CS51 Final Project: Writeup

Olivia Ferraro

May 2, 2018

0.1 Extensions

0.1.1 Lexical Scoping

The first extension I chose to include is the implementation of a lexically-scoped environment for `MiniML`. I chose this extension because the concept of a `Closure of (expr * env)` was at first both confusing and intriguing to me, and I wanted to explore it more and try my own hand at its implementation. To do this, I completed `eval_l`, beginning first with the skeleton code provided to me in the source code.

Lexical scoping means that a given function is evaluated within the environment in which it was defined. This is in contrast to substitution and environment semantics, both of which I implemented in `eval_s` and `eval_d` respectively. In `eval_s` the substitution model is used to evaluate expressions in a constant environment, while in `eval_d` expressions are evaluated a dynamic environment.

To best describe the relationship between environment and lexical semantics, consider an expression `e1` that is used within another expression `e2`. Within the environment in which it was defined, `e2` evaluates to some value, `v1`. Following environment semantics, if `e1` is redefined between the time `e2` is defined and the time `e2` is evaluated, evaluating `e2` in the most current environment will yield different results, some value, `v2`, rather than `v1`.

However, if evaluated in a lexically-scoped environment, `e2` will always evaluate to `v1`. This is true because expressions are evaluated within the environment in which they were originally defined, meaning any redefinition of any components of the original expression, `e2`, will have no influence on the evaluation of `e2`. To implement this, I used my `eval_d` as a starting off point. The only cases that vary between the two environments are those for `Fun of varid * expr` and `App of expr * expr`. Yet, I did not call `eval_d` on these cases as to shorten my code because for simple operations like `Binop of binop * expr * expr`, each element must be evaluated to a value before the binary operation can be completed, and there might be an `App of expr * expr` within one of the expressions of in which evaluating within a dynamic enviroment would be incorrect.

For the `Fun of varid * expr` case, instead of returning just an `Env.Val` containing the expression, I returned an `Env.Closure` containing a tuple with the expression, `exp`, and the current environment, `env`.

```
| Fun (_, _) -> Env.close exp env
```

As a result, within the `App of expr * expr` case, evaluating the first expression within the tuple should yield an `Env.Closure` as only functions that eventually match with `Fun of varid * expr` can be applied to arguments, and the `Fun of varid * expr` is the only case that will ever produce a `Env.Closure` when evaluated. Thus, `Fun of varid * expr` is the only possible expression that can be stored in a `Env.Closure`. As a result, an additional match statement to check for other types of expressions is unnecessary. The environment contained within the resulting `Env.Closure` is then extended to include the `varid` of the function bound to a `ref Env.Val` containing the original argument of `App of expr * expr` evaluated. This extended environment is the one that is then used to evaluate the body of function.(1)

```
| App (a, b) ->
  (match eval_l a env with
  | Env.Closure (Fun (x, y), fun_env) ->
    let new_env = Env.extend fun_env x (ref (eval_l b env)) in
    eval_l y new_env
  | Env.Val x ->
    (match x with
    | App (_, _) as j -> eval_l j env
    | _ -> raise(EvalError
      "This is not a function; it cannot be applied."))) ;;
```

In addition to matching for a `Env.Closure` when evaluating the first element of `App of expr * expr`, I also included a match for `Env.Val` with expression `App of expr * expr`. I did this because in the case that the application itself contains other applications and so on, nested applications would need to be evaluated accordingly.

0.1.2 Floats

The second extension I chose for my version of `MiniML` is support for the `float` type. In adding `floats`, I differentiated between the original numeric constructor provided, renaming the source code's `Num` expression `Int` and adding the `Float` expression.

I did this by reading through documentation online, (2) `miniml_parse.mly`, `miniml_lex.mll`, as well as the project specification and links provided in `project.pdf` about parsing and lexing. Once I had a basic understanding of the topic, I added the appropriate token:

```
%token <float> FLOAT
```

along with the corresponding grammar definition, both in `miniml_parse.mly`:

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

From there, I edited `miniml_lex.mll` to contain the regular expressions necessary to support the `float` type: (2)

```
let frac = '.' digit*
let exp = ['e' 'E'] ['-' '+']? digit+
let float = digit* frac? exp?
```

and the appropriate lexing rules:

```
| float as ifl
    { let fl = float_of_string ifl in
        FLOAT fl
    }
```

With `miniml_parse.mly` and `miniml_lex.ml` updated, I then revisited my helper functions in `expr.ml` and `evaluation.mll`. Because I abstracted away conversion to string and simple operations by delegating them to helper functions in both these files, adding functionality to support `floats` was not tremendously laborious. In addition, to make my version if `MiniML` more user-friendly, I abstracted away the separate symbols for binary operations OCaml uses for `ints` and `floats`. Instead, the simpler symbols (reserved exclusively for `ints` in OCaml) can be applied to `floats` as well, provided both arguments are of the same type.

0.1.3 More Binary Operations

In addition to including a lexically-scoped environment and support for the `float` type, I also added two new binary operations of type `binop`. I added `Div`, which divides the first expression by the second, as well as `GreaterThan`, which returns `Bool true` if the first argument is greater than the second. Although the user could derive both of these operations through the functionality provided in the original source code, multiplying by the inverse for `Div` and reversing the arguments of a `LessThan` expression for `GreaterThan`, providing them simplifies the user's experience.

To add both of these operations, I followed a similar protocol as outlined above in the implementation of the `float` type. My first steps were to add both the / and > symbols as recognized tokens, designate their association, and add the corresponding grammar definition in `miniml_parse.mly`.

Tokens:

```
%token GREATERTHAN
%token TIMES DIV
```

Association:

```
%nonassoc GREATERTHAN
%left TIMES DIV
```

Grammar Definition:

```
| exp DIV exp            { Binop(Div, $1, $3) }
| exp GREATERTHAN exp    { Binop(GreaterThan, $1, $3) }
```

Then, I added them to my sym_table in miniml_lex.mll. sym_table provides a sort of dictionary for parsing by forming the link between the recognized tokens and the grammar definitions of expressions in my implementation of MiniML:

```
(">", GREATERTHAN);
("/", DIV);
```

With this, the only thing left was to go into expr.ml and evaluation.mll and update my helper functions to deal with the new operations. As aforementioned, this step was quite simple as the changes required were contained within a handful of smaller helper functions rather than the larger, essential functions. In addition, I also abstracted away the Div operator's differing functionality on ints vs. floats to simplify the user's experience.

Bibliography

[1] Cornell CS 3110: Lecture 8. Closures,
https://www.cs.cornell.edu/courses/cs3110/2014fa/lectures/8/lec08.pdf

[2] Real World Ocaml: Chapter 16. Parsing with OCamllex and Menhir,
https://realworldocaml.org/v1/en/html/parsing-with-ocamllex-and-menhir.html