CS611 Lecture 15: Parameter passing 10/01/99
Aleksey Kliger

Today we will take a look at several different styles of parameter passing and function call semantics. We will use the language REC+ which is similar to the REC language defined in Chapter 9 of Winskel.

1 REC+ Syntax

$$
\begin{array}{rcl}
program & ::= & d\ e \\
d & ::= & \epsilon \\
 & | & f_i(x_1,\ldots,x_{a_i}) = d_i\ d \\
e & ::= & n \mid x \mid \textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \mid f_i(e_1,\ldots,e_{a_i}) \mid e_1\ op\ e_2 \mid \neg e \\
op & ::= & + \mid - \mid / \mid * \mid \wedge \mid \vee \mid \leq \mid =
\end{array}
$$

Where d_i is an expression that is the body of function f_i.

1.1 Sample program

Computes the next prime number following 1000:

$$
\begin{array}{rcl}
f_1(n,m) & = & m*n \leq m \wedge (n = m*(n/m)) \vee f_1(n, m+1) \\
f_2(n) & = & f_1(n,2) \\
f_3(n) & = & \textbf{if}\ f_2(n)\ \textbf{then}\ f_3(n+1)\ \textbf{else}\ n \\
 & & f_3(1000)
\end{array}
$$

2 Semantics

We will take a look at several different versions of the semantics for $REC+$ by varying several attributes of the language (Fig. 1).

Parameter passing:	Functions may refer to:
A. Call by value.	1. Earlier functions in the list (ala FORTRAN).
B. Call by name.	2. Earlier functions and themselves (ala C).
	3. All functions.

Figure 1: Variations on REC+

2.1 Operational Semantics

In the operational semantics, we will use $\langle d,\ e\rangle$ as a configuration, where d is the set of definitions in the program and e is an expression.

2.1.1 A1

The A1 language is not very powerful. In fact, it cannot even loop. We can express the semantics of A1 by simply recursively inlining all function calls into the body of the expression and then trivially evaluating the arithmetic expressions.

$$
\begin{array}{lll}
\langle d,\, e\rangle & \rightarrow e & [d_n\,[a_1/x_1, a_2/x_2, \cdots, a_{a_n}/x_{a_n}]/f_n(a_1, a_2, \ldots, a_{a_n})] \\
& & [d_{n-1}\,[a_1/x_1, a_2/x_2, \cdots, a_{a_{n-1}}/x_{a_{n-1}}]/f_{n-1}(a_1, a_2, \ldots, a_{a_{n-1}})] \\
& & \cdots \\
& & [d_1\,[a_1/x_1, a_2/x_2, \cdots, a_{a_1}/x_{a_1}]/f_1(a_1, a_2, \ldots, a_{a_1})]
\end{array}
$$

The operational semantics given above do not give us any meaningful distinction between A1 and B1.

2.1.2 A3

For most expressions, the semantics of the A3 language are similar to the semantics for IMP:

$$\frac{\langle d,\, e_1\rangle \rightarrow n_1 \quad \langle d,\, e_2\rangle \rightarrow n_2}{\langle d,\, e_1 \text{ op } e_2\rangle \rightarrow n}\; n = n_1 \text{ op } n_2$$

In the case of a function call, we evaluate each of the arguments in turn and substitute the values into the function body:

$$\frac{\forall j \langle d,\, e_j\rangle \rightarrow n_j}{\langle d,\, f_i(e_1, \ldots, e_{a_i})\rangle \rightarrow \langle d,\, d_i\,[n_1/x_1]\rangle}$$

In this call by value language it is possible to write expressions that will loop indefinitely even if the value of the divergent calculation is not needed:

$$
\begin{array}{rcl}
f_1() & = & f_1() + 1 \\
f_2(x) & = & 0 \\
& & f_2(f_1())
\end{array}
$$

In a call by name language (such as B3 below), this program will evaluate to 0, since the value of the argument $(f_1())$ is not needed by f_2.

2.1.3 B3

In the call by name language, the function call semantics change such that we substitute argument expressions into function bodies before evaluating them:

$$\frac{}{\langle d,\, f_i(a_1, a_2, \ldots, a_{a_i})\rangle \rightarrow \langle d,\, d_i\,[e_1/x_1, e_2/x_2, \ldots]\rangle}$$

In actual practice, call by value semantics are easier to implement than call by name.

Call by need A common improvement to call by name semantics, known as call by need, is often used. Call by need is essentially call by name, except that we evaluate each argument at most once and use the resulting value wherever the parameter occurred:

$$
\begin{array}{rcl}
f_1() & = & 1 \\
f_2(x) & = & x + x \\
& & f_2(f_1())
\end{array}
$$

In call by name semantics, we would be evaluating the expression $f_1() + f_1()$. In call by need, we would evaluate the same expression, but after evaluating $f_1()$ once, we would remember the result and reuse it.

2.2 Denotational Semantics

2.2.1 Call by value languages

We will use the compilation function:

$$\begin{array}{rcl}\mathcal{C}[\![e]\!] & : & \mathit{Fenv} \times \mathit{Env} \to \mathbb{Z}_\perp \\ \mathit{Fenv} & = & \mathbb{Z} \to (\mathbb{Z}^{a_i} \to \mathbb{Z}_\perp) \\ \mathit{Env} & = & \mathit{Var} \to \mathbb{Z}\end{array}$$

Here Fenv is the function environment (a mapping from the function's integer index to a mathematical function denoted by the corresponding REC+ function), and Env is the variable environment.

Some simple denotations:

$$\begin{array}{rcl}\mathcal{C}[\![n]\!] & = & \lambda\phi\rho.\lfloor n \rfloor \\ \mathcal{C}[\![x]\!] & = & \lambda\phi\rho.\lfloor \rho(x) \rfloor \\ \mathcal{C}[\![\mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]\!] & = & \lambda\phi\rho.\text{let } b = \mathcal{C}[\![e_0]\!]\phi\rho \text{ in } \left\{\begin{array}{ll}\perp & \text{if } b = \perp \\ \mathcal{C}[\![e_1]\!]\phi\rho & \text{if } b \neq \lfloor 0 \rfloor \wedge b \neq \perp \\ \mathcal{C}[\![e_2]\!]\phi\rho & \text{otherwise}\end{array}\right.\end{array}$$

Recall that $\lfloor\ \rfloor$ is the lifting operator such that $(\forall x \in S)\lfloor x \rfloor \in S_\perp$.

As usual, function calls are the interesting case:

$$\begin{array}{rcl}\mathcal{C}[\![f_i\,(e_1, \ldots, e_{a_i})]\!] & = & \lambda\phi\rho.\text{let } v_1 = \mathcal{C}[\![e_1]\!]\phi\rho, v_2 = \mathcal{C}[\![e_2]\!]\phi\rho, \ldots \text{ in} \\ & & \text{if } v_1 = \perp \vee v_2 = \perp \vee \ldots \text{ then } \perp \text{ else } (\phi i)(v_1, v_2, \ldots, v_{a_i})\end{array}$$

We cannot just write $\ldots \text{in}(\phi i)(v_1, v_2, \ldots, v_{a_i})$ because a function call should diverge if any of its arguments diverge.

Similarly, expressions involving the primitive operators should also be strict:

$$\begin{array}{rcl}\mathcal{C}[\![e_1 \text{ op } e_2]\!] & = & \lambda\phi\rho.\mathcal{C}[\![e_1]\!]\phi\rho \text{ op}_\perp\, \mathcal{C}[\![e_2]\!]\phi\rho \\ \text{op}_\perp & = & \left\{\begin{array}{ll}\perp & \text{if } n_1 = \perp \vee n_2 = \perp \\ n_1 \text{ op } n_2 & \text{otherwise}\end{array}\right.\end{array}$$

Next time: How do we compute (ϕi) and how do we update ρ.

2.2.2 Call by name

In the call by name languages, the function environment and the variable environment can contain unfinished computations, and those computations may diverge:

$$\begin{array}{rcl}\mathit{Fenv} & = & \mathbb{Z} \to (\mathbb{Z}_\perp^{a_i} \to \mathbb{Z}_\perp) \\ \mathit{Env} & = & \mathit{Var} \to \mathbb{Z}_\perp\end{array}$$

The denotations for constants and variables are similar to the call by value semantics, while the primitive operators and function calls are no longer strict:

$$\begin{array}{rcl}\mathcal{C}[\![n]\!] & = & \lambda\phi\rho.\lfloor n \rfloor \\ \mathcal{C}[\![x]\!] & = & \lambda\phi\rho.\rho(x) \\ \mathcal{C}[\![f_i(e_1, \ldots, e_{a_i})]\!] & = & \lambda\phi\rho.(\phi i)(\mathcal{C}[\![e_1]\!]\phi\rho, \mathcal{C}[\![e_2]\!]\phi\rho, \ldots, \mathcal{C}[\![e_{a_i}]\!]\phi\rho)\end{array}$$

The additional complication in actual practical implementation of call by name semantics is reflected in the more complicated definition of Env which now needs to be able to save computations.

2.3 Call by denotation

In call by denotation semantics, variables are evaluated in the environment where they are used. Call by denotation is especially popular in macro languages such as TEX or `cpp(1)`. Consider the REC+ (assuming that let-expression syntactic sugar has been added to the language) program:

$$\begin{array}{rcl} f(x) & = & \text{let } a = 0 \text{ in } a + x \\ & & \text{let } a = 10 \text{ in } f(a) \end{array}$$

In the call by denotation semantics, the above program returns zero as a gets substituted for x in the body of f, and a evaluated in the body of f yields zero.

To represent call by denotation semantics we will need to extend our notion of an environment (here, let us suppose that we have a single unified environment for both variables and functions), as figure 2.3 shows, for call by denotation semantics an environment is a mapping from variable names to functions from environments to saved computations. You are encouraged to ignore the fact that *Env* is defined in terms of itself (Chapter 12 of Winskel talks about such recursive definitions, we will not worry about it now).

Call by value	$Env : Var \to Result$
Call by name	$Env : Var \to Result_{\perp}$
Call by denotation	$Env : Var \to (Env \to Result_{\perp})$

Table 1: Environment representations in various semantics.

Returning to REC+, we express call by denotation as follows:

$$\begin{array}{rcl} Env & = & Var \to (Fenv \times Env \to \mathbb{Z}_{\perp}) \\ \mathcal{C}[\![x]\!] & = & \lambda \phi \rho . \rho(x)(\phi \rho) \\ \mathcal{C}[\![f_i(e_1, \ldots, e_{a_i}) & = & \lambda \phi \rho . (\phi i)(\mathcal{C}[\![e_1]\!], \ldots, \mathcal{C}[\![e_{a_i}]\!]) \end{array}$$