CS51 : Final Project

Wassim Marrakchi

Wednesday, May 2, 2018

1 Introduction

For my final project, I implemented a small subset of an Ocaml-like language called MiniMl including only a subset of constructs and limited support for types without type inference. I implemented three models of MiniMl using the substitution model (eval_s), a dynamically scoped environment model (eval_d) and, finally, a lexically scoped environment model as my extension (eval_l). The substitution model was lexically scoped. It "purely" handles all the initial subset of constructs we were given. However, the model is not good enough as it is impossible to extend the language to handle references, mutability and imperative programming. To remedy to that, we chose to make use of an environment to store mapping from variables to their values.

2 Lexically vs. Dynamically Scoped

In a dynamic scope, the scope of a function's variable name is determined by the dynamic ordering in which they are evaluated. In a lexical scope, the values of variables are governed instead by the lexical structure of the program, allowing us to account for overrides. Consider:

```
let x = 1 in let f = fun y -> x + y in let x = 2 in f 3 ;;
```

In a dynamically scoped environment, when evaluation the application of f to 3, we take x as 2, the most recent assignment, instead of taking x as 1 which is the x that outscopes the body of f. In this case, a dynamic scoped environment (eval_d) would return 5 while the right answer 4 is given by the lexically scoped environment (eval_l).

3 Extension: Lexically Scoped Environment Semantics

To fix the dynamic semantics, it is important to handle to handle function values statements differently by packaging up the function being defined and a snapshot of the environment at the time of its definition into a closure so that, at evaluation time, we use the definition environment instead of the dynamic environment. To do that, I changed three parts of my dynamically scoped environment implementation: Fun, Letrec, and App.

In the case of Fun, as explained before, we want to keep track of both the function definition and the environment at the moment of definition. To do that, we added the construct Closure which only is only relevant to functions.

In the case of Letrec, we want to assign Unassighed to *id* first so that we evaluate the definition independently from the *id*'s value. From there, we can replace all the Unassigned by the evaluated definition.

App is certainly the more complicated case. We start by evaluating the function in the current environment so that it returns the closure of the function with the environment of its definition. We then evaluate the function again in an environment we extend by mapping the definition variable with the evaluation of what is passed to the function.

4 Conclusion

This project was very interesting as I learned a lot about scoping and the intricacies of interpreters. As I was trying the various abstraction techniques we've learned in the course, I realized how similar all these models were but also how important the differences are in deciding the behavior of my MiniML interepreter.