CS611 Lecture 15 Parameter Passing September 21, 2000
Scribe: Niranjan Nagarajan and Bo Pang Lecturer: Andrew Myers

In this lecture we shall discuss several different styles of parameter passing and ways of defining the scope of function definitions. For the purpose of illustration of these ideas we shall use the REC$^+$ language which is basically an extension of the REC language used in Chapter 9 of Winskell. We start off by presenting the Operational and Denotational Semantics of REC$^+$.

1 REC$^+$ language

Syntactic Forms
$d ::= f_1(x_1, \ldots, x_{a_1}) = e_1 \ldots f_n(x_1, \ldots, x_{a_n}) = e_n$
$e ::= n \mid X \mid e_1 \oplus e_2 \mid \textbf{ifz}\, e_0 \,\textbf{then}\, e_1 \,\textbf{else}\, e_2 \mid \textbf{let}\, x = e_1 \,\textbf{in}\, e_2$
$program ::= d\ e$

Operational Semantics
$v ::= n$
$(d, \textbf{ifz}\, 0 \,\textbf{then}\, e_1 \,\textbf{else}\, e_2) \to (d, e_1)$
$(d, \textbf{ifz}\, v \,\textbf{then}\, e_1 \,\textbf{else}\, e_2) \to (d, e_2)\ \ where\ v \neq 0$
$(d, v_1 \oplus v_2) \to (d, n_3)$ where $n_3 = v_1 \oplus v_2$
$(d, \textbf{let}\, x = v \,\textbf{in}\, e) \to (d, e\{v/x\})$
$(d, f_i(v_i \ldots v_{ai})) \to (d, e_i\{v_j/x_j\ ^{j \in 1..\, a_i}\})$

Domains
$\mathcal{C}[\![e]\!] \in FEnv \to Env \to Result$
$Result = Z_\perp$
$Env = Var \to Z$
$FEnv = (Z^{a_1} \to Result) \times \ldots \times (Z^{a_n} \to Result)$

Denotational Semantics
$\mathcal{C}[\![n]\!]\phi\rho = \lfloor n \rfloor$
$\mathcal{C}[\![x]\!]\phi\rho = \lfloor \rho(x) \rfloor$
$\mathcal{C}[\![e_1 \oplus e_2]\!]\phi\rho = \mathcal{C}[\![e_1]\!]\phi\rho \oplus_\perp \mathcal{C}[\![e_2]\!]\phi\rho$
$\mathcal{C}[\![\textbf{ifz}\, e_0 \,\textbf{then}\, e_1 \,\textbf{else}\, e_2]\!]\phi\rho = let\ v = \mathcal{C}[\![e_0]\!]\phi\rho\ .\ if\ v = 0\ then\ \mathcal{C}[\![e_1]\!]\phi\rho\ else\ \mathcal{C}[\![e_2]\!]\phi\rho$
$\mathcal{C}[\![\textbf{let}\, x = e_1 \,\textbf{in}\, e_2]\!]\phi\rho = let\ v = \mathcal{C}[\![e_1]\!]\phi\rho\ .\ \mathcal{C}[\![e_2]\!]\phi\rho[x \mapsto v]$
$\mathcal{C}[\![f_i(e_1, \ldots, e_{a_i})]\!]\phi\rho = let\ v_1 = \mathcal{C}[\![e_1]\!]\phi\rho.(\ldots let\ v_{a_i} = \mathcal{C}[\![e_{a_i}]\!]\phi\rho\ .\ \pi_i\phi(v_1, \ldots, v_{a_i}) \ldots)$

The Operational and Denotational Semantics given here do not address the question of how we get the function environments ϕ. Also in the Denotational Semantics that we have given we define the language constructs like let in terms of the corresponding mathematical constructs and the definition seems to be suspiciously circular. In the following section we shall try to address these problems and also demonstrate an application of Denotational Semantics.

2 Parameter Passing and Function Scope

REC$^+$ as we have defined above has call-by-value semantics as we need to β-reduce operands before function evaluations. Let us explore the semantics of other methods of parameter passing using REC$^+$. Here is a table of the various kinds of parameter passing modes and function scopes that we shall consider:

	Param. Passing		Function Scope
A	CBV	1	Function only in scope in later functions (FORTRAN 77 or earlier)
B	CBN	2	Function can be recursive but can't call later functions (C without fwd decls.)
C	CBD	3	Function can call later functions (letrec in Scheme, Java)

CBV, CBN and CBD stand for call-by-value, call-by-name and call-by-denotation respectively. We shall refer to the languages that we consider by the alphabet, number pairs indicated by the table, so for e.g. B3 is the name for the language that has call-by-name and in which functions can call later functions.

We invent a new meaning function $\mathcal{D}$ s.t. $\mathcal{D}[\![d]\!]$ gives us the function environment that corresponds to the meaning of the function declarations d in a particular semantics. We would then distinguish the various semantics based on the way in which $\mathcal{D}$ is defined. So the meaning of a program $[\![d e]\!]$ is given by the meaning of e in the function enviorment $\mathcal{D}[\![d]\!]$ with the initial variable environment being the empty set (since in the beginning none of the variables are bound to any values.)

$$[\![d\,e]\!] = \mathcal{C}[\![e]\!]\mathcal{D}[\![d]\!]\emptyset$$
$$\mathcal{D}[\![d]\!] = \langle \mathcal{F}_1, \ldots, \mathcal{F}_n \rangle$$

where the $\mathcal{F}_i$ are denotations of the f_i in d.

3 A1 and B1

A1. The semantics for the A1 language can be expressed by the following additional definitions:

$$\mathcal{D}_{i-1}[\![d]\!] = \langle \mathcal{F}_1, \ldots, \mathcal{F}_{i-1} \rangle$$
$$\mathcal{F}_i = \lambda v \in Z^{a_i}.\, \mathcal{C}[\![e_i]\!]\mathcal{D}_{i-1}[\![d]\!]\{x_j \mapsto \pi_j v^{\;j \in 1..a_i}\}$$

Here we are defining $\mathcal{F}_i$ on previous $\mathcal{F}$, and we can inductively build up the $\mathcal{D}_i[\![d]\!]$ such that $\mathcal{D}[\![d]\!] = \mathcal{D}_n[\![d]\!]$. Note that here our definitions make sense even when the result domain is not a pointed domain. This indicates that the A1 language does not have divergent computations. The A1 language is not very powerful and in fact one cannot even write infinite loops in it. The semantics of A1 can also be expressed by inlining function calls in e for the functions in d in sequential order.

B1. Since the semantics of call-by-name differs from call-by-value only in the presence of non-terminating computations and since all languages X1 (where X $\in$ {A, B, C}) lack non-terminating computations, the semantics of B1 is identical to that of A1.

4 A2 and B2

A2. By allowing functions to be recursive we force our definitions of $\mathcal{F}_i$ to be recursively defined in terms of $\mathcal{D}_i[\![d]\!]$. We therefore need to use fixed points to remove the circularity in the definitions:

$$\mathcal{F}_i = fix(\lambda f \in Z^{a_i} \to Z.\, \lambda v \in Z^{a_i}.\, \mathcal{C}[\![e_i]\!]\langle \mathcal{F}_1, \ldots, \mathcal{F}_{i-1}, f\rangle\{x_j \mapsto \pi_j v^{j \in 1 \ldots a_i}\})$$

Clearly by allowing recursion we open up the possibility of divergent computation in A2. We also note that for the A2 language the result domain has to be pointed in order for us to be able to take the fixed point given by the above definition.

B2. We need to modify our definition for *Env* and *FEnv* to allow for the presence of divergent computations as arguments to functions.

$$Env = Var \to Z_\perp$$
$$FEnv = ({Z_\perp}^{a_1} \to Z_\perp) \times \ldots \times ({Z_\perp}^{a_n} \to Z_\perp)$$

In addition we need to make the following changes to our definitions:

$$\mathcal{C}[\![x]\!]\phi\rho = \rho(x)$$
$$\mathcal{C}[\![\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2]\!]\phi\rho = \mathcal{C}[\![e_2]\!]\rho[x \mapsto \mathcal{C}[\![e_1]\!]\phi\rho]\phi$$
$$\mathcal{C}[\![f_i(e_1, \ldots, e_{a_i})]\!]\phi\rho = \pi_i\phi(\mathcal{C}[\![e_1]\!]\phi\rho, \ldots, \mathcal{C}[\![e_{a_i}]\!]\phi\rho)$$

Suprisingly the denotational semantics for call-by-name is much more compact and simple than the semantics for call-by-value. This is because the underlying mathematical language used by us is biased towards lazy languages.

5 A3 and B3

A3. When a function can refer to all the other functions, how do we obtain ϕ? We will first pretend that we already have this function environment ϕ, and then we could write:

$$\mathcal{F}_i = \lambda v \in Z^{a_i}.\mathcal{C}[\![e_i]\!]\phi\rho[x_j \mapsto \pi_j v]$$

But this is a circular definition, since

$$\phi = \langle \mathcal{F}_1, \ldots, \mathcal{F}_n \rangle$$

Clearly, we have to again take a fixed point!

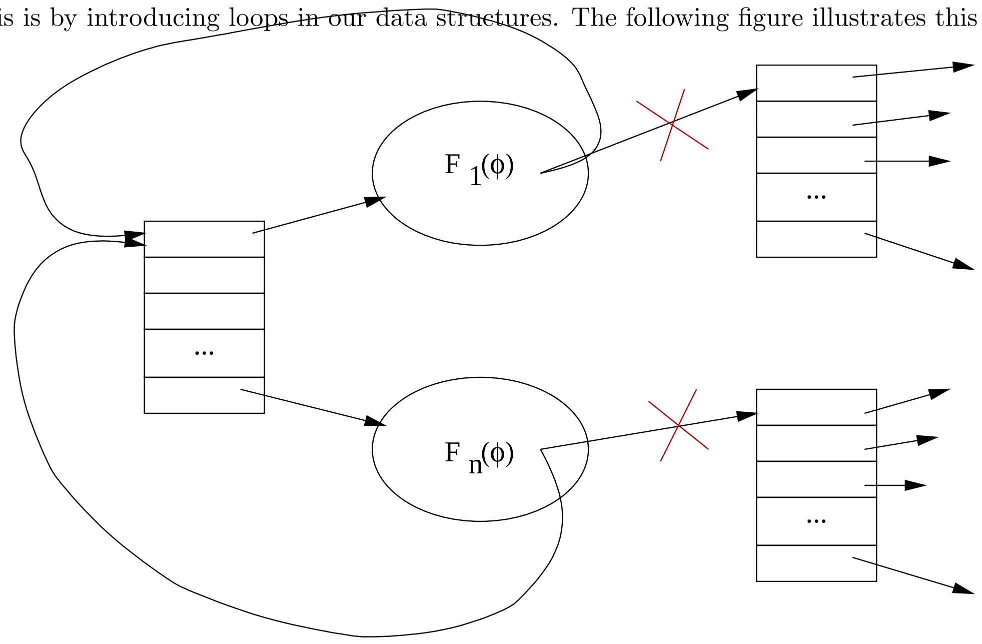
$$\phi = fix\ \lambda\phi.\langle \mathcal{F}_1(\phi), \ldots, \mathcal{F}_n(\phi) \rangle$$

We know that this function is continuous and we need to check if the domain is a pointed cpo, we have

$$\phi \in (Z^{a_1} \to Z_\perp) \times \ldots \times (Z^{a_n} \to Z_\perp)$$

Since $Z_\perp$ is a pointed cpo, so each function domain $Z^{a_i} \to Z_\perp$ is also a pointed cpo, and the product domain of all these pointed cpo function domains is still a pointed cpo.

Since the function environment is defined in terms of fixed points which in turn are defined by LUB's of infinite sequences it seems like our function environments would need to be an infinite data structure. However our function environments need to be finite in any realistic implementation. The way we accomplish this is by introducing loops in our data structures. The following figure illustrates this notion:



The crossed-out pointers indicate the option of an infinite data structure that we do not implement and instead replace with the pointers back to the function environment. Fixed points are useful because they can capture the semantics of implementation entities containing loops in a way that is amenable to inductive reasoning.

B3. Here we can perform the same transformation as we did for going from A2 to B2 for going from A3 to B3 (that is, we get rid of the strict lets from our definitions, making it lazy.)

6 Call by denotation

Call by denotation is a bizzare parameter-passing convention that describes what happens in macro languages such as TEXor CPP. Consider the following example:

$$f(x) = \textbf{let}\ a = 0\ \textbf{in}\ a + x$$

$$\textbf{let}\ a = 10\ \textbf{in}\ f(a)$$

With call by denotation, we will not obtain 10 as a result, instead, we will substitute a where x occurs, and get

$$\textbf{let}\ a = 0\ \textbf{in}\ a + a$$

and have 0 as the result. So how should we formally define this? We will introduce the notion of a *Binding* for this, and the domains now take the form of:

$$\mathit{Env} = \mathit{Var} \rightarrow \mathit{Binding}$$

$$\mathit{Denotation} = \mathit{FEnv} \rightarrow \mathit{Env} \rightarrow \mathit{Restult}$$

$$\mathit{Function} = \mathit{Binding} \rightarrow \mathit{Result}$$

For different forms of Parameter Passing the notion of Binding takes on different meanings:

Param Passing	Meaning of Binding
CBV	Z: a value
CBN	$Z_\perp$: a computation
CBD	a Denotation: $\mathit{FEnv} \rightarrow \mathit{Env} \rightarrow \mathit{Restult}$

But with Call by Denotation, the definition of the domain *Env* depends on itself, since

$$\mathit{Env} = \mathit{Var} \rightarrow \mathit{Binding} = \mathit{Var} \rightarrow \mathit{FEnv} \rightarrow \mathit{Env} \rightarrow \mathit{Result}$$

We might want to resort to taking fixed points as we have done so often before, but it turns out that taking fixed points over domains is a bit trickier, and we will ignore this problem for the moment. So now the defination of domain *FEnv* changes to:

$$\mathit{FEnv} = (\mathit{Binding}^{a_1} \rightarrow \mathit{Result}) \times \ldots \times (\mathit{Binding}^{a_n} \rightarrow \mathit{Result})$$

and we also need to update the semantics:

$$\mathcal{C}[\![x]\!]\phi\rho = \rho(x)\phi\rho = \rho x\phi\rho$$

$$\mathcal{C}[\![f_i(e_1, \ldots, e_{a_i})]\!]\phi\rho = (\pi_i\phi)(\mathcal{C}[\![e_1]\!], \ldots, \mathcal{C}[\![e_{a_i}]\!])$$

Note that when we pass parameters to a function, we do not use ϕ, ρ to interpret them, but instead pass them in textually and interpret them in the ϕ and ρ of the function. Also note that this call-by-denotation semantics is easy to implement, but it is difficult to write modular programs with it as the meaning of an expression can change unpredictably in the environment in which it is finally evaluated.