CS51 Final Project (MiniML) Write-Up

Dinesh Vasireddy
May 1, 2024

I added **four extensions** to MiniML for my final project:

- **Lexical** Environment Semantic Evaluator
- **Float** Data Type
- **String** Data Type (Including Concatentation)
- **Lists** (Including Cons and Append Operations)

Lexical evaluation

As detailed in the ReadMe, I introduced a Lexical Environment Semantic Evaluator (eval_l in evaluation.ml (http://evaluation.ml)) that adheres to lexical environment scoping rules. Unlike dynamic scoping, lexical scoping uses the environment in place at the time functions are defined, not when they are called. This is crucial for supporting closures effectively, allowing functions to carry with them the environment of their definition.

This evaluator modifies the handling of function applications and variable bindings. For instance, when a function is defined, it captures the current environment in a closure, and this environment is used later when the function is called, regardless of the current scope at the call site.

As I coded this lexical evaluator, I also decided to abstract out the common logic between environment semantics evaluators (dynamic and lexical)into a separate function called eval_env, which exhibited different behaviors based on the environment semantics type (which I specified using a separate algebraic data type).

Float data type

```
==> Float(2.)
<== 2. +. 3.;;
==> Float(5.)
<== 2. -. 3.;;
==> Float(-1.)
<== 2. *. 3.;;
==> Float(6.)
<== 2./.3.;;
==> Float(0.666666666667)
<== 2. + 3.;;
xx> evaluation error: int operation on float
<== 2./.0.;;
xx> evaluation error: Division by zero
```

I also added the Float data type to MiniML, along with specific float operations (Fplus, Fminus, Ftimes, Fdivide). This was a relatively simple extension as it simply required an adaptation of the existing logic for integers for the simple binop operations.

This extension required modifications to the expr data type, the available binop operations, and the evaluators to handle operations involving floats correctly as shown above, specifically ensuring that the binop float operations correctly apply to floats and not integers. I also

checked for the Division by Zero edge case and designed the extension such that an EvalError is thrown when that invalid operation occurs.

String data type

```
<== "hello";;
==> String(hello)
<== "hello" ^ " world" ^ "!";;
==> String(hello world!)
<== "hello" ^ 3 ;;
xx> evaluation error: binop on non-compatible types
```

Similar to the Float data type, I also added a String data type to MiniML. However, since Strings don't use similar binop operations to integers and floats, I went ahead and added the concatenating feature (^) as demonstrated above.

Handling strings required adjustments in both the parser miniml_parse.mlly (adding symbols for concatentation and quotation mark recognition for strings) and evaluation.ml (http://evaluation.ml), ensuring that string literals are correctly interpreted and that string-specific operations (only concatenation in this case) are correctly executed.

List data type

I also extended MiniML by implementing the List data type, along with list manipulation operations like cons and appending. This was one of my more complex extensions as it required modifications of the expression syntax, evaluation logic, and parsing mechanisms.

In the expr.ml (http://expr.ml) file, I expanded the expr type to include new constructors: List for representing lists of expressions, ListCons for the cons operation (prepending an element to a list), and ListAppend for concatenating two lists. In evaluation.ml (http://evaluation.ml), I adapted the evaluators to handle these new list operations. The evaluator for List evaluates each expression in the list, collecting the results. For ListCons, it ensures the second expression evaluates to a list before prepending the evaluated first expression, and for ListAppend, it concatenates two evaluated lists if both expressions evaluate to lists.

```
<== [1;2;3];;
==> List(Num(1), Num(2), Num(3))
<== 1 :: [2;3];;
==> List(Num(1), Num(2), Num(3))
<== [1;2;3] @ [4;5];;
==> List(Num(1), Num(2), Num(3), Num(4), Num(5))
```

Furthermore, I modified miniml_parse.mly to include parsing rules for these list operations, recognizing new syntax such as brackets for list literals and handling tokens for list operations (CONS (::) and APPEND_LST(@)). In miniml_lex.mll, I defined these tokens and adjusted the lexer to recognize list-related syntax, ensuring that list operations are correctly tokenized during lexical analysis.