CS 51 Final Writeup

Jeremy Laue

May 2023

1 Intro

Overall I found this project to be an effective synthesis of the course as a whole. Not only does it cause us to bring together every element of course curriculum gleaned throughout the semester but it gets us to think about the underlying mechanics of Ocaml and other languages in a way that I personally have never thought of before in such detail. With the exception of the time involvement, I have found this project very fulfilling and enjoyable and I appreciated the creative freedom of the extensions.

2 Extension: Atomic Type - Unit

Starting with the simplest of the extensions we have the unit value. There is next to no functionality with the unit type so I didn't need to implement any unop, binop, or other functions as a result. I implemented units by creating a varid type:

```
Unit of unit                    (* unit *)
```

To add units the parser and ensure their recognition, I merely inserted the following code into the parser file, creating a unit token and connecting units with the characters "()".

```
%token UNIT
| '(' ')' { UNIT }
```

3 Extension: Atomic Type - Float

Next we have the slightly more complex float type. In addition to the types implementation, I new instances of binops for float addition, multiplication, subtraction, division, and negation. While I was at it, I also added integer division. Folliwing a similar patter as for units, I first created the varid type :

```
Float of float                  (* floats *)
```

To create parser float recognition I created a float token (not shown because of repetition) and then instantiated them as digits followed by decimals with the following lines of code.

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

For my new float operations (and integer division) I merely created them as new instances of binops in the expr files. Then in the evaluation file, I implemented each to evaluate to their correct evaluation in my binop auxiliary function pattern matching a binop expression and following the same pattern as below.

```
| Fl_plus, Val (Float (x)), Val (Float (y)) -> Float (x +. y)
```

Finally, I instantiated the operation to be recognized in the parser with the following line of code (one per each function):

```
| exp FL_PLUS exp       { Binop(Fl_plus, $1, $3) }
```

4 Extension: Atomic Type - String

For strings, I created them as an atomic type in a similar fashion as floats and units and instantiated them into the parser this time with the following line of code that recognized them as a series of characters enclosed in quotation marks.

```
| '"' id* '"' as str
{ STRING str }
```

Next, I implemented the concat function that combines two strings into one. I did this by first ensuing the ˆ character is recognized as the concat operator and then created an binop concat expression (see above for FL_PLUS) that merely took two expressions separated by CONCAT and created a string out of them.

Since all atomic types evaluate to themselves regardless of the type of evalutation, I needed to make no changes to any of my evaluation functions to incorporate these new additions.

5 Extension: Lexical Scope Evaluation

To implement a lexical scope evaluation function there are a few notable differences between lex and dynamical that must be taken into account. These lie in the following varid types : Fun and App. The remainder of the types can be evaluated using a call to the dynamic evaluation function.

1. Fun. Functions are just evaluated to create a closure with the environment at the time of definition. This is done by evoking the Env.close function in the following line of code.

```
        | Fun (var, def) -> Env.close (Fun (var, def)) _env
```

2. App. Application of functions are evaluated lexically by using the store created when the function was defined. This is done by utilizing the following lines of code

```
    | App (func, argu) ->
        (match eval_d func _env with
        | Closure (Fun (var, def), env') ->
        (eval_l def (Env.extend env' var (ref (eval_l argu _env))))
```

   which first pattern match the application to the closure of the function and then evaluates the function as defined in the closure. As stated before, the rest of the cases of varid's are evaluated the same in lexical and dynamic evaluation so apart from small adjustments in let and letrec, the remainder of the function merely calls the dynamic evaluation function.

6 Extension: References

The most difficult extension I implemented was certainly the addition of reference types in Miniml. First and foremost, I created two additional expression types, one that creates a reference type and the other that dereferences a reference:

```
    | Ref of varid * expr                    (* references *)
    | Deref of varid                         (* dereferencing *)
```

My greatest challenge lied in determining the evaluation of references. I knew that references only exist in dynamical and lexical evaluation so I only needed raise an error with substitution evaluation when encountered with references.

However as learned throughout the course, the semantics of references involve both a return type and an augmentation to the memory of the program. So the return of reference evaluation needs to return a tuple of both the return value and the reference mapping (environment). Since this doesn't fit the type constraint of lexical and dynamic evaluation, I decided to implement the eval_e function as my reference evaluator. Here I used the following code to dictate what to do when encountered with referencing or dereferencing:

```
    | Ref (r, ex) -> let ex', env' = eval_e ex _env in
      Val (Unit ()), Env.extend env' r (ref ex')
    | Deref ex -> Env.lookup _env ex, _env
```

When a reference is created, the unit type is returned along with an extension of the environment that maps the reference name to its value. The dereference expression looks up the value based on the variable name and returns the value along with the environment containing this mapping. I additionally added the reassignment operator ":=" by creating another binop expression that alters the values of refs. I adjusted the parser in a similar fashion to other binop characters to recognize ":=" as the reassignment operator. Well, it seems like we've implemented all of references right? Wrong!

Since this new evaluation function doesn't obey the type constraint of the rest of the evaluation functions, the miniml file doesn't compile and therefore the program doesn't run. As a result, I created a supplemental miniml file called refsminiml that uses evaluation functions that return a tuple of type Env.value * Env.env. I augmented this file in two place to make this change. First I let the default environment be a reference to an empty environment instead of merely an empty environment:

```
    let env = ref (Ev.Env.empty ()) ;;
```

Then, I call the eval_e function and update the reference to the environment to incorporate any new changes.

```
let res, env' = Ev.eval_e exp !env in
         env := env' ;
```

And with that we have our completed miniml language fitted with many different modes of evaluations, atomic types like floats, strings, and units, and last but not least, reference types and mutable storage.