CS51 Final Project Writeup

Kaden Zheng

May 1st, 2024

1. Overview

In my CS51 final project, I implemented the MiniML language beyond its minimal implementation of substitution and dynamic environment semantics evaluations. My first extension introduced lexical scoping (akin to the 'normal OCaml' evaluator), which I achieved by allowing the capture of a function's environment using the Closure type and adapting *Fun, App,* and *Letrec* to handle lexical scoping. Second, I added support for *Floats* to expand the range of operations available in MiniML, specifically through modifying the parser to recognize float literals, extending the evaluator to handle float values and operations, and updating the *eval_ binop* and *eval_ unop* functions accordingly. Third, I added support for *Strings* and their respective operations, similar to the implementation for floats. My final extension adds support for function currying. Though the implementation of this extension was the shortest of the three—only in having to modify *miniml_ parse.mly*—I required much Office Hours assistance to understand how to support functions with multiple arguments. Throughout this write-up, I will reference the thorough testing conducted in *tests.ml* to validate the performance of each extension, as well as key decisions I made throughout their implementations.

2. Lexically-Scoped Environment

In Stage 9, we implement *eval_ d*, a function that evaluates in a dynamically-scoped environment. Then, my first extension augments this to form a lexically scoped environment (as in OCaml itself). Lexical environments ensure the bindings of variables

are determined by their static position in the code, whereas in a dynamically scoped environment, the bindings depend on the sequence of function calls that lead to the point of the variable's reference.

We achieve this in *eval_ l*. There exist three key differences between *eval_ l* and *eval_ d*, tied to the implementation of a *Closure*. As defined in the *readme*, a closure packages a snapshot of the environment at the time of a function's definition, which allows the function to access variables from a lexical scope.

Functions

In *eval_ l*, we create a closure using the *close* function provided in our defined *Env* module. This closure captures the function and its environment at the point of definition. Observe the contrast to *eval_ d*, which simply returns the function without capture its environment.

```
| Fun (v, e) ->
      (match model with
       | Lexical -> close (Fun (v, e)) env
       | Dynamic -> Val (Fun (v, e)))
```

Application

When applying a function in *eval_ l*, we check if the evaluated function is a closure and then extend the closure's captured environment with a binding of the function parameter v to the evaluated argument value *e2_ val*. The function body e is then evaluated in this new environment, whereas, in *eval_ d*, the function application directly extends the current environment without considering the closure's captured environment. For brevity, the code snippet is only of my lexical implementation.

```
| App (e1, e2) ->
      (match model with
       ...
       | Lexical ->
         let e1_val = eval_aux e1 in
         let e2_val = eval_aux e2 in
```

```
      (match e1_val with
       | Closure (Fun (v, e), c_env) ->
         let new_env = extend c_env v (ref e2_val) in
         eval_generator model e new_env
       | _ -> raise (EvalError "app: invalid function application")))
```

Letrec

Letrec was the trickiest. After following the guide in the *readme*, I implemented *Letrec* by extending the environment with a reference to the recursive function, then evaluating the body in the new environment (with an error raised when an unbound variable appears). Further details are in the code comments.

```
| Letrec (v, e1, e2) ->
      (match model with
       ...
       | Lexical ->
         let r = ref (Val Unassigned) in
         let new_env = extend env v r in
         let vs = eval_generator model e1 new_env in
         (match vs with
          | Val (Var _) -> raise (EvalError "letrec: unbound variable")
          | _ -> r := vs; eval_generator model e2 new_env))
```

Testing

Testing these differences boiled down to writing expressions with different outputs when evaluated in a dynamic vs. lexical environment.

```
(* for instance, this expression evaluates to 4 with lexical scoping,
and 5 with dynamic scoping *)
let x = 1 in let f = fun y -> x + y in let x = 2 in f 3 ;;

(* Tested by: (and likewise for dyn) *)
unit_test (eval (str_to_exp "let x = 1 in let f = fun y -> x + y in
let x = 2 in f 3 ;;") = Val (Num 4)) "eval_s lexical scoping";
```

I similarly test for *Letrec* through a simple recursive factorial function.

```
(* for instance, this expression evaluates to 1 with lexical scoping,
and 2 with dynamic scoping *)
let x = 1 in
let rec f n =
  if n = 0 then x
  else let x = 2 in f (n - 1)
in f 5
(* Tested by: (and likewise for dyn) *)
unit_test (eval (str_to_exp "let x = 1 in let rec f n = if n = 0 then x else
let x = 2 in f (n - 1) in f 5") = Val (Num 2)) "eval_l shadowing in letrec";
```

The full set of unit tests can be viewed in *tests.ml*.

Abstraction

As an additional note, I abstracted out the similarities between the substitution, dynamic, and lexical models into a helper function *eval_generator*, which takes the additional argument *model* of type *evaluator*, which indicates the evaluation model to be used. The actual evaluation of the chosen model is handled by my nested function *eval_aux*. Similarly, I abstracted out the test cases common to both into a *model_test_generator* function that takes the model as a parameter and generates the appropriate set of tests for that model (since many of the trivial tests (binary ops, conditionals, etc.) evaluate the same for all models).

3. Floats

I then implemented float support for MiniML, which was complicated not for the implementation itself but since I needed to understand the lexical analyzer (*miniml_lex.mll*) and the parser (*miniml_parse.mly*).

First, I realized that I needed to add a new token *FLOAT* into *miniml_lex.mll* since it was responsible for defining elements, and then I added a new regex rule to recognize floating-point numbers (e's for sci. notation). Specifically,

```
(* regex for identifying floats *)
let float = digit+ ('.' digit*)? (['e' 'E'] ['+' '-']? digit+)?
...
rule token = parse
| float as fnum
        { let num = float_of_string fnum in
          FLOAT num
        }
```

handles this implementation.

I also needed to update the parser in *miniml_parse.mly* which constructs the abstract syntax tree based on the grammar rules (which I updated for floats) and the tokens from the analyzer above.

This I achieved by adding a new *FLOAT* token to the list of tokens, updating the *exp* and *expnoapp* rules to include this *FLOAT* token, and adding new binary operator rules for float operations. That is,

```
%token <float> FLOAT
...
expnoapp:
        | exp FPLUS exp       { Binop(FPlus, $1, $3) }
        | exp FMINUS exp      { Binop(FMinus, $1, $3) }
        | exp FTIMES exp      { Binop(FTimes, $1, $3) }
```

handles the updated 'grammar rules.'

Evaluating Floats

Finally, adding floats mandates updating the expression types and evaluation methods. In *expr.ml*, I added the *Float* constructor as an *expr* type, and updated the *binop* type to include float-specific binary operators, namely *FPlus, FMinus, FTimes, FDivide.*

```
type binop =
  ...
```

```
  | FPlus
  | FMinus
  | FTimes
  | FDivide
  ...
type expr =
  ...
  | Float of float                    (* floats *)
```

Here, in modifying the unary and binary operations for evaluating floats, I had to make my first key decision: whether to make MiniML strongly typed (that is, operations only occur between operands of the same expected type) or weakly typed. Since OCaml itself is strongly typed (and due to the plethora of extra operations I would need to add to *eval_ binop* if MiniML were weakly typed), I chose to make MiniML strongly typed. Thus, I updated the *eval_ binop* function to handle *FPlus, FMinus, FTimes, FDivide* and float comparisons *Equals, LessThan* accordingly.

```
let eval_unop (op: unop) (e: value) : value =
  match (op, e) with
  | (Negate, Val (Num n)) -> Val (Num (-n))
  | (Negate, Val (Float f)) -> Val (Float (-.f))
  | _ -> raise (EvalError "Invalid unary operator")
...
let eval_binop (op: binop) (e1: value) (e2: value) : value =
  match (op, e1, e2) with
  ...
  | (FPlus, Val (Float f1), Val (Float f2)) -> Val (Float (f1 +. f2))
  | (FMinus, Val (Float f1), Val (Float f2)) -> Val (Float (f1 -. f2))
  | (FTimes, Val (Float f1), Val (Float f2)) -> Val (Float (f1 *. f2))
  | (FDivide, Val (Float f1), Val (Float f2)) ->
      if f2 = 0.0 then raise (EvalError "float: division by zero")
      else Val (Float (f1 /. f2))
  | (Equals, Val (Float f1), Val (Float f2)) -> Val (Bool (f1 = f2))
  | (LessThan, Val (Float f1), Val (Float f2)) -> Val (Bool (f1 < f2))
  | _ -> raise (EvalError "eval_binop: invalid binary operator")
```

handles the appropriate float adjustments. These evaluations are rigorously tested in *tests.ml*, omitted for brevity.

4. Strings

My third extension adds support for *Strings*. Having gone through the process of adding FLOAT tokens, a regex identifier, and float grammar rules, this process was relatively simple. First, I added a new STRING token into *miniml_lex.mll* through:

```
(* regex for identifying strings (namely matching the quotes) *)
let string = ['"'] [^ '"']* ['"']
...
rule token = parse
| string as str
  (* Removing the quotes as part of the string *)
        { let len = String.length str in
          STRING (String.sub str 1 (len - 2)) }
```

I also updated the parser in *miniml_parse.mly* by simply updating the grammar rules and adding the string token.

```
%token <string> STRING
...
expnoapp:
        | STRING                { String $1 }
```

handles such rules.

Evaluating Strings

Implementing the evaluation of strings followed a similar procedure to the implementation of floats. Here, I did make the decision to simply use the existing binary operators *Plus, LessThan,* and *Equals* to represent concatenation and comparing strings. While I could have implemented a new *Concat* binary operator to handle concatenation, I figured the hassle of refactoring was unnecessary.

Thus, I simply added the *string* type to *expr*:

```
type expr =
    ...
    | String of string                          (* string *)
```

and then updated the *eval_binop* function to handle these operations accordingly.

```
let eval_binop (op: binop) (e1: value) (e2: value) : value =
  match (op, e1, e2) with
  ...
  (* Support for strings *)
  | (Plus, Val (String s1), Val (String s2)) -> Val (String (s1 ^ s2))
  | (Equals, Val (String s1), Val (String s2)) -> Val (Bool (s1 = s2))
  | (LessThan, Val (String s1), Val (String s2)) -> Val (Bool (compare s1 s2 < 0))
  | _ -> raise (EvalError "eval_binop: invalid binary operator")
    ;;
```

handles this implementation.

I will say that I spent far too long debugging where I kept getting the *EvalError "eval_binop: invalid binary operator"*. My mistake was placing the pattern matching for string operations after the catch-all pattern *_ -> raise (EvalError "eval_binop: invalid binary operator")*, and so the catch-all pattern was matched before reaching the string cases.

Testing

Since the evaluation of strings in all the environment semantics remains constant, I added a set of string unit tests to the non-model-dependent portion of my test generator, the most relevant of which are featured below:

```
unit_test (eval (str_to_exp "\"hello\" + \" world\" ;;") =
    Val (String "hello world")) (eval_name ^ " string concatenation");
unit_test (eval (str_to_exp "\"hello\" = \"hello\" ;;") =
    Val (Bool true)) (eval_name ^ " string equality");
unit_test (eval (str_to_exp "\"abc\" < \"def\" ;;") =
    Val (Bool true)) (eval_name ^ " string less than");
```

```
unit_test (eval (str_to_exp "let greet = fun name -> \"Hello, \"
    + name in greet \"Alice\" ;;") = Val (String "Hello, Alice"))
    (eval_name ^ " string function application");
```

I also added two cases into the lexical testing that were dependent on scoping, but omitted them here since scoping was extensively tested before (and the handling of strings is no different than ints, floats, etc.).

5. Function Currying

My final extension allows for function currying. At first, I thought this would require modifying *Expr* to allow for a new form of function application. However, after revisiting *Section 6.2*, I realized that currying was essentially syntactic sugar for nested functions. For instance, we may define:

```
let f x y = x * y in f 1 2 ;;
```

as

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

and native OCaml will evaluate the two as the same. However, as OCaml evaluates with lexical scoping, this is only available to us when MiniML runs with *eval_l*. This is key because, in our MiniML lexical implementation, we are able to nest functions inside one another, meaning the inner function has access to the variables and parameters of the outer function (closure). This means MiniML already supports capturing variables from an outer surrounding scope, and thus, we can 'encapsulate' variables. What this means, then, is that we need only to augment the parsing in *miniml_parse.mly* to allow for function currying (technically, we are implementing grammar rules that allow for the first expression above to be evaluated as the second).

idlist Rule

First, I implemented the *idlist* rule to handle lists of identifiers. I learned (after OH) that I could not parse idlist directly into an expression, hence my treatment of the ids as a list.

```
idlist:  ID idlist              { $1 :: $2 }
       | ID                     { [$1] }
```

This is key because this rule matches when there is an identifier followed by more identifiers (another idlist) i.e. matching sequences of identifiers like x y z. Technicality-wise, this is implemented thru the first case, which prepends an identifier (via ::) if it is followed by another *idlist*, or, in the second case, it wraps the single identifier in a list using [$1].

Grammar Rules

Second, I allowed for the definition of functions with multiple parameters using the *let*, *let rec* constructs, and for anonymous functions with multiple parameters (*DOT*) via:

```
| LET ID idlist EQUALS exp IN exp   { Let($2, convert_to_fun $3 $5, $7) }
| LET REC ID idlist EQUALS exp IN exp { Letrec($3, convert_to_fun $4 $6, $8) }
| FUNCTION ID idlist DOT exp   { Fun($2, convert_to_fun $3 $5) }
```

convert_to_fun

Finally, I implemented a helper function *convert_to_fun* that converts *idlist* to a nested function structure as demonstrated with the example expressions above.

```
let convert_to_fun idlist exp =
match idlist with
| [] -> exp
| _ -> List.fold_right (fun x acc -> Fun(x, acc)) idlist exp
```

When *idlist* is empty, we know there are no additional parameters, and thus, the function returns the expression as is. Otherwise, I use *List.fold_right* to iterate over the identifiers from right to left (preserving the nesting order) to create a new *Fun* expressions with x as the parameter and *acc* as the accumulated body expression. In the end, the nested structure of *Fun* expressions represents the curried function! These are all applied with modified grammar rules for *Let*, *Letrec*, and *Fun*.

Testing

Finally, I tested this currying functionality thoroughly through curried expressions such as:

```
let add x y = x + y in add 1 2 ;;
(* via the test: *)
unit_test (eval (str_to_exp "let add x y = x + y in add 1 2 ;;") =
Val (Num 3)) "eval_l currying basic";

(* or this more complicated, 'nested currying' w/ function app *)
let add x y = x + y in let apply f x y = f x y in apply add 1 2 ;;
unit_test (eval (str_to_exp "let add x y = x + y in let apply f x y =
f x y in apply add 1 2 ;;") = Val (Num 3)) "eval_l currying apply";

(* and so forth. full set in tests.ml *)
```

As these tests pass cleanly, I have successfully implemented the function currying into MiniML for the lexical environment.

Conclusion

Overall, my extension of the MiniML language gave me a deeper understanding of how variable bindings are handled in various programming languages and how parsers converts concrete syntax to abstract representations. By introducing lexical scoping, I enabled the capturing of function environments using closures, which allowed for static variable bindings. Then, my addition of float and string support introduced modifications into the lexical analyzer *miniml_lex.mll* and the parser *miniml_parse.mly*. By introducing new grammar rules and adding the *FLOAT* + *STRING* tokens accordingly, I gained an understanding of how the parser handles data types and how the evaluation functions handle operations. I used this understanding, as well as TF assistance, for the implementation of function currying, which allows for the creation of partially applied functions and higher-order programming techniques akin to OCaml itself.

Throughout this process, I faced three crucial junctions that allowed me to exercise judgment as a programmer. First, as the notion of implementing semantics

is quite abstract, I had much trouble initially with handling closures in *eval_l.ml*. However, by closely following the *readme*, I was able to implement the correct variable capture and access from the lexical scope. Debugging my *Env* module and completely implementing *eval_l.ml* gave me the confidence to attempt the *Float* extension. Second, while implementing float support, I had to decide whether to make MiniML strongly or weakly typed when introducing float support. Ultimately, I chose to make MiniML strongly typed which aligns more with OCaml's type principles and avoiding the complexities of type coercion. Finally, when implementing function currying, I had to decide how to parse and transform multiple-parameter functions. By introducing the *idlist* rule and the *convert_to_fun* helper function (treating the extra arguments as a list vs. expressions themselves), I was able to convert function definitions with multiple parameters into a nested structure of curried functions.

Having validated each step through thorough testing in *tests.ml*, I am pleased with my implementation of *MiniML* (which has much abstraction!). Thank you to the TF who is reading my writeup, and to all the TFs for saving my life during labs, code review, and Office Hours!

Best wishes and take care,
Kaden :)