CS611 Lecture 16: Defining a function environment for REC+ 10/04/99
Matthew Fleming

Today we continued discussion of a denotational semantics for the language REC+. We will consider a function $\mathcal{C}[\![e]\!] : \mathit{Fenv} \times \mathit{Env} \to \mathcal{Z}_\perp$, where *Fenv* is our function environment, and can be thought of as similar to a jump table constructed by the linker; *Env* is our environment, and can be thought of as the stack on which variables are stored. We have a state $\rho : \mathit{Env} = \mathit{Var} \to \mathcal{Z}$ (in a call-by-value semantics, $\mathcal{Z}_\perp$ for call-by-name), and $\phi : \mathit{Fenv} = \mathcal{Z} \to (\mathcal{Z}^{a_i}_{(\perp)} \to \mathcal{Z}_\perp)$, where function arguments can possibly be $\perp$, depending on the semantics.

We have already seen most of the denotational semantics for REC+; the one missing piece was how to define the function environment ϕ. Today we'll fill in this missing piece. As a reminder, the denotation of functions is:

$$\mathcal{C}[\![f_i(e_1, \ldots, e_n)]\!] = \lambda\phi\rho.(\phi i)(\mathcal{C}[\![e_1]\!]\phi\rho, \ldots, \mathcal{C}[\![e_n]\!]\phi\rho) \qquad \text{(call-by-value)}$$

How we define ϕ_i depends on how we allow functions to refer to each other in our language. We identified 3 possibilities:

1. earlier functions only
2. self or earlier functions
3. all functions

For case 1, all programs terminate, and we can easily define ϕ as

$$\phi(i) = \lambda(n_1, \ldots, n_{a_i}).\mathcal{C}[\![d_i]\!]\phi_0[1 \mapsto \phi(1), 2 \mapsto \phi(2), \ldots, i-1 \mapsto \phi(i-1)]\rho_0[x_1 \mapsto n_1, \ldots, x_{a_i} \mapsto n_{a_i}]$$

In other words, the ith member of the function environment (ϕi) is defined in terms of all the previous members of the function environment. Here we use ϕ_0 to denote an empty function environment, and ρ_0 to denote an empty environment. For case 2, we must define ϕ_i in terms of a fixed point operator, since we may have references within function f_i to f_i.

$$\phi i = \mathit{fix}\ \lambda f\ .\ \lambda(n_1, \ldots, n_{a_i})\ .\ \mathcal{C}[\![d_i]\!]\phi_0[1 \mapsto \phi(1), \ldots, i-1 \mapsto \phi(i-1), i \mapsto f]\rho_0[x_1 \mapsto n_1, \ldots, x_{a_i} \mapsto n_{a_i}]$$

For case 3, we want a map ϕ such that $i \mapsto \mathcal{C}[\![d_i]\!][1 \mapsto \phi_1, \ldots, n \mapsto \phi_n]$. We obtain this using a function F such that $F(\phi) = \phi$. We define F as

$$F = \lambda\phi : \mathit{Fenv}.\lambda i.\lambda(n_1, \ldots, n_{a_i}) : \mathcal{Z}.\mathcal{C}[\![d_i]\!]\phi\rho_0[x_1 \mapsto n_1, \ldots, x_{a_i} \mapsto n_{a_i}]$$

Now suppose we add a fixed point operator to our language, like (rec x e), which intuitively works like *fix* $\lambda x.e$; i.e. we evaluate e, binding occurrences of x within e to the result of e. With this rec operator, we no longer need our functions to be implicitly recursive. For example, we can write factorial as

$$\text{let } \mathit{fact}\text{=(rec } f \text{ (fn}(x) \text{ if } x < 2 \text{ then } 1 \text{ else } f(x-1) \times x))$$

But can we actually define such an operation? In call-by-name semantics, we have

$$\mathcal{C}[\![\text{rec } x\ e]\!] = \lambda p : \mathit{Env}\ .\ \mathit{fix}\ \lambda e' : \mathit{Value}_\perp.\mathcal{C}[\![e]\!]\rho[x \mapsto e']$$

How is it possible to implement this? We can set up a placeholder object, install it in the environment, evaluate the code (which may refer to this placeholder), and then mutate the the placeholder to point to the code we just created. Of course, this is an imperative operation that we cannot describe in a functional meta-language like the one that we are using. The *fix* operator allows us to get around this limitation of our meta-language.

In a call-by-name semantics, when we get something from the stack, we don't know if it's been evaluated. We may need to evaluate our thunks. This is some overhead, so we prefer a call-by-value semantics.

Consider (rec x (cons 1 x)), which is the list 1,1,1, … , or equivalently, a cons cell containing the item 1, with the next field pointing back to itself. In call-by-value, the denotation doesn't type check, because of the type of e'. We can't start with a $\perp$ in call-by-value and use *fix* to get a value here, because our environment is a function $Var \to \mathcal{Z}$; we can't stick $\perp$ into the environment, as we can in a lazy language. To support this form of recursion, we really need a pointed CPO as the domain of our variables. But we don't usually have this, since *Value* is not a pointed CPO. In call-by-name, we can do this because an environment maps variables to a pointed CPO; in the actual implementation, the indirection in this infinite list goes through a closure that we patch up later.

We can support the rec operator in call-by-value semantics somewhat. Consider a domain $\mathit{Value}_{\perp_B}$, where $\perp_B$ is a bottom for bindings only; it is an actual value that is not the same thing as $\perp$. However, it is considered to be less than any other value in *Value*, and therefore $\mathit{Value}_{\perp_B}$ is a pointed CPO. Now, we change our environment *Env* to a function $Var \to \mathit{Value}_{\perp_B}$ and we define $\mathcal{C}[\![\text{rec } x\ e]\!] = \lambda p : \mathit{Env}.\mathit{fix}\ \lambda e' : \mathit{Value}_{\perp_B}\ \mathcal{C}[\![e]\!]\rho[x \mapsto e']$. This fixed point works because $\mathit{Value}_{\perp_B}$ is a pointed CPO.

To implement this semantics, we create a concrete "forwarding object". We evaluate the recursive expression with the variable x bound in the environment to an empty forwarding object (in other words, $\perp_B$.) We then patch the forwarding object to point to the value produced. If the evaluation of the expression tries to evaluate the variable x, we have a problem because the value of x is not yet available; we will consider this to create an infinite loop. We redefine our denotation of variables to take care of the case that we encounter a forwarding object in the environment:

$$\mathcal{C}[\![x]\!]\rho = \begin{cases} \perp & \text{if } \rho(x) = \perp_B \\ \rho(x) & \text{otherwise} \end{cases}$$

But this still won't let us handle (rec x (cons 1 x)), because an eager cons operator will try to look at x, and will go off into an infinite loop. We can simulate infinite lists, but only with a function indirection: (rec x (cons 1 (fn (a) x))) will give us our infinite list — we don't evaluate x inside the expression itself, but only mention it inside a lambda. So we get a list that has 1 as the first element, and a function which gives back the same list as the second element.

As a summary, denotational semantics made it easy to create a call-by-name semantics, with lazy evaluation. But we need to be careful because call-by-name is hard to implement, and we need to be sure that what we write for call-by-value is implementable.