x -> Float (~-. x)
  | Sin, Num x -> Float (sin (float_of_int x))
  | Sin, Float x -> Float (sin x)
  | Cos, Num x -> Float (cos (float_of_int x))
  | Cos, Float x -> Float (cos x)
  | Tan, Num x -> Float (tan (float_of_int x))
  | Tan, Float x -> Float (tan x)
  | _, _ -> raise (EvalError " invalid unary operator ") ;;
```

To make these unary operators easier to use within the language, I added the ability for integers to be passed in as arguments as well as floats, so there is no need to cast integers to floats beforehand (in fact, this functionality is not available with the current version of MiniML).

2.2.3 Interoperability of Floats and Integers

For my final float-based extension for my implementation of MiniML, I thought that I would implement one of my favorite features of Python. I took advantage of the chance to modify OCaml's core behavior by adding interoperability of integers and floats with binary operators. In standard OCaml, executing a command such as `3 + 4.2 ;;` results in an error, because the + operator can strictly be called with integer arguments. However, instead utilizing the +. operator results in the same issue, because both arguments have to be floats. In order to solve this, I eliminated the binary arithmetic operators with the '.' suffix, and introduced arithmetic interoperability between types to MiniML. My implementation is similar to default semantic rules in Python, where common binary operators can be performed on both integers and floats, and the outputs are of type float.

However, the pattern matching for handling every case of integer/float binary operation combinations is very arduous. For example, to handle every combination of integer and float inputs to the addition binary operation, we need to match the case in which both inputs are integers, when the first input is a float, when the second input is a float, and when both inputs are floats. As the number of binary operators supporting data type interoperability increases, so does the length of the code required to handle all the combinations. Instead, I developed a helper function to handle these combinations. See below:

```
let float_int_helper (f_int : int -> int -> int)
  (f_float : float -> float -> float)
  (x : 'a) (y : 'b) : Expr.expr =
  match x, y with
  | Num a, Num b -> Num (f_int a b)
  | Num a, Float b -> Float(f_float (float_of_int a) b)
  | Float a, Num b -> Float(f_float a (float_of_int b))
```

```
  | Float a, Float b -> Float(f_float a b)
  | _, _ -> raise (EvalError "invalid binary operator") ;;
```

The function takes in the binary operator as a function in its integer form, the same binary operator in its float form, and the two arguments as ambiguous types. Then, the two arguments are matched to the four possible cases, and the correct function is invoked on the two arguments after they have both been casted to floats. Note that, if both arguments are integers, the output is an integer. Otherwise, the output is a float (just like Python). The helper function is called by another helper function, which handles all binary operations during evaluation. The implementation of this function is shown below.

```
let evalbinop (b : binop) (ab : expr) (bc : expr) : expr =
  match b with
  | Plus -> float_int_helper (+) (+.) ab bc
  | Minus -> float_int_helper (-) (-.) ab bc
  | Times -> float_int_helper ( * ) ( *.) ab bc
  | _ ->
    match b, ab, bc with
    | Equals, Num x, Num y -> Bool (x = y)
    | Equals, Float x, Float y -> Bool (x = y)
    | Equals, Float x, Num y -> Bool (x = (float_of_int y))
    | Equals, Num x, Float y -> Bool ((float_of_int x) = y)
    | LessThan, Num x, Num y -> Bool (x < y)
    | LessThan, Float x, Float y -> Bool (x < y)
    | LessThan, Float x, Num y -> Bool (x < (float_of_int y))
    | LessThan, Num x, Float y -> Bool ((float_of_int x) < y)
    | Equals, Bool x, Bool y -> Bool (x && y)
    | _, _, _ -> raise (EvalError " invalid binary operator ") ;;
```

Finally, the behavior of the MiniML language with this new feature is given below.

```
<== 3 + 4 ;;
==> 7
<== 4.2 + 3.4 ;;
==> 7.6
<== 3 + 9.8 ;;
==> 12.8
```