Lecture #5: The Language FL

October 4, 2007

1 Today

The FLK Language, FL. Reading: Chapter 6.

2 The FLK Language

FLK is "FL Kernel", and FL is the language that "surrounds" it. We'll give a 'desugaring' language, that transforms the FL language back down into expressions in FLK. And then we can describe a formal semantics for the kernel, FLK, and we get a formal semantics for FL 'for free.'

$$
\begin{aligned}
&L \in Lit = \#\mathtt{u} | Bool | Nat | (\mathtt{sym}\ \mathtt{y}) \\
&B \in BoolLit = \{\mathtt{true}, \mathtt{false}\} \\
&N \in IntLit = \{\ldots -1, 0, 1, \ldots\} \\
&Y \in SymLit = \{\mathtt{x}, \mathtt{list}, etc.\} \\
&Keyword = \{\mathtt{app}, \mathtt{err}, \mathtt{flk}, \mathtt{if}, \mathtt{pair}, \mathtt{prim}, \mathtt{lam}, \mathtt{rec}, \mathtt{sym}\} \\
&I \in Ident = SymLit - Keyword - \{@y | y \in SymLit\}
\end{aligned}
$$

Now, we can give the actual grammars for the expressions in our kernel FLK.

$$
\begin{aligned}
E \in Exp \quad ::= \quad & L | I | (\mathtt{error}\ y) | (\mathtt{if}\ E_1\ E_2\ E_3) | \\
& (\mathtt{prim}\ O\ E) | (\mathtt{lam}\ I\ E) | (\mathtt{app}\ E_1\ E_2) | \\
& (\mathtt{pair}\ E_1\ E_2) | (\mathtt{rec}\ I\ E) \\
P \in Prog \quad ::= \quad & (\mathtt{flk}\ (I\ *)\ E)
\end{aligned}
$$

Dave continues by giving some examples of FLK expressions and their values on the board, which I won't re-transcribe here. Check your course text.

3 The FL Language

So, FLK is pretty wordy. What kind of 'creature comforts' could we add, to make FLK programs a little easier to write?

$$
\begin{array}{rcl}
P & ::= & \ldots|(\texttt{fl}\ (\texttt{I}*)\ \texttt{E}\ \texttt{D}) \\
D & ::= & (\texttt{def}\ \texttt{I}\ \texttt{E})|(\texttt{def}\ (\texttt{I}_\texttt{p}\ \texttt{I}*)\ \texttt{E}) \\
E & ::= & \ldots|(@\texttt{o}\ \texttt{E})|(\texttt{abs}\ (\texttt{I}*)\ \texttt{E})| \\
 & & (\texttt{E}\ \texttt{E}*)|(\texttt{list}\ \texttt{E}*)|(\texttt{quote}\ \texttt{sx})| \\
 & & (\texttt{cond}\ (\texttt{E}_\texttt{t}\ \texttt{E}_\texttt{c})\ (\texttt{else}\ \texttt{E}_\texttt{d}))|(\texttt{scond}\ \texttt{E}*)| \\
 & & (\texttt{scon}\ \texttt{E}*)|(\texttt{let}\ ((\texttt{I}\ \texttt{E})*)\ \texttt{E})|(\texttt{letrec}\ ((\texttt{I}\ \texttt{E})*)\ \texttt{E}_\texttt{b})
\end{array}
$$

And the way that we'll interpret FL into FLK, by providing a 'desugaring' function that expresses constructs in FL solely using constructs in the FLK kernel language.

Dave now writes out the definition, through example, of the desugaring function $\mathcal{DS}[\cdot]$. But if I tell you that this function is defined, in all its glory, on page 205 of the course text, I'm sure you won't fault me for writing it out again here.

A question from the back of the room: "Is he saying, 'desugar' or 'deshiver'?" He's saying "desugar," because you can think of FL as 'syntactic sugar' around the kernel FLK. Syntactic sugar is a way of saying, 'additions to a language which are added for the convenience of the user, but that shouldn't change the meaning of the language (that is, they can always be reduced to expressions from the original language).' You can think of them as macros.

Please double-check, and understand, the desugaring of letrec and why it works the way it does. This one is probably the trickiest, and most important, of the desugarings. It uses the 'list' and 'projection' trick, which was implicitly used when discussing the reduction of multiply-recursive definitions to singly-recursive versions at the beginning of the last lecture.

4 Free and Bound Variables

A 'bound' variable is, informally, a variable whose 'meaning is defined.' Conversely, a free variable is one whose meaning is not defined. The scope of a variable is the region of a program phrase where the particular variable can be accessed. In a 'let' construct, for instance, the score of a variable defined in the let head is the body of the let statement.

Notice that, in FLK, only 'lam' and 'rec' actually bind variables. We say that a 'bound variable' is 'declared in an expression' and 'free variables' are variables references not in the scope of a declaration. This is starting to get towards a more formal definition of 'bound' and 'free'.

We introduce two functions, FrIds[] and BdIds[] which define the free and bound sets of identifiers in particular expressions of FLK. These are defined recursively, but they change only when processing a rec or lam expression.

A sense of where variables are bound, and their scope, leads to a notion of variable renaming which is safe. This is called "alpha-substitution," and expressions which are the same up to an alpha transformation of their bound variables are said to be 'alpha-equivalent'.

5 Substitutions

We define a substitution operator, which works with definitions of bound and unbound variables. This will be used a great deal in the future.

$$\begin{aligned}
[E_1/I]I &= E_1 \\
[E_1/I]I' &= I' \\
[E_1/I](\texttt{lam I E}) &= (\texttt{lam I E}) \\
[E_1/I](\texttt{lam I}'\texttt{ E}) &= (\texttt{lam I}_{\texttt{fresh}}\ [E_1/I][I_{fresh}/I']\texttt{E})
\end{aligned}$$

Understanding this last definition is to understand the core of the semantics of this language.

6 An SOS for FLK

Recall, that an SOS is $< Exp, \Rightarrow, ValueExp, IF, OF >$, where $IF : Prog \times InputExp* \rightarrow Exp$, $OF : Exp \rightarrow AnsExp$, and we define the domain V,

$$V ::= N|B|\#\texttt{u}|(\texttt{pair E}_1\ \texttt{E}_2)|(\texttt{lam I E})$$

Now, we will introduce the notion of an evaluation context.

$$\begin{aligned}
\mathbb{E} \in EvalContext &= |\,(\texttt{if}\ \mathbb{E}\ \texttt{E}_1\ \texttt{E}_2) \\
&\quad |(\texttt{prim O V}_1 \ldots \texttt{V}_\texttt{k}\ \mathbb{E}\ \texttt{E}_{\texttt{k+1}} \ldots \texttt{E}_1)| \\
&\quad (\texttt{app}\ \mathbb{E}\ \texttt{E}_1)
\end{aligned}$$

And then we define the $\Rightarrow$relationship using the notion of, $\mathbb{E}\{E\} \Rightarrow \mathbb{E}\{E'\}$if we can transform E to E'.