CS 51: Final Project

Albert Qi

May 4, 2022

Table of Contents

1. Introduction

I implemented three extensions for this project: (1) floats, strings, and units, (2) lexically scoped environment semantics, and (3) lazy expressions. Each of the following sections will break down how I implemented a certain extension and provide examples demonstrating its behavior.

2. Floats, Strings, and Units

I began by implementing floats, strings, units, and their related operators. I added a token in `miniml_parse.mly` for each data type and operator. The appropriate grammar, keywords, and symbols were extended in `miniml_parse.mly` and `miniml_lex.mll` when necessary.

Floats

I define the float type in `miniml_lex.mll` as `let float = digit+ '.' digit*` . Thus, a float is recognized as a set of one or more digits followed by a `.` and ending with a set of zero or more digits. Furthermore, the float data type is defined as a `Float of float` in the `Expr.expr` variant.

Extending the interpreter to include floats is as simple as extending the parser and the `Unop` and `Binop` match cases. Note that extensions to the `Unop` and `Binop` match cases are written such that the language that is strongly typed.

```
match b, eval e1 env |> extract, eval e2 env |> extract with
...
| Plus, Num n1, Num n2 -> Num (n1 + n2)
| PlusFloat, Float n1, Float n2 -> Float (n1 +. n2)
```

Any type mismatches will raise an error.

```
<== ~-1. ;;
xx> evaluation error: invalid unop
<== 1 +. 1 ;;
xx> evaluation error: invalid binop
```

With this implementation, I can now perform basic arithmetic using floats.

```
<== 5. +. 7. ;;
==> 12.
<== 0.5 -. 2.5 ;;
==> -2.
<== 3. *. 4. ;;
==> 12.
<== 0.2 /. 20. ;;
==> 0.01
<== 5. ** 7. ;;
==> 78125.
```

Note that `**` is right associative, unlike the other operators.

```
<== 3. ** 1. ** 4. ;;
==> 3.
```

Moreover, I can perform many unary operators that are commonly associated with floats.

```
<== ~-.7. ;;
==> -7.
<== log (2.71828 ** 10.) ;;
==> 9.99999327347
<== sin (3.14159 /. 2.) ;;
==> 0.999999999999
<== cos (0.) ;;
==> 1.
<== tan (3.14159 /. 4.) ;;
==> 0.999998673206
```

Note that the `log` keyword represents the natural log as opposed to the base 10 log.

Strings

I define the string data type in `miniml_lex.mll` as `let string = '\"' [^ '\"']* '\"'` . Thus, a string is recognized as a set of zero or more characters that are not a `"` in between a set of double quotes. Furthermore, the string data type is defined as a `String of string` in the `Expr.expr` variant.

The most basic of string operators are `PrintString` and `PrintEndline` , both additional tags to the `Unop` variant. They can be called with the `print_string` and `print_endline` keywords, respectively.

```
<== print_string "hello" ;;
hello==> ()
<== print_endline "goodbye" ;;
goodbye
==> ()
```

Notice that `print_string` prints the input on the same line, whereas `print_endline` adds a new line to the input.

Moreover, I define the `Concatenate` tag under the `Binop` variant to allow for string concetenation. This is represented in the parser as the `^` symbol.

```
<== "a" ^ "b" ^ "c" ;;
==> "abc"
```

Note that string concatenation is right associative, as defined in `miniml_parse.mly` .

```
%right CONCATENATE
```

Units

While units might initially seem insignificant, they allow for the delaying of computation, sequencing, and side effects. The unit data type is defined in `miniml_lex.mll` as the symbol `()` and is defined as `Unit` in the `Expr.expr` variant.

I can easily create a unit through the `()` symbol.

```
<== () ;;
==> ()
```

More can be done with units, though. To pass units into functions, I define a unit function as `FunUnit of expr * expr` in the `Expr.expr` variant. With this, functions can now take units as their arguments.

```
<== let f = fun () -> 7 in f () ;;
==> 7
```

This functionality is crucial as it allows for the delaying of computation.

```
<== let rec forever = fun x -> 0 + forever x in
    let f = forever 0 in if true then 7 else f ;;
Fatal error: exception Stack overflow
<== let rec forever = fun x -> 0 + forever x in
    let f = fun () -> forever 0 in if true then 7 else f () ;;
==> 7
```

In the first expression above, `forever 0` is evaluated immediately when it is bound to `f` , thus resulting in a stack overflow error. In the second expression, however,

`f` is defined as a unit function that returns `forever 0` when applied. Because the body of `f` is not evaluated until the function itself is applied and because `f ()` is never called, there is ultimately no stack overflow error.

Moreover, I define `;` as the sequencing operator, evaluating both its left and right expressions but only returning the value of the latter. A sequence is defined as a `Sequence of expr * expr` in the `Expr.expr` variant.

```
<== (); 7 ;;
==> 7
<== print_endline "hello world!"; 7 ;;
hello world!
==> 7
```

This implementation allows for side effects. In the second expression above, the value returned is `7` but `"hello world!"` is printed as a side effect. These side effects are very important since they would be heavily utilized if the implementation were to be extended even further to allow for `refs` and mutable storage.

Additional Operators

I also extended the implementation to include more comparison operators, namely `<>`, `>`, `<=`, and `>=`.

While they may be used across multiple data types, they require both sides to be of the same data type.

```
match b, eval e1 env |> extract, eval e2 env |> extract with
| GreaterThan, Num n1, Num n2 -> Bool (n1 > n2)
| GreaterThan, Float n1, Float n2 -> Bool (n1 > n2)
| GreaterThan, String n1, String n2 -> Bool (n1 > n2)
```

This addition allows for more freedom in comparing expressions.

3. Lexically Scoped Environment Semantics

Furthermore, I extended the interpreter to allow for lexically scoped environment semantics. It is identical to the dynamically scoped interpreter with the exception of the `Fun`, `FunUnit`, `App`, and `Letrec` match cases.

The lexically scoped interpreter works as follows. When a `Fun` or `FunUnit` is passed into `eval_l`, I simply close the expression and the environment into an `Env.Closure`.

```
match exp with
...
| Fun _ | FunUnit _ -> Env.close exp env
```

Closures allow the interpreter to keep the environment in which a function is defined and not in which it is applied. I can then deconstruct these closures in the `App` match case.

```
match exp with
...
| App (e1, e2) ->
    match eval_l e1 env, eval_l e2 env with
    | Env.Closure (Fun (v, e), closure_env), repl_exp ->
      let ext_env = ref repl_exp |> Env.extend closure_env v
      in eval_l e ext_env
    | Env.Closure (FunUnit (Unit, e), closure_env), Env.Val Unit ->
      eval_l e closure_env
    | _ -> raise (EvalError "invalid app")
```

In the code above, I utilize one match case for `Fun` and another for `FunUnit`.

In the case for `Fun`, I begin by extending the closure environment to map the variable `v` to the expression over which the function should be applied. I then evaluate the expression in the function itself under the new environment.

Because there is no variable `v` in a unit function, I need not extend the environment; instead, I can simply evaluate the expression `e` in the closure environment `closure_env`.

Now, all that remains is the `Letrec` match case. However, unlike in dynamically scoped environment semantics, recursion is not given "for free". Rather, I utilize the mutability of the environment and write the `Letrec` match case as follows.

```
match exp with
...
| Letrec (v, e1, e2) ->
    let ref_v = ref (Env.Val Unassigned)
    in let ext_env = Env.extend env v ref_v
    in let def_e1 = eval_l e1 ext_env in
    if def_e1 = Env.Val Unassigned then raise (EvalError "invalid letrec")
    else ref_v := def_e1; eval_l e2 ext_env
```

I use the following steps for evaluating `let rec v = e1 in e2` : 1. I extend the environment to include a binding of `v` to `Unassigned` . This reference is stored as `ref_v` and the extended environment as `ext_env` . 2. I evaluate `e1` in this extended environment `ext_env` to get some `def_e1` . 3. If `def_e1` is not `Unassigned` , I can then mutate `ref_v` so it maps to `def_e1` . 4. I finally evaluate `e2` in the extended environment `ext_env` and return the result.

With these four new match cases for `Fun` , `FunUnit` , `App` , and `Letrec` , I get the benefits of environment semantics without the drawbacks that come with a dynamically scoped interpreter.

Take this simple curried function, for example. In lexically scoped environment semantics, the expression evaluates to `12` .

```
<== let f = fun x -> fun y -> x + y in f 5 7 ;;
==> 12
```

Yet, in dynamically scoped environment semantics, the expression fails to evaluate at all!

```
<== let f = fun x -> fun y -> x + y in f 5 7 ;;
xx> evaluation error: varid not found
```

Lexically scoped environment semantics are crucial because they open up more possiblities regarding further extensions to the interpreter; for example, they allow for `refs` while still maintaining the lexical structure of the program.

4. Lazy Expressions

Lastly, I extended the implementation to include lazy expressions. Lazy expressions are defined as `Lazy of expr ref` in the `Expr.expr` variant with `Force` as an added tag to the `Unop` variant. Note that I use references since it allows for memoization.

Prefixing an expression with the `lazy` keyword creates a lazy expression, and I can force a lazy expression by calling `force` .

```
<== force (lazy 7) ;;
==> 7
<== force (force (lazy (lazy 7))) ;;
==> 7
```

The evaluator thus works as follows. Evaluating a lazy expression simply returns a closure of itself and its environment.

```
match exp with
...
| Lazy _ -> Env.close exp env
```

I utilize closures because I ultimately want the environment in which the lazy expression is defined and not in which it is forced. Take the following expression, for example.

```
<== let x = 7 in let y = lazy x in let x = 0 in force y ;;
==> 7
```

The expression above evaluates to `7` under the current implementation. If closures were not utilized, however, the expression would have evaluated to `0` instead, more similar to an implementation involving dynamically scoped environment semantics.

The closure is only deconstructed when the lazy expression is forced.

```
match exp with
...
| Unop (u, e) ->
    (match u, eval_e e env with
     | Force, Env.Closure (Lazy e, env_closure) ->
       let res = eval_e !e env_closure
       in e := extract res; res
     | Force, _ -> raise (EvalError "invalid lazy")
     | _ -> eval_all eval_e exp env)
```

In the code above, I begin by evaluating the lazy expression in the closure environment. Then, I mutate the expression so it maps to the result and return the result afterward. This means the next time the expression is forced, I need not reevaluate the entire expression; and thus, memoization is achieved.

Delaying Computation and Memoization

Recall the `forever` function from section `2. Floats, Strings, and Units`.

```
<== let rec forever = fun x -> 0 + forever x in
    let f = forever 0 in if true then 7 else f ;;
Fatal error: exception Stack overflow
<== let rec forever = fun x -> 0 + forever x in
    let f = fun () -> forever 0 in if true then 7 else f () ;;
==> 7
```

I could delay the `forever` function by attaching it to the body of a unit function. Now, however, the same delaying of computation can be achieved through lazy expressions.

```
<== let rec forever = fun x -> 0 + forever x in
    let f = lazy (forever 0) in if true then 7 else force f ;;
==> 7
```

The current implementation of lazy expressions allows for more than just the delaying of computation, though. Unlike unit functions, lazy expressions allow for memoization, eliminating the need to reevaluate an expression every time it is called.

```
<== let f = fun () -> print_endline "running..." in f (); f() ;;
running...
running...
==> ()
<== let f = lazy (print_endline "running...") in force f; force f ;;
running...
==> ()
```

This allows for a lot more efficiency over a non-memoized implementation; and if the interpreter were to be extended even futher, these capabilities even allow for the addition of infinite streams.