

Scoping, Environment, and Closures

Vincent Rahli

CS6110 Lecture 11

Monday Feb 13, 2015

1 Source language

This document discusses the issue of the scope of a variable in a program using a simple programming language, which we call **SL** for Source Language, i.e., the language we want to analyze, study and run. Its syntax is:

t	\in	Term	::=	x	(variable)
				\underline{n}	(natural number)
				$\underline{\lambda}x.t$	(λ -abstraction)
				$t_1 t_2$	(application)
				$t_1 \underline{+} t_2$	(addition)
				$\underline{\text{let}}\ x = t_1\ \underline{\text{in}}\ t_2$	(let-expression)

It is an applied λ -calculus with variables, λ -abstractions, and applications as in the λ -calculus, and in addition has natural numbers, an addition operator, and a let operator. We sometimes write $t_1(t_2)$ for $t_1 t_2$. Note that we use $\underline{\cdot}$ to distinguish the constructors of our source language from the constructors of our meta-language. Our meta-language (ML) will also be an applied λ -calculus such as Milner et al.'s ML language [2].

We define below the operational semantics of this language to be lazy (also called call-by-name), i.e., to evaluate fa , we first have to compute f to a λ -abstraction of the form $\underline{\lambda}x.b$ and then we can β -reduce the redex $(\underline{\lambda}x.b)a$ to $b[x \backslash a]$ without computing first a to a value. Our let operator allows us to selectively and eagerly evaluate terms. To evaluate $\underline{\text{let}}\ x = t_1\ \underline{\text{in}}\ t_2$, we first have to compute t_1 to a value before evaluating t_2 . If t_1 reduces to the value v then $\underline{\text{let}}\ x = t_1\ \underline{\text{in}}\ t_2$ reduces to $t_2[x \backslash v]$. This let operator allows us to define an eager (or call-by-value) application operator as follows: $\text{eager}(f, a) = \underline{\text{let}}\ x = a\ \underline{\text{in}}\ f(x)$, where x is chosen to be fresh w.r.t. f . Another advantage of let constructs is that they also make programs more readable.

Therefore, let constructs allow to selectively choose between a lazy and an eager evaluation strategy in a lazy programming language. This is especially useful given that some programs are more efficient when run lazily and some programs are more efficient when run eagerly. Note that even though most

programming languages are eager such as OCaml, SML, Scala, Lisp, etc., some programming languages still use a lazy evaluation strategy by default, such as Haskell or Nuprl. Note that all these languages support both eager and lazy evaluation, because the eager languages can always simulate lazy evaluation using thunks (wrapping an expression t into a thunk means building something like $\lambda x.t$, where x is free w.r.t. f in order to delay the evaluation of t) and the two lazy languages mentioned above feature let-constructs as the one discussed above.

The sole purpose of adding numbers to our language is to get other kinds of values than λ -expressions.

Exercise 1. *Give an example of a program that runs faster if ran lazily, and one that runs faster if ran eagerly.*

2 Dynamic vs. static scoping

Let us now discuss variables and the issue of the scope of a variable in a program. The *scope* of a variable x in a program is the region which can reference x . There are mainly two kinds of scoping: one is called static or lexical, and the other one is called dynamic. Let us consider the following example:

$$\begin{array}{c} \underline{\text{let } x = 1 \text{ in}} \\ \underline{\text{let } y = x \text{ in}} \\ \underline{\text{let } x = 3 \text{ in}} \\ y \end{array}$$

What is a sensible way of evaluating this program? We first evaluate x , which then gets bound to 1. We then evaluate y which gets bound to x , which is equal to 1. Therefore, the only sensible thing to do here is to bind y to 1. Then, we evaluate the last x which gets bound to 3. And finally we evaluate y which is equal to 1. Our program then returns 1. So, far there does not seem to be any ambiguity. Let's now consider a slightly different program (where we wrapped the definition of y inside a thunk to delay its evaluation):

$$\begin{array}{c} \underline{\text{let } x = 1 \text{ in}} \\ \underline{\text{let } y = \lambda z.x \text{ in}} \\ \underline{\text{let } x = 3 \text{ in}} \\ y(17) \end{array}$$

Again we evaluate x , which gets bound to 1. Then we evaluate y , which gets bound to the value $\lambda z.x$. Then, we evaluate the last x , which gets bound to 3. Finally, we evaluate $y(17)$, which reduces to x . Which x is that? By looking at the structure of our program, it could be the first x . However, at the time we evaluate $(\lambda z.x)17$, x has been rebound to 3, so it could be the second x . The first solution is called static or lexical scoping because matching variables to their bindings can be determined just by looking at the structure of the program, without having to run it. The second solution is called dynamic

binding because one needs to run the program to figure out what variables are bound to. Static scoping is standard in functional programming languages for several reasons. The main one being that programs are easier to read and reason about. Some scripting languages such as Perl still use dynamic scoping. Also, some versions of Common Lisp support both static and dynamic scoping. Note that dynamic binding has some advantages. For example, it can be used to modify the behavior of a program at runtime.

Exercise 2. Give an “interesting” example of a program that uses dynamic scoping to modify its behavior at runtime (using the Y combinator for example, or using a fix that computes as follows $fix(F) \mapsto F(fix(F))$).

3 Environments and closures

Let us make that difference more precise by defining two interpreters for our language: one that uses simple environments of type $e \in \text{Var} \rightarrow (\text{Term} \cup \{\text{error}\})$ to implement dynamic scoping, and one that uses environments of type $c \in \text{Var} \rightarrow (\text{Closure} \cup \{\text{error}\})$ to implement static scoping, where a closure is a pair of a value v and an environment that provides the values of the free variables of v . Closures were first introduced by Landin to design his SECD abstract machine [3]. (We will see later in the course how to build an abstract machine for our source language using Danvy’s methodology [1].) Closures are used to “save” the static/lexical environments of values when evaluating a term in order to get the static scoping of variables right. The empty environment is $\text{em} = \lambda x. \text{error}$. We write $e[x \mapsto t]$ for the environment $\lambda z. \text{if } z = x \text{ then } t \text{ else } e(z)$. (Note that we assume that our meta-language is lazy too.)

Dynamic scoping:

$$\begin{aligned}
 \text{eval}(x, e) &= e(x) \\
 \text{eval}(\underline{n}, e) &= \underline{n} \\
 \text{eval}(\underline{\lambda}x.t, e) &= \underline{\lambda}x.t \\
 \text{eval}(fa, e) &= \text{let } \underline{\lambda}x.b = \text{eval}(f, e) \text{ in} \\
 &\quad \text{eval}(b, e[x \mapsto \text{eval}(a, e)])^1 \\
 \text{eval}(n \pm m, e) &= \text{let } v = \text{eval}(n, e) \text{ in} \\
 &\quad \text{let } w = \text{eval}(m, e) \text{ in} \\
 &\quad v + w \\
 \text{eval}(\underline{\text{let}} x = t \text{ in } b, e) &= \text{let } v = \text{eval}(t, e) \text{ in} \\
 &\quad \text{eval}(b, e[x \mapsto v])
 \end{aligned}$$

¹Note that we need $\text{eval}(a, e)$ to be evaluated lazily here to respect the laziness of our source language (it can be done using thunks).

Static scoping:

$$\begin{aligned}
\text{eval}(x, c) &= c(x) \\
\text{eval}(\underline{n}, c) &= (\underline{n}, c)^2 \\
\text{eval}(\underline{\lambda}x.t, c) &= (\underline{\lambda}x.t, c) \\
\text{eval}(fa, c) &= \text{let } (\underline{\lambda}x.b, c') = \text{eval}(f, c) \text{ in} \\
&\quad \text{eval}(b, c'[x \mapsto \text{eval}(a, c)])^3 \\
\text{eval}(n \pm m, c) &= \text{let } (v, c_1) = \text{eval}(n, c) \text{ in} \\
&\quad \text{let } (w, c_2) = \text{eval}(m, c) \text{ in} \\
&\quad (v + w, c)^4 \\
\text{eval}(\underline{\text{let}} x = t \text{ in } b, c) &= \text{let } (v, c') = \text{eval}(t, c) \text{ in} \\
&\quad \text{eval}(b, c[x \mapsto (v, c')])
\end{aligned}$$

Another solution could be to not only store values in environments but any term, and instead of having the call to eval in the environment in the application case, we could call eval once we have fetched a closure from an environment in the variable case:

$$\begin{aligned}
\text{eval}(x, c) &= \text{let } (t, c') = c(x) \text{ in } \text{eval}(t, c') \\
\text{eval}(\underline{n}, c) &= (\underline{n}, c) \\
\text{eval}(\underline{\lambda}x.t, c) &= (\underline{\lambda}x.t, c) \\
\text{eval}(fa, c) &= \text{let } (\underline{\lambda}x.b, c') = \text{eval}(f, c) \text{ in} \\
&\quad \text{eval}(b, c'[x \mapsto (a, c)]) \\
\text{eval}(n \pm m, c) &= \text{let } (v, c_1) = \text{eval}(n, c) \text{ in} \\
&\quad \text{let } (w, c_2) = \text{eval}(m, c) \text{ in} \\
&\quad (v + w, c) \\
\text{eval}(\underline{\text{let}} x = t \text{ in } b, c) &= \text{let } (v, c') = \text{eval}(t, c) \text{ in} \\
&\quad \text{eval}(b, c[x \mapsto (v, c')])
\end{aligned}$$

4 Side note: Bootstrapping

We see here that our source language **SL** is in fact a subset of our meta-language. How can one then build a compiler for our meta-language? One solution would be to directly code our interpreters in, say **C** or an assembly language, instead of using the high-level language we have used in this document. Note that one can then bootstrap the **SL**'s compiler by first building a compiler for a small subset of **SL** in **C** or assembly and then using that subset to build the rest of the language. To bootstrap a compiler this is what one usually does: (1) write a compiler C_1 for a small subset of **SL**, say **SL0**, using another language for which there is already a compiler; (2) write and compile a compiler C_2 for **SL0** in **SL0** using C_1 ; (3) write a compiler for **SL** in **SL0** using C_2 .

²Here c could be any environment such as **em** because \underline{n} does not have any free variable.

³Again, we need $\text{eval}(a, c)$ to be evaluated lazily to respect the laziness of our source language (it can be done using thunks).

⁴Again, here c could be any environment because $v + w$ does not have any free variable.

5 Side note: Nuprl

Remember that Nuprl uses a uniform syntax for terms. In Nuprl, terms are of the form $t \in \text{NTerm} ::= \{\text{operator}\}(b_1; \dots; b_n)$, where each b_i is called a *bound term* and is of the form $\bar{l}.t$, where \bar{l} is a list of variables (separated by commas, and we omit the dot when \bar{l} is empty) meant to be binders. For example, the underline representation of a lambda $\underline{\lambda}x.t$ in Nuprl is $\{\text{lambda}\}(x.u)$, where u is the Nuprl representation of t , and the underline representation of an application fa is $\{\text{apply}\}(g; b)$, where g is the Nuprl representation of g and b is the Nuprl representation of a . Let-expressions are called *call-by-value* expressions in Nuprl and are of the form $\{\text{callbyvalue}\}(t_1; x.t_2)$ (let us write $\{\text{cbv}\}(t_1; x.t_2)$ for short). An operator can be more than a “name”, it can also have parameters. We use parameters, e.g., to represent variables and numbers. An operator is really an identifier followed by a list of parameters, and a parameter is a value and a type. For example, the variable x is represented as $\{\text{variable}, x : v\}()$ in Nuprl, where x is the value of our parameter and v its type indicating that it is a variable. The natural number 17 is represented as $\{\text{natural_number}, 17 : n\}()$.

6 To be continued

In a compiler for a functional programming language, once the front-end (not covered here) has generated some internal representation of the code, programs go through a number of transformations. The evaluator that uses closures above gives rise to a conversion called *closure conversion*. This is a key transformation to, among other things, get the binding of variables right. In future lectures we will cover the CPS and defunctionalization transformations, and we will show how to get from a simple evaluator that uses substitution to an abstract state machine [1].

References

- [1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden*, pages 8–19. ACM, 2003.
- [2] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, volume 78. Springer-Verlag, NY, 1979.
- [3] Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(5):308–320, 1964.