Final Project Write-up

Tomás Arevalo

May 3 2023

1 Overview

The final project tasked me to create my own miniature version of an OCaml interpreter called MiniML. We were given a parser implemented in `miniml.ml`, `miniml_lex.mll`, and `miniml_parse.mly`, as well as some starter code in both `expr.ml` and `evaulation.ml`. My main responsibility was to create two different interpreters based on either substitution or environment semantic rules. The first was based on substitution semantics which we called `eval_s`; the second was `eval_d`, based on the semantics of a dynamically scoped environment. In order to implement these, I had to implement functions that counted the number of free variables in an expression, functions that took expressions and converted them into strings with either concrete or abstract syntax, and functions that were detailed with all the rules for substitution and for dynamically scoped environment.

After implementing the substitution model and the dynamic model, which can be seen below, it was time to implement my own extensions. Possible extensions varied from adding additional atomic types, modifying the environment semantics to implement a lexical scope, to even adding type inference to the language.

```
(* The SUBSTITUTION MODEL evaluator -- to be completed *)

let eval_s (exp : expr) (_env : Env.env) : Env.value =
  let rec eval_s' (exp: expr) : expr =
    match exp with
    | Var v -> raise (EvalError ("Unbound var " ^ v))
    | Num _ | Float _ | Bool _ | Fun (_, _) | Unassigned -> exp
    | Unop (op, e) -> unop_helper op (eval_s' e)
    | Binop (op, e1, e2) -> binop_helper op (eval_s' e1) (eval_s' e2)
    | Conditional (e1, e2, e3) ->
        (match eval_s' e1 with
        | Bool true -> eval_s' e2
        | Bool false -> eval_s' e3
        | _ -> raise (EvalError "Conditional expects a bool value"))
    | Let (x, def, body) ->
        let value = eval_s' def in
        eval_s' (subst x value body)
    | Letrec (x, def, body) ->
        eval_s' (subst x (subst x (Letrec (x, def, Var x)) def) body)
    | Raise -> raise EvalException
    | App (e1, e2) ->
        match eval_s' e1 with
        | Fun (v, e) -> eval_s' (subst v (eval_s' e2) e)
        | _ -> raise (EvalError "First arg must be function ")
  in
  Env.Val (eval_s' exp) ;;
```

```
(* Orginal eval_d; before make_model *)

let rec eval_d (exp : expr) (env : Env.env) : Env.value =
  match exp with
  | Var v ->
      (match Env.lookup env v with
      | Env.Val e -> Env.Val e
      | Env.Closure (_, _) -> raise (EvalError "Closure in Dynamic"))
  | Num _ | Float _ | Bool _ | Fun (_, _) | Unassigned -> Env.Val exp
  | Unop (op, e) -> Env.Val (unop_helper op (val_to_expr(eval_d e env)))
  | Binop (op, e1, e2) ->
      Env.Val (binop_helper op
                  (val_to_expr (eval_d e1 env))
                  (val_to_expr (eval_d e2 env)))
  | Conditional (e1, e2, e3) ->
      (match eval_d e1 env with
      | Env.Val (Bool true) -> eval_d e2 env
      | Env.Val (Bool false) -> eval_d e3 env
      | _ -> raise (EvalError "Conditional expects a bool value"))
  | Let (v, def, body) | Letrec (v, def, body) ->
      let valref = ref (eval_d def env) in
      eval_d body (Env.extend env v valref)
  | Raise -> raise EvalException
  | App (e1, e2) ->
      let valref = ref (eval_d e2 env) in
      match eval_d e1 env with
      | Env.Val (Fun(v, e)) -> eval_d e (Env.extend env v valref)
      | _ -> raise (EvalError "First arg must be function ") ;;
```

2 Extensions

Due to a time constraint and other course work, I implemented a third interpreter based on the environment semantics to manifest a lexical scope, added the atomic type `Float` and its corresponding operations, and implemented extra Boolean operations.

2.1 Lexical Extension

Ocaml follows the rules of a lexically scoped environment and thus, it was the first extension that I approached. Following the rules in Chapter 19 of the textbook, I quickly realized that a lot of the rules were similar or the same when compared those of `eval_d`. Thus, when implementing `eval_l`, the rules I had to change were that of `Fun`, `Let` expressions, `Letrec` expressions, function `App`, and a slight tweak to `Var`. This makes sense because the only real differences in ouputs between dynamic and lexical environments, are when storing the values of variables, and then applying them later to functions or Let expressions. We can see this difference below:

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

(a) Dynamic Environment Output

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

(b) Lexical Environment Output

As we can see in the figures above, the dynamic environment doesn't store the original value of x, bur rather waits until the function is called, and then uses the most recent value of x. However, in the lexical environment, the function stores the most recent value of x before the function was made, and then preserves it, even if changes to the value of x occur. Below we can see the implemented version of `eval_l`

```
(* Orginal eval_d; before make_model *)

let rec eval_d (exp : expr) (env : Env.env) : Env.value =
  match exp with
  | Var v ->
      (match Env.lookup env v with
      | Env.Val e -> Env.Val e
      | Env.Closure (_, _) -> raise (EvalError "Closure in Dynamic"))
  | Num _ | Float _ | Bool _ | Fun (_, _) | Unassigned -> Env.Val exp
  | Unop (op, e) -> Env.Val (unop_helper op (val_to_expr(eval_d e env)))
  | Binop (op, e1, e2) ->
      Env.Val (binop_helper op
                  (val_to_expr (eval_d e1 env))
                  (val_to_expr (eval_d e2 env)))
  | Conditional (e1, e2, e3) ->
      (match eval_d e1 env with
      | Env.Val (Bool true) -> eval_d e2 env
      | Env.Val (Bool false) -> eval_d e3 env
      | _ -> raise (EvalError "Conditional expects a bool value"))
  | Let (v, def, body) | Letrec (v, def, body) ->
      let valref = ref (eval_d def env) in
      eval_d body (Env.extend env v valref)
  | Raise -> raise EvalException
  | App (e1, e2) ->
      let valref = ref (eval_d e2 env) in
      match eval_d e1 env with
      | Env.Val (Fun(v, e)) -> eval_d e (Env.extend env v valref)
      | _ -> raise (EvalError "First arg must be function ") ;;
```

2.2 Float Extension

The next extension I implemented was adding the `Float` atomic type. This was done by adding `Float` to the `type expr`, and then editing the parser to be be able to identify a `Float`. The main edit to the parser, which is shown below, allows the parser to be able to differentiate between a `Float` and an `Int`.

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

After doing this, an input such as `3.+ 4.` would output `7.`, but in Ocaml, this would be an error. Thus, I decided to implement all the operations for a Float which were FloatPlus (+.), FloatMinus (−.), FloatTimes (∗.), FloatDivided (/.), and FloatNegate (∼ −.). (Side note: I also added regular Divided for Num, as it didn't make sense to only have it for floats and not Nums). Adding all these operations were easily added by adding the to the hash table, making them tokens, and then adding their respective grammar rules.

```
| exp DIVIDED exp       { Binop(Divided, $1, $3) }
| exp FLOATPLUS exp      { Binop(FloatPlus, $1, $3) }
| exp FLOATMINUS exp     { Binop(FloatMinus, $1, $3) }
| exp FLOATTIMES exp     { Binop(FloatTimes, $1, $3) }
| exp FLOATDIVIDED exp   { Binop(FloatDivided, $1, $3) }
  | FLOATNEG exp          { Unop(FloatNegate, $2) }
```

Obviously, after editing the parser, I went ahead and added all these operations into functions in `expr.ml` and `evaluation.ml`.

2.3 Boolean Extensions

The last extension I added was Boolean operators such as `not`, && `(AND)`, `||` `(OR)`. This would allow me to do this like `not true = false`, `false` && `true = false`, and `false || true = true`. Similar to the `Float` operators, it was just adding them to the hash table, making sure '|' was in `sym []`, making them tokens, and modifying the grammar rules as seen below.

```
| BOOLNEGATE exp        { Unop(BoolNegate, $2) }
| exp AND exp            { Binop(And, $1, $3) }
| exp OR exp             { Binop(Or, $1, $3) }
```

3 Abstraction

After implementing my extensions, I realized that there was so much redundant code between `eval_d` and `eval_l`. This let me to the idea of making a function `make_model` which would intake the same variables as the dynamic and lexical models, but also intake the type of environment semantic it had to follow (either `Dynamic` or `Lexical`). Thus, I implemented the following code which saved me 14 lines of codes before adding comments.

```
type semantics = | Dynamic | Lexical ;;

let make_model (sem : semantics) (exp : expr) (env : Env.env) : Env.value =
  let rec make (exp: expr) (env : Env.env) : Env.value =
    match exp with
    | Var v ->
        (match Env.lookup env v with
        | Env.Val e -> Env.Val e
        | Env.Closure (ex, new_env) ->
                    if sem = Dynamic then
                      raise (EvalError "Closure in Dynamic")
                    else
                    make ex new_env)
    | Num _ | Float _ | Bool _ | Unassigned -> Env.Val exp
    | Fun _ -> if sem = Dynamic then Env.Val exp
              else Env.close exp env
    | Unop (op, e) -> Env.Val (unop_helper op (val_to_expr(make e env)))
    | Binop (op, e1, e2) ->
        Env.Val (binop_helper op
                            (val_to_expr (make e1 env))
                            (val_to_expr (make e2 env)))
    | Conditional (e1, e2, e3) ->
        (match make e1 env with
        | Env.Val (Bool true) -> make e2 env
        | Env.Val (Bool false) -> make e3 env
        | _ -> raise (EvalError "Conditional expects a bool value"))
    | Let (v, def, body) ->
        let valref = ref (make def env) in
        make body (Env.extend env v valref)
    | Letrec (v, def, body) ->
        if sem = Dynamic then
          let valref = ref (make def env) in
          make body (Env.extend env v valref)
        else
          (* Follows steps on pg 14 of Readme *)
          let valref = ref (Env.Val Unassigned) in
          let env_x = Env.extend env v valref in
          let v_D = make def env_x in
          (match v_D with
          | Env.Val Var _ -> raise(EvalError "Unassigned Var")
          | _ -> valref := v_D; make body env_x)
    | Raise -> raise EvalException
    | App (e1, e2) ->
          let valref = ref (make e2 env) in
          match make e1 env with
          (* Will Only Happen if Dynamic *)
          | Env.Val (Fun(v, e)) -> make e (Env.extend env v valref)
          (* Will Only Happen if Lexical *)
          | Env.Closure (Fun(v, e), previous) ->
            make e (Env.extend previous v valref)
          | _ -> raise (EvalError "First arg must be function ")
  in
  make exp env ;;
```