Lecture Notes: Implementing Functional Languages

17-363/17-663: Programming Language Pragmatics (Fall 2022)
Jonathan Aldrich
jonathan.aldrich@cs.cmu.edu

1 Closures

How do we implement first-class functions, the defining feature of functional programming languages? In our semantics, we define function calls in terms of substitution. For example, if we have a function $x \Rightarrow y \Rightarrow x + y$ and we invoke it with a value 3, then we substitute 3 for x in the body of the function, and we get $y \Rightarrow 3 + y$. But substitution is not an efficiently implementable operation; if an expression is represented as an abstract syntax tree and we substitute a value for a variable within it, we must traverse the tree to find all occurrences of the variable. Thus substitution is an O(n) operation; using this as an implementation technique will change the asymptotic run time (and wall-clock time) of many algorithms.

A better approach is to defer the substitution. When we call the function described above, instead of getting $y \Rightarrow 3 + y$ we will leave the function as $y \Rightarrow x + y$, but pair this function with an *environment* $[x \mapsto 3]$ that specifies values for the free variables in the function. This pairing is called a *closure*, because it contains the variable to value mappings that are necessary to make the function into a closed expression.

A big-step semantics with environments and closures forms the basis of an efficient interpreter for functional programming languages. We'll study closures in the context of the lambda calculus. Environments E map variables to values. Functions are no longer values; instead, values are closures, i.e. a pair of an environment and a function:

$$\begin{array}{lcl} E & \in & \mathit{Var} \rightarrow \mathit{Value} \\ e & ::= & x \Rightarrow e[x] \mid e_1(e_2) \mid x \\ v & ::= & \langle E, x \Rightarrow e[x] \rangle \end{array}$$

Most of the evaluation rules are similar to what we wrote for our earlier big-step substitution semantics. The differences are that we now pass an environment around. Evaluating a variable looks up the value in the environment, while evaluating a function captures the current environment, restricted to the free variables of the function body, in a closure. When a function call is made, the body is evaluated in the environment captured in the closure, with a mapping for the argument added:

$$\frac{E(x) = v}{E \vdash x \Downarrow v} \qquad \frac{E_1 = E|_{free\_vars(e)}}{E \vdash x \Rightarrow e \Downarrow \langle E_1, x \Rightarrow e \rangle}$$

$$\frac{}{E \vdash v \Downarrow v} \qquad \frac{E \vdash e_1 \Downarrow \langle E_1, x \Rightarrow e \rangle \quad E \vdash e_2 \Downarrow v_2 \quad E_1[x \mapsto v_2] \vdash e \Downarrow v}{E \vdash e_1(e_2) \Downarrow v}$$

Exercise 1. Assume that we add let, addition, and integer constants to the language above, as we did in prior lectures. Show the derivation of the following expression using the rules above:

$$(x \Rightarrow y \Rightarrow x + y)(3)(4)$$

Memory use considerations: In a real implementation, closures should only capture variables that are actually used by the function. Otherwise, the closure may have the only reference to a data structure, preventing that data structure from being reclaimed by the garbage collector. Doing this slows down closure creation a bit, since you can't just use the existing environment, but it can speed up variable lookup (depending on the implementation strategies used) because the set of variables in environments that come from closures is minimized. We will omit this in our formal treatment to keep things simple, but it is important in practice.

2 Closure Conversion

Let's consider how to compile code with closures. What we want to do is convert code written in a language with first-class functions (and therefore closures) to a language without. In the process, we will encode first-class functions and closures in terms of simpler constructs. The first stage of this process is called closure conversion; it will convert first-class functions into functions that cannot close over their environment along with records.

Our source language will be an extension of the lambda calculus with let and recursive functions, because recursive functions are an interesting special case. We also allow functions to have multiple arguments:

$$e \quad ::= \quad \overline{x} \Rightarrow e[\overline{x}] \mid e_1(\overline{e}) \mid x \mid \text{let } x = e_1 \text{ in } e_2 \mid \text{fun } f(\overline{x}) = e[f, \overline{x}]$$

The destination language will be the lambda calculus with let and records:

$$e \quad ::= \quad \overline{x} \Rightarrow e[\overline{x}] \mid e_1(\overline{e}) \mid x \mid \text{let } x = e_1 \text{ in } e_2 \mid e.x \mid \{\overline{x = e}\}$$

We'll define closure conversion with a judgment $\Delta \vdash e \rightsquigarrow e$, where the first e is in our source language and the second e is in our destination language. Δ is an environment mapping variables to expressions, representing what we need to do to access each variable after the translation. We now define the rules:

$$\frac{\Delta(x) = e}{\Delta \vdash x \rightsquigarrow e}$$

$$\frac{\Delta \vdash e_1 \rightsquigarrow e_1' \quad \Delta, x \mapsto x \vdash e_2 \rightsquigarrow e_2'}{\Delta \vdash \text{let } x = e_1 \text{ in } e_2 \rightsquigarrow \text{let } x = e_1' \text{ in } e_2'}$$

$$\frac{free\_vars(e) = \overline{x}, \overline{y} \quad \overline{x \mapsto x}, \overline{y \mapsto \$env.y} \vdash e \rightsquigarrow e' \quad \overline{\Delta(y) = e''}}{\Delta \vdash \overline{x} \Rightarrow e \rightsquigarrow \{\$fn = (\overline{x}, \$env) \Rightarrow e'; \overline{y = e''}\}}$$

$$\frac{\Delta \vdash e \rightsquigarrow e' \quad \Delta \vdash \overline{e} \rightsquigarrow \overline{e'}}{\Delta \vdash e(\overline{e}) \rightsquigarrow \text{let } \$closure = e' \text{ in } \$closure.fn(\overline{e'}, \$closure)}$$

$$\frac{free\_vars(e) = \overline{x}, \overline{y} \quad \overline{x \mapsto x}, \overline{y \mapsto \$env.y}, f \mapsto \$env \vdash e \rightsquigarrow e' \quad \overline{\Delta(y) = e''}}{\Delta \vdash \text{fun } f(\overline{x}) = e \rightsquigarrow \{\$fn = (\overline{x}, \$env) \Rightarrow e'; \overline{y = e''}\}}$$

Exercise 2. Show the result of closure conversion on the following program:

$$(\text{let } y = 1 \text{ in } x \Rightarrow x + y)(2)$$

This approach only works for immutable variables, because it copies variable contents into the closure; updates to variables in the surrounding scope will not be visible from inside the closure, and vice versa. However, if we implement mutable variables as a pointer to a heap-allocated cell of memory, then copying the pointer works properly. Alternatively, the environment can capture the stack frame of the surrounding procedure directly, rather than copying just the captured values, but this approach has the disadvantage that any variables in the captured stack frame are not garbage collected until all inner closures are also garbage collected.