MiniML Language Interpreter

Austin Hwang

May 2, 2018

1 Introduction

The MiniML language is a Turing-complete or computationally universal language, meaning that it can be used to simulate any Turing machine. This interpreter can interpret expressions written in OCaml; thus, it is called a metacircular interpeter. It supports basic computation and recursive functions, but does not have type inference, custom data types, classes and objects, or functors. However, there are three types of semantics that were implemented in the MiniML lanaguage: the subtitution model, dynamically scoped environmental model, and lexically scoped environmental model.

2 Extensions

2.1 Abstraction

To make the code more efficient, reduce code redundancies, and generally write cleaner code, I abstracted away a lot of the similar functionality between different functions.

2.1.1 Free Vars

For the `free_vars` function, I noticed that a lot of the expressions were returning `SS.empty` since they were not considered free variables. Thus, I put them all under the same match case.

```
| Num _ | Float _ | Bool _ | Raise | Unassigned -> SS.empty
```

2.1.2 Substitution

For the `subst` function, the `Let` and `Letrec` match cases had very similar functinality in how they substituted in variables. The only difference was in the case in which the variable `v` from the expression was equal to `var_name`. Thus, I abstracted into two helper functions. The first of which returned either a `Let` or `Letrec` expression based on a boolean flag `isrec`.

```
let substhelper (v : varid) (e1 : expr) (e2: expr) =
    if isrec then Letrec (v, e1, e2) else Let (v, e1, e2)
```

The second helper funtion just took the original functionality and substituted either `Let` or `Letrec` based on the given experession, using the previous helper function.

```
let rechelper (v : varid) (e1 : expr) (e2: expr) (isrec : bool) : expr =
    if v = var_name then
        if isrec then exp
        else Let (v, subst var_name repl e1, e2)
    else if SS.mem v (free_vars repl) then
        let x = new_varname() in
        substhelper x (subst var_name repl (subst v (Var x) e1))
                      (subst var_name repl e2)
    else substhelper v (subst var_name repl e1) (subst var_name repl e2)
```

2.1.3 Expression to Concrete and Abstract String

In the `exp_to_concrete_string` and `exp_to_abstract_string` both had very similar functionality with binop and unop expressions and since I was going to add other atomic types and other basic operators, it was much easier to abstract it into two helper functions: `unophelper`

```
let unophelper (u : unop) (isconcrete : bool) : string =
    match u with
    | Negate -> if isconcrete then "~-" else "Negate"
    | Negatef -> if isconcrete then "~-." else "Negatef"
    | Not -> if isconcrete then "not" else "Not"  ;;
```

and `binophelper`.

```
let binophelper (b : binop) (isconcrete : bool) : string =
    match b with
    | And -> if isconcrete then "&&" else "And"
    | Or -> if isconcrete then "||" else "Or"
    | Plus -> if isconcrete then "+" else "Plus"
    | Minus -> if isconcrete then "-" else "Minus"
    | Times -> if isconcrete then "*" else "Times"
```

```
    ...
```

Again, using a boolean flag `isconcrete`, I could determine whether it should return concrete or abstract syntax, making the code cleaner and more easy to add on extra operators later.

2.1.4 Eval Substitution, Dynamical, and Lexical

With all three evaluation methods, I also abstracted away unop and binop expressions by similar logic to the previous section into `unophelper`

```
let evalunop (u : unop) (e : expr) : expr =
    match u, e with
    | Negate, Num x -> Num (~-x)
    | Negatef, Float x -> Float (~-.x)
    | Not, Bool x -> Bool (not x)
    | _, _ -> raise (EvalError "Invalid Unop") ;;
```

and `binophelper`

```
let evalbinop (b : binop) (e1 : expr) (e2 : expr) : expr =
    match b, e1, e2 with
    | Exponent, Float x, Float y -> Float (x ** y)
    | Plus, Num x, Num y -> Num (x + y)
    | Minus, Num x, Num y -> Num (x - y)
    | Times, Num x, Num y -> Num (x * y)
    | Divide, Num x, Num y -> Num (x / y)
    ...
```

For `eval_s`, I used a helper function `eval` so that I could wrap the result at the end with `Env.Val` to prevent having to wrap the end of every match case with `Env.Val`. I also abstracted away both `eval_d` and `eval_l` into one helper function to reduce repetitive code since every match case between the two evaluation methods are the same except `Fun` and `App` where in `eval_l` you need to wrap the expressions in closures. Thus, I created the `eval_all` helper function with a boolean flag `isdynam` to determine when to use `eval_d` and `eval_l`. The main reason for this difference is that for `eval_l` we need to extend the enviornment in order keep track of the scope of certain functions.

```
let eval_all (exp : expr) (env : Env.env) (isdynam : bool) =
  let rec eval (exp : expr) (env : Env.env) =
    ...
    | Fun _ -> if isdynam then Val exp else close exp env
    | App (e1, e2) ->
        match eval e1 env with
        | Val (Fun (v, e)) ->
            if isdynam then
```

```
            eval e (extend env v (ref (eval e2 env)))
          else raise EvalException
      | Closure (Fun (v, e), en) ->
          if not isdynam then eval e (extend en v (ref (eval e2 env)))
          else raise EvalException
      | _ -> raise (EvalError "Cannot apply non functions")
```

2.2 Atomic Types and Operators

2.2.1 Floats

Besides implementing `eval_l` as an extension, I also implemented floats into the miniml interpreter and its respective operators. To do so, I first implemented an expression type:

```
type expr =
  | Float of float
```

I also needed to add extra functionality into `miniml_parse.mly` and `miniml_lex.mll` in order to make the interpreter compatible with floats using the following code:

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

let float = digit* frac? exp?

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

Furthermore, I also added functionality for adding, subtracting, multiplying, dividing, and exponentiating floats.

```
type binop =
    ...
    | Plusf
    | Minusf
    | Timesf
    | Dividef
    | Exponent
```

2.2.2 Additional Operators

Additionally, I added operators that made the miniml language more complete, including greater than `>`, less than or equal to $\leq$, greater than or equal to $\geq$, not equals `<>`, logical and `&&`, logical or ||, and `not` for negating boolean expressons.

```
type unop =
    | Not

type binop =
    | NotEquals
    | GreaterThan
    | LessEquals
    | GreaterEquals
    | And
    | Or
```

3 Testing

For testing, I utilized the `test_lite` testing suite file provided in lecture to throroughly test each of the match cases. One of the big parts of testing was creating test cases where the dynamical environment model and lexical environment model would evaluate the same expression differently. One test case for let expressions was:

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

In this case, the dynamical environment model evaluates the expression to 3 because it determines the scope of variables at runtime and reassigns x to 2 and calculates it as $2 + 1 = 3$. However, the lexical environment model evaluates the expression to 2 because it considers the local lexical environment at compile time, causing x to remain as 1 and calculates it as $1 + 1 = 2$. Several other similar test cases were written for `Fun`, `Let`, and `Letrec`.

4 Conclusion and Next Steps

If I had more time to work on the project, I would include a way to determine which type of evaluator to use (`eval_s`, `eval_d`, `eval_l`) through the terminal since it was inconvenient to manually change the evaluator each time in `evaluation.ml`. It would also be interesting to include more extensions such as lists and records to further extend the miniml interpreter.