CS51 writeup

Andy Martinez

May 2023

Extensions: Floats and Their Functions

The first note is that my implementation of eval_e is done with lexical scoping. The particulars of how I implemented lexical semantics are shown under this portion on floats. Implementing eval_e was done in a lexical scope to bring miniML as close to Ocaml as possible and Ocaml uses Lexical scoping. In implementing floats I not only added the type but also the different functions for floats specifically. Doing so required the addition of a few new binary operators which I have added to the binop data type. Since Ocaml is statically typed I made sure that miniML followed this same idea and made it so that float operators were distinct from integer operators. That means that even though both could be present in the same Binop construct, the evaluator would return errors accordingly. For example, with the introduction of floats, there are 2 errors that could happen. The first is the mismatching of types. We could have a case where someone tries to add an int and a float or compare an int and float using equals or less than.

The second type-related error deals with the use the normal integer binary operators to deal with floats. All these examples should (and do in my implementation) raise an error, this is done to reinforce the statically typed behavior of OCaml. To make this clear to the user, miniML adopts OCaml's alternative operators for floats "-.", "+.", "*.", and I also added a division operator "/.". The reason division was difficult to implement before was that miniML did not support floats so if two integers did not divide evenly the interpreter would not know how to handle that case. I also added the ability to negate floats through the Unop constructor. Again, I made sure to add a new type for negating a floating type value.

Extension: Lexical Semantics

The final project started by requiring that we implement substitution semantics. While substitution semantics are pretty simple, that simplicity is also limiting in the sense that it limits the miniML language to being a pure language. One without mutable variables, and environments mostly solve this issue. That is where dynamic semantics come into play. Dynamic semantics expand the language from being a pure one to one that allows for mutable data. Nevertheless, dynamic semantics have their own issues, most notably that the output values can diverge from the results given by the substitution semantics. That's where lexical semantics comes into play. Lexical semantics removes the restrictions that come with substitution semantics but its results are still consistent with substitution semantics, unlike dynamic semantics. Ocaml uses lexical semantics and as a result, to bring the miniML language closer to Ocaml the foundation for lexical semantics is necessary. Hence why it plays a key role in my extension.

The first part of implementing lexical semantics is to establish where the rules differ from the dynamic semantics. The main difference between dynamic semantics and lexical is the use of closures. When evaluating functions we must wrap it in a closure with the environment in which the function was defined, this makes it so that functions evaluate to the same value that they do in substitution semantics. By having functions wrapped in closures we must also alter our implementation for the "App(f, argument)" case from the dynamic implementation. So I added some match statements for cases in which a function is not in a closure (return an error because all functions should be wrapped in a closure) and cases where a function is wrapped in a closure. After that, the application implementation is identical to the one in eval_d. In dynamic semantics let and let rec had identical implementations whereas in lexical semantics we can keep the same implementation for let from eval_d but we must change the let rec implementation. This is finally where the "Unassigned" type as well as references come into play. The first stage is to map the recursive variable to an Unassigned value and then update the environment to reflect this mapping. Next, we must evaluate the body of the recursive definition in the new environment, let's call this value v_d. Now that we have a value for v_d, we can use references to update the value of the recursive variable. The reference is needed to update the value of the recursive variable for all of its occurrences in the closures. The last step is to evaluate the body in this new environment. After implementing these augmentations to the dynamic semantics we have a completed lexical

interpreter.