CS51 Final Project Writeup

Arthur Abrantes

May 3, 2017

Abstract

In programming languages, expressions can be evaluated in either dynamically or lexically scoped environmnets. In the first case, the expressions are evaluated in accordance with their dynamic ordering - when the expressions are called. In the second, the expressions are evaluated in accordance with their lexical ordering - when the expressions are defined. When implementing an interpreter for a programming language, it is possible to convert it from a dynamically scoped to a lexically scoped interpreter by saving the environment in which some expressions are defined and using this saved environment when the expressions are evaluated.

1 Introduction

An environment is a data structure that maps variables to references of their values, which, in turn, can either be another expression or a closure of an expression and an environment. We use references to values instead of the values themselves so that we can imperativelly convert these values whenever necessary. For example, whenever a variable changes its value, we will be able to mutate the environment to reflect this change.

When building an interpreter, in order to evaluate an expression, we can either use the enviroment that was defined when the expression was defined, or the environment that was defined when the expression was being evaluated. In the first case, we speak of an interpreter that evaluates expressions according to their lexically ordering. In the second, we speak of an interpreter that evaluates expressions according to their dynamic ordering.

It is possible to convert dynamically scoped interpreter into a lexically scoped interpreter, and I will explain here how to do so.

2 Converting the Interpreters

Except for functions and function applications, both the lexically scoped and the dynamically scoped interpreters we will be dealing with evaluate the expressions in exactly the same way. Therefore, I will first explain how they evaluate all the expressions but functions and function applications, and then I will explain how they differently evaluate functions and function applications so that it can become evident the main differences between interpreting expressions in a lexically and in a dynamically scoped environment.

In relation to variables, the interpreters simply look the values of these variables in their respective environments. In relation to numbers or bools, they return the expressions themselves, since a number and a bool is already a value. In relation, to let statements, of the type "let v = def in body", first we extend the environment to map v to the value of def, and then we evaluate the body in this new extended environment. This way, by assigning any occurrences of v in the body to be equal to the value of def, we will be able to correctly evaluate the body. In the case of recursive let statements with the type "let rec v = def in body" the process of interpreting the expression is more complex. First, we extend the current environment to map v to an "unassigned value." Then, we evaluate def in this new environment. In this case, insofar as def is a function, it will just evaluate to itself, independently of the environment. However, we need to carefully handle other cases in which ocurrences of v in def can be evaluated to unassigned. After evaluating def in this environment, we change the environment so that v is mapped to whatever the value of def is, instead of unassigned. Finally, we evaluate the body in this new environment, and so any ocurrences of v in the body will be evaluated to the value of def, and we will be able to get the value of the body correctly.

In the case of unary operators, of type "Unop (op, exp)" we evaluate the expression "exp" in the given environment and return the negation of the value we get, insofar as the value we get is a number. If it is not, it is impossible to negate it and we return an error. In the case of a binary operator of the type "Binop (b, exp1, exp2)" we simply return the corresponding operation "b" applied to the values we get from evaluating exp1 and exp2 and the given environment. Again, for each possible type of operation "b" we need to check whether the values we get are numbers. Finally, in the case of contitionals of type "if exp1 then exp2 else exp3", we first evaluate exp1 in the given environment. Then, we check whether the value we get from this evaluation is a bool true or false. In the first case, we return the value of evaluating exp2 in the current environment. In the second case, we return the value of evaluating exp3. However, if the value of exp1 is not a bool, we raise an error.

Although both interpreters evaluate these expressions through the same processes, they evaluate functions and function applications differently. In relation to functions, the dynamically scoped interpreter simply return the functions themselves, regardless of the environment in which they are being defined, whereas the lexically scoped interpreter, when evaluating functions, returns a closure of the function and the environment in which it is being defined. This way, when the lexically scoped interpreter evaluates the application of functions to values, it is able to "retrieve" the environment in which the function was defined, whereas the dynamically scoped interpreter cannot, and so it evaluates the application of functions to values according to the environment in which the application is called.

Therefore, in order to convert a dynamically scoped interpreter to a lexically scoped interpreter, all we need to do is to change the way the interpreter evaluates functions and function applications. In relation to functions, we make the intepreter return a closure that maps the function to its environment, rather than simply the function. For example, if the current environment is "env" and we are trying to evaluate a function of type "fun x -> x + 1", in the case of a lexically scoped environment in which a closure has type "closure (expression, environment)" we return closure ((fun x -> x + 1), env). Finally, in relation to a function application of type "exp1 exp2", in the case of a dynamically scoped environment we would evaluate exp1, what would simply yield a function of type "fun x -> exp3." Then, we would evaluate "exp2" in the current environment, and finally we would evaluate exp3 in the environment we get as a result of extending the current environment to associate the variable "x" to whatever is the value of "exp2." In order to transoform this process into a lexically scoped one, we would instead evaluate an application of type "exp1 exp2" in the following way. First, we evaluate the "exp1" in the current environment, just like we did in a dynamically scoped evaluation. However, what we would get is not simply a function, but a closure that associates the function "fun x -> exp3" to the environment env in which it was defined. Then, we would also evaluate "exp2" in the current environment, but instead of extending the current environment to associate x with the value we get from evaluating "exp2", we would extend the environment "env" that we got from the closure to associate x with the value we get from evaluating "exp2." This way, we evaluate "exp3" in this new environment that we derived from manipulating the environment existent when exp1 was defined, rather than the environment existent when we called the application, and that is how we can evaluate function applications in a lexically scoped environment that, in order to evaluate expressions, takes into account the environment of when the fuctions were defined, rather than the environments of when the functions were called.