CS51 Final Project (MiniML) Write-Up

Dinesh Vasireddy *May 1, 2024*

Hello CS51 TFs👋

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

- **Lexical** Environment Semantic Evaluator
- **Float** Data Type
- **String** Data Type
- **Lists** with Type Safety

I also added a few minor functionality adjustments, such as adding greater than (>) and not equals (<>) comparison features for the appropriate data types. However, I wouldn't consider these extensions, but rather some helpful quality of life features.

Lexical Evaluation

In the implementation of the Lexical Environment Semantic Evaluator (`eval_l in evaluation.ml`), I focused on adhering to the principles of lexical scoping, where the environment at the time of function definition is used during function calls, rather than the environment at the call site. This approach is essential for the correct handling of closures, ensuring that functions encapsulate and utilize the environment from their point of definition.

To achieve this, I modified the handling of function applications and variable bindings significantly. When a function is defined, it is now wrapped into a closure along with its defining environment. This closure is then used when the function is called, ensuring that the original environment is preserved and used for evaluation, regardless of the current execution context.

The core of this functionality is encapsulated in the `eval_env` function, which I designed to handle expressions based on the specified environment type—dynamic or lexical. For lexical scoping, the function captures and uses the environment at the point of function definition. This is facilitated by the `Env.close` method, which creates a closure from a function expression and its environment.

Furthermore, I introduced a type distinction (`envspec`) that allows `eval_env` to adjust its behavior based on whether dynamic or lexical scoping is required. In the case of lexical scoping (`Lexical`), when evaluating a function application, the environment stored within the closure is used. This ensures that the function operates with the variables and bindings that were available at the time of its definition, not at the time of its invocation.

This implementation not only supports higher-order functions but also enhances the modularity and predictability of code by adhering to the lexical scoping rules. It was crucial to ensure that all parts of the environment, including function definitions and variable bindings, are correctly managed and consistent across different parts of the program.

By abstracting the common logic for environment handling into `eval_env` , I was able to streamline the implementation of both dynamic and lexical evaluators, reducing redundancy and improving maintainability. This abstraction also simplifies the extension of the evaluator for additional features or optimizations in the future.

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
<== 2. ** 3.;;
==> Float(8.)
```

In my extension of MiniML, I introduced the Float data type to handle floating-point arithmetic, which is crucial for applications requiring precision beyond integers.

In `expr.ml`, I expanded the `expr` type to include `Float of float`, allowing the representation of floating-point numbers directly in the syntax tree. This change required updates to the string representation functions to correctly display floats in both concrete and abstract syntax forms.

In `miniml_lex.mll`, I defined rules to recognize floating-point numbers, distinguishing them from integers. This involved handling optional fractional parts and exponents in the lexer. The lexer now correctly identifies and converts floating-point literals into the `FLOAT` token, which is passed to the parser. In `miniml_parse.mly`, I added parsing rules to handle the `FLOAT` token, ensuring that floats are correctly parsed into the `Float` expression type. I also introduced float specific binary operators (`FPLUS`, `FMINUS`, `FTIMES`, `FDIVIDE`, `POWER`) to handle arithmetic operations specific to floats.

In `evaluation.ml`, I extended the evaluation functions to handle operations on floats. This included defining behavior for the new floating-point binary operators, ensuring that simple evaluation operations (plus, minus, etc.) are performed correctly on floats. Special care was taken to handle edge cases such as division by zero, which raises an `EvalError`. I also implemented error handling cases when float operations were executed on non-float data types or non-float operations were executed on a float data type to ensure type safety with float computation.

String data type

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

To enhance MiniML's capabilities, I implemented the String data type, enabling the manipulation and comparison of text.

In `expr.ml`, I introduced `String of string` within the `expr` type to represent string literals directly in the syntax tree. This addition necessitated updates to functions handling string representations to ensure strings are correctly displayed in both concrete and abstract syntax forms.

In `miniml_lex.mll`, I defined rules to recognize and correctly tokenize string literals, encapsulated by quotation marks. This setup ensures that string values are accurately captured and passed to the parser as the `STRING` token. In `miniml_parse.mly`, I incorporated parsing rules for the `STRING` token, ensuring that string literals are parsed into the `String` expression type. Additionally, I introduced the `CONCAT` operator to handle string concatenation (`^`).

In `evaluation.ml`, I extended the evaluation functions to handle string-specific operations. The primary operation, string concatenation (`CONCAT`), involves checking that both operands are strings before concatenating them, raising an `EvalError` for type mismatches.

To facilitate richer interactions with strings, I also implemented comparison operations for strings, such as `Equals`, `NotEquals`, `LessThan`, and `GreaterThan`. These operations allow for direct comparison between string literals, broadening MiniML's type system and also improving its applicability for constructing expressions involving string data.

Lists with Type Safety

I also extended MiniML by implementing the List data type (along with list manipulation operations) with type protections.

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

In `expr.ml`, I introduced a new constructor `List of expr list` to the `expr` type to represent lists of expressions. This addition required updates to functions handling string representations to ensure lists are correctly displayed in both concrete and abstract syntax forms.

In `miniml_lex.mll`, I defined tokens for list-specific syntax such as brackets (`[`, `]`) and list operations (`::` for cons and `@` for append). This setup ensures that list structures and operations are accurately recognized and tokenized. In `miniml_parse.mly`, I incorporated parsing rules for these tokens, ensuring that list literals and operations are parsed into the appropriate `expr` types. This includes handling empty lists and the use of list-specific operations within expressions.

In `evaluation.ml`, I extended the evaluation functions to handle list-specific operations. This includes evaluating each expression within a list, handling the `ListCons` operation to prepend an element, and the `ListAppend` operation to concatenate two lists. I implemented type checks to ensure all elements within a list are of the same type, enhancing type safety.

Type Inference and Safety

I introduced a function `check_list_type` that verifies all elements in a list share the same type and ensures type consistency within lists by throwing an `EvalError` when list definitions have conflicting types. This function is used during list creation and all list-related operations, including cons and append, to prevent type errors in list manipulation. This was one of the most important parts of my list extension as it was necessary to prevent invalid data being stored in lists, which would have gone against the basic rules of OCaml (and thus MiniML).

Handling of Empty Lists

Special handling for empty lists was added to ensure operations like concatenation and cons behave correctly when applied to or with empty lists. This includes returning the non-empty list during appending (list concatentation) if one of the lists is empty.

List Comparison

I also ensured that the list implementation supported comparison operations (`Equals`, `NotEquals`, `LessThan`, `GreaterThan`) to allow list structures to be compared lexicographically, similar to strings (as it works in OCaml). This feature could be useful for conditional expressions and assertions within MiniML programs.

Final Thoughts

Overall, I had a great experience with this MiniML project. I gained a much better understanding of different semantics rules and found it exciting to implement nuances of a programming language by hand. I also feel that all of my extensions added a nice layer of usability to MiniML and expanded the scope of the project in a way that I found meaningful.

Thank you all for making CS51 a great course✌ !