CS51 Final Project Extensions

Ryan Wallace

April 27, 2016

As an extension to my implementation of the miniml language, I focused primarily on the addition of a lexically scoped environment semantics to the language. In addition, I implemented minor improvements to the user interface. These extensions and the processes of implementing them are described below in turn.

Lexically Scoped Environment Semantics

A lexically scoped environment semantics was the major focus of the extensions to the miniml language. In contrast to OCaml and the majority of modern programming languages, the environment semantics described in Part 4 of the problem set manifest a dynamic scoping. In many cases, this scoping method is more difficult to reason about because the value assigned to a variable at a given point can not be determined by looking backwards in the code, but instead depends on the state of program when the variable is referenced. Nevertheless, the environment semantics model is useful for the naturalness of its implementation. Therefore, it is desirable to extend the environment semantics to allow for lexical scoping, as well.

When implementing the lexical scoping feature, it was first important to realize that the evaluation function for the lexically scoped and dynamically scoped versions of the environment semantics would be very similar. Thus, rather than implement each function separately, it was preferable to factor out the commonalities between the two implementations. Because only three total changes appear in the code (for the evaluations of variables, functions, and applications), the preferable way to achieve this is to write a more general function, eval_method, that is parameterized by the scope of evaluation to be done. eval_method can evaluate an argument expression in its argument environment with either dynamic or lexical scope, depending on the value of the s parameter.

The eval_method function was originally implemented with type expr → Env.env → bool → Env.value, with the bool representing whether the expression should be with lexical scope or not, that is, with a lexical or dynamic scope. However, after consideration, this implementation was changed to a function with type expr → Env.env → scope → Env.value, where scope is a newly defined variant type with the three value constructors L (referring to lexical scoping), D (referring to dynamic scoping), and S (referring to substitution model). [1] This design change was made in consideration of future extensibility of the language. For instance, if in the future we desired to implement a new type of scope, it would be as simple as including a new type constructor in the scope variant, and matching against the new constructor in the eval_method function with the appropriate results. Whereas with the previous implementation using a bool, only two types of scope can be represented. An additional design decision that was made here was to write an auxillary recursive function inside the definition of the eval_method function with type expr → Env.env → Env.value, dropping the scope parameter, as it stays constant throughout a call to eval_method, reducing the chance of the recursive method being called incorrectly and reducing clutter.

Now, to implement eval_d and eval_l, it suffices to call eval_method with the same input expression and environment, and with the corresponding scope constructor (D and L, respectively). It should be noted that this implementation is extensible to other evaluation flavors. However, eval_s was kept as a separate function because it differs substantially from the environment model implementation, and would result in a very cluttered eval_method function, with matching to the scope constructor in each match case of the expression.

[1] Although the S type constructor is not used in the eval_method function, it is included for completeness and because the same scope variant type is used in the test functions, where all three type constructors are used.

User Interface Changes

Error Handling

As supplied, the skeleton code for miniml results in termination of the program upon an evaluation error exception (EvalError or EvalException) being thrown. This is not ideal because it forces the user to call ./miniml.ml again to re-open the program before re-entering a properly formed expression. Rather than this cumbersome experience, it would be preferable to display the exception message triggered, then allow the user to input another expression, keeping the program running, and removing the need to re-open the program.

This change was implemented simply by adding two additional match cases in the with clause of the try/with construct for the results of the evaluation of the input expression in the file miniml.ml. In the case of evaluate returning an EvalError exception, the "with" statement now matches against that pattern, and prints the associated error message, without terminating the program. In the case of evaluate returning an EvalException, the "with" statement matches against that pattern as well, and prints a nicely formatted Evaluation Exception message.

Result Display

As described in the problem set specification, the result of an evaluation printed to the screen is the string representation of the abstract syntax tree for the result of the evaluation of the expression. Displaying the abstract syntax tree makes interpreting the result very difficult, especially if the resultant expression is more complex than a single literal. Ideally, we would instead print a natural, human-readable expression.

This change was implemented while respecting the type signature and specification of the exp_to_string function provided by implementing a new function, exp_to_string_f. exp_to_string_f takes an expression (expr) and a bool as arguments. The expression is the expression to be made into a string, and the bool represents whether or not the expression in question should be "pretty-printed". With this, we can implement the original exp_to_string function, maintaining its type signature, by calling exp_to_string_f with the bool set to false. Now, exp_to_string returns the expression printed as an abstract syntax tree as described in the specification (and passing unit tests), but the exp_to_string_f function with the bool true can be used when printing the results to the screen. Importantly, this implementation avoids copying the code for printing, slightly modified, in both functions, and instead abstracts the match statement into the exp_to_string_f function.