Lecture #6: FL Semantics

October 4, 2007

1 Today

FL Operational Semantics, FL Denotational Semantics. Reading: Chapter 6 (not 6.6), and Context-Based Semantics (3.2.6).

2 Review

We review the content of the last lecture (#5) from last week. We defined a desugaring function $\mathcal{DS} : Exp_{FL} \rightarrow Exp_{FLK}$, that translated expressions from FL into the FLK 'kernel' language. Recall that we defined a notation for substitution expressions, and we noted that they have a non-trivial interaction with language constructs that 'bind' variable names. That is to say, they pay attention to the free/bound status of the variables in the expressions that they substitute. This is going to be important, because we'll use the substitution operator to define the semantics of FLK in a reasonably interesting way.

2.1 Evaluation Context

Another concept from last week is the 'evaluation context,' which some people seemed confused by. Recall that $\mathbb{E} \in EvalContext ::= | (\texttt{if}\, \mathbb{E}\, E_1\, E_2) |$ *etc.*And we defined a 'reduction relation,' which is a shorthand way of writing the progress rules that we saw before. $\mathbb{E}\{E\} \Rightarrow \mathbb{E}\{E'\}\, if\, E \rightsquigarrow E'$. So in order to define the semantics of the language, we need to define the reduction relation.

3 The FL Reduction Relation

Let's talk about the reduction relation for the 'if' statement, to begin with.

$$(\texttt{if}\, \#\texttt{t}\, E_1\, E_2) \rightsquigarrow E_1$$

So the 'evaluation contexts' are just a shorthand themselves, for 'where' (that is, what sub-region) of the program we're 'looking at' at any given time – "Where

we do our work." The two most important forms are 'apply' and 'rec'.

$$(\texttt{app}\,(\texttt{lam}\,I\,E_1)\,E_2) \rightsquigarrow [E_2/I]E_1$$

And this is the 'meaning of application.'

$$(\texttt{rec}\,I\,E) \rightsquigarrow [(\texttt{rec}\,I\,E)/I]E$$

Whoa. Think about why this works. (It provides us with the 'infinite unrolling' that we need to define the meaning of E.) And then we work out an example with primop, to show how the rec form works. For instance, is it obvious that

$$(\texttt{rec x x}) \Rightarrow [(\texttt{rec x x})/\texttt{x}]\texttt{x} \Rightarrow (\texttt{rec x x})$$

Is that clear? In some sense, (rec x x)'represents' non-termination.

How would we adapt these semantics to implement Call-By-Value? (You could add another form of your 'app' statement, to require values, and then update the reduction rule to only 'call' the function when the argument has been evaluated to a value.)

4 FL Denotational Semantics

The first thing we do when we talk about the 'denotational semantics' of a programming language, we want to think about the values that the programs can express. And we do this, by writing down some semantic domains. So let's write down some semantic domains for FL.

$$\begin{aligned}
c &\in Comp = Expressible \\
x &\in Expressible = (Value + Error)_\perp \\
v &\in Value = Unit + Bool + Int + Sym + Pair + Proc \\
p &\in Proc = Nameable \to Comp \\
n &\in Nameable = Comp \\
e &\in Env = Ident \to BindingVal \\
\beta &\in BindingVal = (Nameable + Unbound)_\perp
\end{aligned}$$

And these are the core of our semantic domains. We'll redefine *Nameable* later, as we re-purpose the language. We will also define a couple of 'helper functions', which you should be familiar with (at least in concept) from the problem set

$$extend \quad : \quad Ident \to Nameable \to Env \to Env$$

which extends an environment into a new environment. So let's have a look at some meaning functions.

$$\begin{aligned}
\mathcal{E} &: Exp \to Env \to Comp \\
\mathcal{E}[L] &= \lambda e.(Value \mapsto Expressible\,\mathcal{L}[L]) \\
\mathcal{L}[N] &= (Int \mapsto Value\,\mathcal{N}[N])
\end{aligned}$$

where $\mathcal{N} : IntLit \to Int$ is a function that maps integer literals into their corresponding integers. Is everyone with us so far? We're defining the mathematical meaning of the script-function $\mathcal{E}$ by breaking it apart based on the grammar.

$$\begin{aligned}
\mathcal{E}[(\texttt{if}\ E_1\ E_2\ E_3)] &= \lambda e.\text{match}\,(\mathcal{E}[E_1]\,e) \\
&\quad \triangleright(Value \mapsto Comp\,v)\,| \\
&\quad \ldots(\text{match}\,v \\
&\quad \ldots \triangleright\,(Bool \mapsto Value\,b)\,| \\
&\quad \ldots\ldots(\text{if}\,b\,\text{then}\,(\mathcal{E}[E_2]\,e)\,\text{else}\,(\mathcal{E}[E_3]\,e)) \\
&\quad \ldots \triangleright\,\text{else}\,(Error \mapsto Comp\,\texttt{error})) \\
&\quad \triangleright\text{else}\,(\mathcal{E}[E_1]\,e) \\
\mathcal{E}[I] &= \lambda e.\text{match}\,(e\,I) \\
&\quad \triangleright(Nameable \mapsto BindingVal\,n)\,|\,n \\
&\quad \triangleright(Unbound \mapsto BindingVal\,\texttt{unbound})\,|\,(Error \mapsto Comp\,\texttt{error}) \\
&\quad \triangleright\text{else}\,(e\,I)
\end{aligned}$$

You'll notice that the helper functions give us more concise, compact (and readable) forms of the same function definitions.

$$\begin{aligned}
\mathcal{E}[I] &= \lambda e.(with-nameable\,(lookup\,I\,e) \\
&\quad \ldots(\lambda n.(name-to-comp\,n)))
\end{aligned}$$

What is the signature of 'with-nameable'? $with-nameable : BindingVal \to (Nameable \to Comp) \to Comp$.

So now we have the meaning of an identifier. Can we define a more complicated meaning?

$$\begin{aligned}
\mathcal{E}[(\texttt{lam}\,I\,E)] &= \lambda e.(Value \mapsto Comp\,(Proc \mapsto Value \\
&\quad \ldots(\lambda n.\mathcal{E}[E]\,[I \mapsto n]e)))
\end{aligned}$$

What about the application statement?

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

This one's in the book. I didn't transcribe it all, because I needed to respond to some email at this point in class.

Finally, we'll define the meaning of the rec construct.

$$\mathcal{E}[(\texttt{rec}\,I\,E)] = \lambda e.(\text{fix}_{Comp}\,\lambda c.(\mathcal{E}[E]\,[I \mapsto c]e))$$

Does everyone see why this is the case? Recall the lecture (and recitation section, ahem) on the meaning of recursive definitions, and on fixed-point operators. The expression $\lambda c.\ldots$ is our generating function here. It has signature $Comp \to Comp$, which means that when we apply fix_{Comp} to it, we get back the element

of the *Comp* domain that is the least fixed point of that generating function – the meaning of the recursive definition, n'est-ce pas?

$$\begin{aligned}
\mathcal{E}[(\texttt{rec x x})]empty - env &= \text{fix}_{Comp}\ \lambda c.(\mathcal{E}[\texttt{x}]\ [\texttt{x} \mapsto c]empty - env) \\
&= \text{fix}_{Comp}\ \lambda c.c \\
&= \bot_{Comp}
\end{aligned}$$

And ... so we've shown that (rec x x) has a particular meaning.