CS51 Final Project Writeup

Joey Bejjani

May 3, 2023

For my MiniML extension, I implemented an evaluator using the lexically-scoped environment model (number 2 in the extension ideas list in the readme). I did this by first copying the `eval_d` function definition and pasting into the `eval_l` function definition, since they share a lot of the same semantics rules. I then modified the copied code to manifest lexical scoping. There were three modifications, which are roughly as follows: first, the evaluation of a function returns a closure with the function and the current environment instead of just the function as in `eval_d`; second, function application evaluates the body of the function in the environment from its updated closure; third, evaluation of a `let rec` expression involves binding the variable to `Unassigned` and later mutating this mapping. The resulting `eval_l` function passed both my generic `eval_tests` (which includes expressions whose evaluations dynamic and lexical semantics models should agree on) and my `eval_lexical_tests` (which includes expressions whose evaluations differ depending on whether lexical or dynamic semantics rules are used); however, `eval_d` and `eval_l` contained a lot of redundant code as a result of the copy and pasting. I factored out the shared code using the modular programming paradigm. I created an `ENV_SCOPE` type that would package together the three differences between the semantics rules of the dynamic and lexical environment models. I then defined two modules of this type, `EnvDynamic` and `EnvLexical`, which implement the evaluation of function expressions, application expressions, and `let rec` expressions according to the appropriate semantics rules. I then defined a functor `MakeEnvEvaluator` satisfying an `ENV_EVALUATOR` type and taking an `ENV_SCOPE` as an argument. The functor returns a module (an `ENV_EVALUATOR`) consisting of the recursive `eval` function. This function evaluates the argument expression inside the argument environment, calling the `f`, `app`, and `letrec` functions defined in the argument `ENV_SCOPE` module in order to evaluate function, application, and `let rec` expressions, respectively. The `MakeEnvEvaluator` functor thus returns a module whose `eval` function evaluates expressions according to the scope manifested in the definition of its argument `ENV_SCOPE` module. I then applied `MakeEnvEvaluator` to `EnvDynamic` to get the module whose `eval` function is equivalent to `eval_d`, and then again to `EnvLexical` to get the module whose `eval` function is `eval_l`. The two evaluators obtained using this abstracted approach passed all the same tests that they passed in the redundant version. Importantly, `eval_s` and `eval_l` agree on the evaluations of the expressions in the `eval_lexical_tests`, verifying that they both manifest lexical scoping.

I also added the `string` atomic type along with the `Concat` (string concatenation) binary operator as a second extension. I did this by extending the definition of the `expr` type to include `String of string` and extending the `binop` type to include `Concat`. I updated `expr.mli` to reflect these type definition changes. In `free_vars` and `subst`, I added a match case to handle the new `String` expression. I extended `exp_to_concrete_string`, `exp_to_abstract_string`, and my `string_of_binop` function to handle `Concat` and `String`. In my `binop_eval` function, I raised `EvalError`s when anything other than two strings are being concatenated, and also when algebraic operators are applied to strings. To extend the language to include strings and concatenation, I extended the parser with the following steps. In `miniml_lex.mll`, I added `("^", CONCAT)` as an entry in the `sym_table` hashtable. I added a new regular expression `string` that matches user input that starts with a " character and ends with a " character. A matched `string` is then parsed by stripping the quotation marks by using the `String.sub` function and returning it as a `STRING` token. I defined this new token using %token <string> STRING in `miniml_parse.mly` along with `%token CONCAT`, which I also specified as left associative using `%left CONCAT`. Then, in `expnoapp`, I specified that inputs of the form `exp CONCAT exp` parse to `Binop(Concat, $1, $3)`, where \$1 and \$3 specify the two `exp` arguments (in the first and third positions). Finally, `String` tokens parse to `String $1`, where \$1 is the actual string returned after stripping the quotation marks inputted by the user as described earlier. I wrote and successfully ran more tests to verify that strings and concatenation work as intended.