CS51 Final Project Report

Jared Hu Ni

May 4, 2022

CS51: Abstraction and Design in Computation

1 Introduction

In the CS51 final project, I was given the opportunity to implement an interpreter for the MiniML language, a functional subset of the OCaml language that has both concrete and abstract syntax and is capable of evaluating basic functional operations. It has truly been a fun project to work on: from implementing the helper functions in expr.ml to the various evaluators, I have had moments of excitement and periods of excitement, but I had never felt bored. Though I have learned a lot through each and every stage of this conceptually challenging yet rewarding project, I feel especially compelled to share with you my personal extensions to MiniML language, as I have completed for stage 8 of the project.

My first extension was the added support for unit, float, and string types. The original MiniML source code only included support for a limited number of atomic types, including integer, variables, booleans, and complex types such as functions, binary and unary operations. in a real programming language, there are much more atomic types that the programmer can work with to perform various tasks. So, I had decided to make it my goal to extend upon the basics and include some of the most fundamental atomic types that define a modern programming language into MiniML. Using OCaml's algebraic data types, I had successfully implemented unit, float, and string types, extended the unary and binary operations to accommodate for these new types (such as string concatenation and float operations), and went through rounds of intensive testing to make sure the interpreter performs the way that I have intended on these new types.

My second extension was implementing the lexical evaluator for the interpreter. The project only required us to complete the substitution and the dynamic evaluators, but neither of them fully models the way that an actual OCaml interpreter works. The OCaml language, in its elegance, utilizes the lazy nature of the dynamic environment, yet outputs results that are identical to the substitution model because it has lexical environment semantics. Implementing and testing the lexical evaluator in MiniML has given me a better grasp of how the lexical environment functions, and it has been, by far, one of the most engaging topics I had the opportunity to learn this semester.

2 Implementation of Atomic Types

In this section, I will explain in detail how I added support for the float, string, and unit types in MiniML. On the surface, the new types are simply extended as additional algebraic data types in type expr, and the operations performed on them are additional algebraic data types in type binop and type unop. In type expr of the expr.ml file, I added Float, String, Unit, and FunUnit (for functions that take in unit); in type binop, I added Concat (for concatenating two strings), FloatPlus, FloatMinus, and FloatTimes (for operating on floats). In type unop, I added FloatNegate (to negate a float). To make the interpreter recognize and understand these new types, we must modify the MiniML Lexical Analyzer (miniml_lex.mll) and MiniML Parser (miniml_parse.mly) files.

In the lexical analyzer, keywords of the language are defined in the keyword_table, and symbols are defined in the sym_table. the sym_table is where I matched the concrete syntax of new operators to their parser-recognizable abstract data type counterparts (notice how these are capitalized versions of the algebraic data types given in expr.ml). These capitalized types correspond to the tokens defined in the parser, where the grammar rules for using and operating on these data types are mapped to their corresponding algebraic data types, outputting the abstract syntax defined using these grammar rules when declaring or performing an operation. For example, FLOAT carries a float type, so its corresponding abstract syntax would be Float $1 (where $1 is a placeholder signifying the place that the data is located in the grammar expression). exp FLOATPLUS exp corresponds to Binop(FloatPlus, $1, $3), where $1 is the first exp (expression) and $3 is the second exp.

In the lexical analyzer, a new rule token must be added for each of the new atomic types (except unit) to recognize and process the syntactic pattern and structure using OCaml's regular expressions (regex), so that the new type can be recognized by the interpreter. For example, in the given source code, an integer is recognized as anything composed of digit numbers 0 through 9 and has a length of at least 1. Its rule token recognizes the said pattern and convert the string to an int type. The finished product, recognized as an INT, can then be used to evaluate expressions given the grammar rules defined in the parser. Extending upon the example, I developed the rule tokens to process and recognize floats and strings; using regex, a float is recognized as repeated digits like that of the integer before and after the symbol ".", a decimal dot. A string is the repetition of any characters except double quotes and double-backslash within a pair of double quotes. I then had to process string to remove all double-quotes in the rule token, so the double quotes disappear in the event of

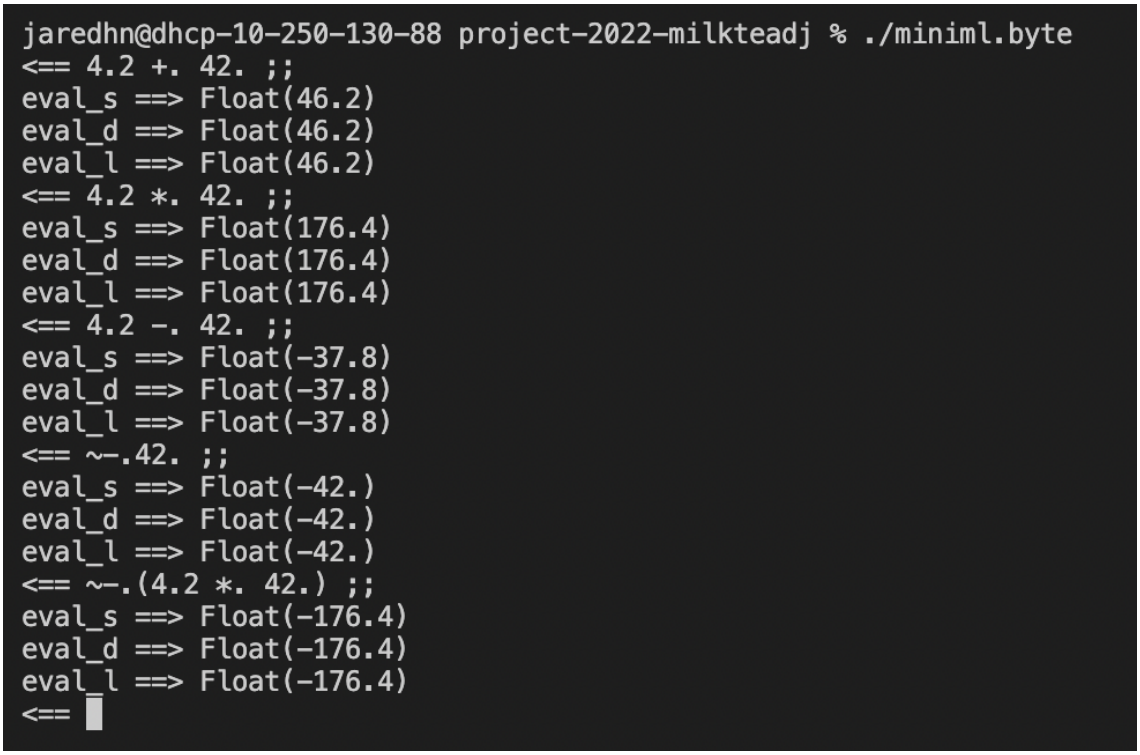
string concatenation, and they reappear in the concrete and abstract syntax if printed.

After the lexical analyzer and parser are able to accommodate the new atomic types and their operations, I extended the case match in every function of expr.ml and the evaluation.ml to include these new types and the behaviors of using their binary and unary operations. This process is very similar to implementing the functional behavior of the other atomic data types and operations given in the source code, so I will not bore you with the details.

3 Demonstration of the Atomic Types

The following demonstrations are the behaviors performed by my MiniML interpreter, which I have modified to display the interpretation of all three evaluators that I have implemented: eval_s for substitution, eval_d for dynamic environment, and eval_l for lexical environment.

Here are some mathematical operations that utilize the float type. In the picture below, float 4.2 and float 42. are added (+.), multiplied (*.), subtracted (-.), and negated (-.) in the MiniML interpreter in using the three evaluators implemented.



Float operations in the MiniML interpreter

Here are some string concatenation operations. In the first example, a function called f takes in the string "42 is" as the argument, and the argument is concatenated to " 42" and outputs the string "42 is 42". In the second example, three sub-strings are concatenated to construct the sentence, "abstraction is its own reward".

```
jaredhn@dhcp-10-250-130-88 project-2022-milkteadj % ./miniml.byte
<== let f = fun x -> x ^ " 42" in f "42 is" ;;
eval_s ==> String("42 is 42")
eval_d ==> String("42 is 42")
eval_l ==> String("42 is 42")
<== "abstraction is" ^ " its own" ^ " reward" ;;
eval_s ==> String("abstraction is its own reward")
eval_d ==> String("abstraction is its own reward")
eval_l ==> String("abstraction is its own reward")
<==
```

String operations in the MiniML interpreter

Here are operations using unit type. the unit type can be printed on its own, or it can be used as a function argument in FunUnit functions that take in a unit as its argument.

```
jaredhn@dhcp-10-250-130-88 project-2022-milkteadj % ./miniml.byte
<== () ;;
eval_s ==> Unit
eval_d ==> Unit
eval_l ==> Unit
<== f () ;;
xx> evaluation error: unbound variable
<== let f = fun () -> 420 in f () ;;
eval_s ==> Num(420)
eval_d ==> Num(420)
eval_l ==> Num(420)
<==
```

unit type in the MiniML interpreter

4 The Lexical Environment Evaluator

My next extension is the lexical environment evaluator, eval_l. I started off by following the instructions in the textbook and copied the code for my eval_d into eval_l. Luckily for me, I have already abstracted away the unop and binop operations into their own functions for all three evaluators, so it was not too much code to deal with.

I then changed the match for Fun (function) and FunUnit (unit function) to return Env.Closure instead of Env.Val, so the state of the environment that the function evaluates using can be preserved where the function is defined. Then, based on the lexical function application rule in the textbook, I changed the return for function applications. Then, I modified the recursive definition of the lexical evaluator by using imperative reference to Unassigned type and reference reassignment, exactly as it would logically map to the direction given in stage 8 of the project in the textbook.

Here is a demonstration of the differences between the evaluators. Lexical evaluator operations should match that of the substitution semantics in output, though it is implemented using environment semantics like that of the dynamic evaluator. A program that yields different results between dynamic and lexical scopes would be reassigning a variable after it is used in a declared function, like in the following:

```
jaredhn@dhcp-10-250-130-88 project-2022-milkteadj % ./miniml.byte
<==
let x = 42 in
let f = fun y -> y - x in
let x = 420 in
f 10 ;;
eval_s ==> Num(-32)
eval_d ==> Num(-410)
eval_l ==> Num(-32)
<==
let x = 1 in
let f = fun y -> y * x + x in
let x = 42 in
f 2 ;;
eval_s ==> Num(3)
eval_d ==> Num(126)
eval_l ==> Num(3)
<==
```

outputs of dynamically-scoped evaluator can differ from lexically-scoped evaluators