Lecture #7: Parameter Passing & Scoping

October 4, 2007

1 Today

Parameter Passing (Call-by-name, Call-by-value), and Lexical-static vs. Dynamic Scope. Reading: 7-7.2

What is "call-by-value" and "call-by-name?" Call-by-name involves passing computations as the formal parameters to functions – these could even be errors. Call-by-value passes values as the parameters of functions, and is typically more efficiently implemented, in practice.

2 The Semantics of Parameter Passing

2.1 Operational View

Recall from last week:

$$\begin{array}{rcl} \mathbb{E}^{CBV} & = & \ldots \mid (\mathtt{app}\,\mathbb{E}^{CBV}\,E) \\ (\mathtt{app}\,(\mathtt{lam}\,I\,E_1)\,E_2) & \rightsquigarrow & [E_2/I]\,E_1 \end{array}$$

In Call-by-Value,

$$\begin{array}{rcl} \mathbb{E}^{CBV} & := & \ldots \mid (\mathtt{app}\,\mathbb{E}^{CBV}\,E) \mid (\mathtt{app}\,V\,\mathbb{E}^{CBV}) \\ (\mathtt{app}\,(\mathtt{lam}\,I\,E)\,V) & \rightsquigarrow & [V/I]\,E \end{array}$$

Now, we will see that if $E \stackrel{*}{\Rightarrow}_{CBV} V$ then $E \stackrel{*}{\Rightarrow}_{CBN} V$. That is to say, that the set of expressions that we can evaluate with call-by-name is strictly larger than the set that can be evaluated with CBV. For instance, try evaluating $(\mathtt{app}\,(\mathtt{lam\,x\,3})\,(\mathtt{rec\,y\,y}))$.

2.2 Denotational View

Again, recall from last time:

$$\begin{array}{rcl} n \in Nameable & = & Comp \\ name-to-comp : Nameable \to Comp & = & \lambda n.n \end{array}$$

$$\begin{aligned}\mathcal{E}[(\texttt{app}\, E_1\, E_2)] &= \lambda e.(with-proc-comp\,(\mathcal{E}[E_1]\, e)\\ &\quad (\lambda p.(p\,(\mathcal{E}[E_2]\, e)))\end{aligned}$$

This is Call-by-Name semantics. How would we convert it to call-by-value?

$$\begin{aligned}n \in Nameable &= Value\\ name-to-comp &= val-to-comp\\ \mathcal{E}[(\texttt{app}\, E_1\, E_2)] &= \lambda e.(with-proc-comp\,(\mathcal{E}[E_1]\, e)\\ &\quad (\lambda p.(with-value\,(\mathcal{E}[E_2]\, e)\\ &\quad (\lambda v.(p\, v)))))\end{aligned}$$

What is this "with-value?" Let's write that out, too.

$$\begin{aligned}with-value &: Comp \to (Value \to Comp) \to Comp\\ &= \lambda cf.\text{match}\, c\\ &\quad \rhd(Value \mapsto Comp\, v) \mid (f\, v)\\ &\quad \rhd else\, c\end{aligned}$$

This makes sense to you, if you've been paying attention in recitation section. Dave then proceeds to work out examples of evaluating (taking the meaning of) the same expression twice, using the denotation semantics of CBV and CBN. I won't reproduce that here. It should be in the course text, I think.

The other note here is that, in going to Call-by-Value, we need to change our definition of the `rec` keyword.

$$\begin{aligned}\mathcal{E}[(\texttt{rec}\, I\, E)] &= \lambda e.\\ &\quad (\text{fix}_{Comp}\,(\lambda c.(\mathcal{E}[E]\,(bind\, I\,(extractvalue\, c)\, e))))\\ bind &: Ident \to BindingVal \to Env \to Env\\ extractvalue &: Comp \to BindingVal\\ &= \lambda c.\text{match}\, c\\ &\quad \rhd(Value \mapsto Expressible\, v) \mid (Value \mapsto BindingVal\, v)\\ &\quad \rhd(Error \mapsto Expressible\, y) \mid \bot_{BindingVal}\end{aligned}$$

Make sense?

2.3 Other Semantics

Are there other kinds of parameter passing semantics out there? Here's an example, of a new parameter passing semantics:

$$((\texttt{abs}\,(\texttt{y})\,(\texttt{let}\,((\texttt{x}\,3))\,\texttt{y}))\,\texttt{x}) \Rightarrow 3$$

(Clearly, this would return $\bot$in either CBV or CBN.) What would you surmise, about the meaning of the new semantics based on this example? "Instead of

passing along the meaning of a computation as a parameter, we're going to be pass along the meaning of the operand without respect to any environment, which we'll only interpret at the point of use." Can we make the smallest change possible, to our FLK semantics, to capture this meaning?

Well, it's clear that what's nameable in these new semantics are not computations.

$$\begin{array}{rcl} n \in Nameable & = & Env \to Comp \\ \mathcal{E}[(\texttt{app}\, E_1\, E_2)] & = & \lambda e.(withproccomp\, (\mathcal{E}[E_1]\, e) \\ & & (\lambda p.(p\, \mathcal{E}[E_2]))) \end{array}$$

And we also need to modify the meaning of identifiers.

$$\begin{array}{rcl} \mathcal{E}[I] & = & \lambda e.(with-nameable\, (lookup\, I\, e) \\ & & (\lambda n.(n\, e))) \end{array}$$

These semantics are usually called "Call-by-Denotation," where our free-variables are not interpreted until their use.

3 Scoping

We've talked, so far, about 'lexical' or 'static' scoping – for each variable reference, we can determine what variable definition it corresponds to. Typically, these are defined using sorts of static, nested scopes.

But an alternative to this is known as 'dynamic' scoping. For instance, exceptions in Java (and most languages) are dynamically scoped. APL is dynamically scoped, as were early versions of LISP. Free-references to variables are resolved using the dynamic environment, not the definition of the procedure itself.

```
(let ((a 1))
  (let ((f (abs (x) (@ + x a))))
    (let ((a 20)) (f 300))))
```

In static scoping, this has a value of 301, but in dynamic scoping it's 320. What changes do we need to make to our semantics, to get dynamic scoping?

$$\begin{array}{rcl} p \in Proc & = & Nameable \to Env \to Comp \\ \mathcal{E}[(\texttt{lam}\, I\, E)] & = & \lambda e_1.(valtocomp\, (Proc \mapsto Value \\ & & (\lambda n.(\lambda e_2.(\mathcal{E}[E_2]\, [I \mapsto n] e_2))))) \\ \mathcal{E}[(\texttt{app}\, E_1\, E_2)] & = & \lambda e_1.(withproccomp\, (\mathcal{E}[E_1]\, e_1) \\ & & (\lambda p.(p\, ((\mathcal{E}[E_2]\, e_1)\, e_1)))) \end{array}$$