Let us construct a functional language FL by augmenting the \(\lambda\)-calculus with some more conventional programming constructs. This is a richer language than any we have seen, one that we might actually like to program in. We will give semantics for this language in two ways: a structural operational semantics and a translation to the CBV \(\lambda\)-calculus.

1 Syntax of FL

In addition to \(\lambda\)-abstractions, we introduce some new primitive constructs: tuples \((e_1, \ldots, e_n)\), natural number constants \(n\), Boolean constants \(true\) and \(false\), and a \texttt{letrec}\ construct for recursive functions. All these constructs are primitive constructs of the language; that is, they are given as part of the basic syntax, not encoded by other constructs. We could also include arithmetic and Boolean operators as before, but let’s leave these out for now for simplicity of the exposition.

1.1 Expressions

Expressions are defined by the following BNF grammar:

\[
e ::= \lambda x_1 \ldots x_n.e \mid e_0e_1 \mid x \mid n \mid true \mid false \\
   \mid (e_1, \ldots, e_n) \mid \#n e \mid \text{if } e_0\text{ then } e_1\text{ else } e_2 \\
   \mid \text{let } x = e_1 \text{ in } e_2 \mid \text{letrec } f_1 = \lambda x_1.e_1 \text{ and } \ldots \text{ and } f_n = \lambda x_n.e_n \text{ in } e
\]

where \(n\) is strictly positive in projections \(\#n e\), \(\lambda\)-abstractions \(\lambda x_1 \ldots x_n.e\), and the \texttt{letrec}\ construct.

Computation will be performed on closed terms only. We have said what we mean by closed in the case of \(\lambda\)-terms, but there are also variable bindings in the \texttt{let}\ and \texttt{letrec}\ construct, and we need to extend the definition to those cases by defining the scope of the bindings. The scope of the binding of \(x\) in \texttt{let } \(x = e_1 \text{ in } e_2\) is \(e_2\) (but \textit{not} \(e_1\!\)), and the scope of \(f_i\) in \texttt{letrec } \(f_1 = \lambda x_1.e_1 \text{ and } \ldots \text{ and } f_n = \lambda x_n.e_n \text{ in } e\) is the entire expression, including \(e_1, \ldots, e_n\) and \(e\).

1.2 Values

Values are a subclass of expressions for which no reduction rules will apply. Thus values are \textit{irreducible}. There will be other irreducible terms that are not values; this will be the \textit{stuck}\ values.

\[
v ::= \lambda x_1 \ldots x_n.e \mid n \mid true \mid false \mid (v_1, \ldots, v_n)
\]

2 Operational Semantics

As before, we will specify our operational semantics structurally in terms of reductions and evaluation contexts.

2.1 Evaluation Contexts

We define evaluation contexts so that evaluation is left-to-right and deterministic.

\[
E ::= [\cdot] \mid E \cdot \mid vE \mid \#n E \mid \text{if } E\text{ then } e_1\text{ else } e_2 \\
   \mid \text{let } x = E \text{ in } e \mid (v_1, \ldots, v_m, E, e_{m+2}, \ldots, e_n)
\]
There are no holes on the right-hand side of if because we will want \( e_1 \) and \( e_2 \) to be evaluated lazily. Even in an eager, call-by-value language, we want some laziness.

The structural congruence rule takes the usual form:

\[
\frac{e \xrightarrow{\Delta} e'}{E[e] \xrightarrow{\Delta} E[e']}
\]

2.2 Reductions

\[
(\lambda x_1 \ldots x_n. e) v \rightarrow (\lambda x_2 \ldots x_n. e)\{v/x_1\}, \quad n \geq 2
\]
\[
(\lambda x. e) v \rightarrow e\{v/x\}
\]
\[
\#n (v_1, \ldots, v_m) \rightarrow v_n, \text{ where } n \leq m
\]
\[
\text{if true then } e_1 \text{ else } e_2 \rightarrow e_1
\]
\[
\text{if false then } e_1 \text{ else } e_2 \rightarrow e_2
\]
\[
\text{let } x = v \text{ in } e \rightarrow e\{v/x\}
\]
\[
\text{letrec . . . } \rightarrow \text{(to be continued)}
\]

We can already see that there will be problems with soundness. For example, what happens with the expressions \( \text{if } 3 \text{ then } 1 \text{ else } 0 \) or \#5 (true, false, true)? In these cases, the evaluation is stuck, because there is no reduction rule that applies, but the expression is not a value. Unlike the \( \lambda \)-calculus, not all expressions work in all contexts. We do not have an explicit notion of type in this language to rule out such expressions. Typically in practice, stuck expressions constitute a runtime type error.

3 Translating FL to \( \lambda \)-CBV

3.1 Application and Abstraction, Numbers and Booleans

To capture the semantics of FL, we can also translate it to the call-by-value \( \lambda \)-calculus. The translation is defined by structural induction on the syntax of the expression. For the basis of the induction,

\[
[x] \triangleq x \quad [n] \triangleq \lambda f x. f^m x \quad [\text{true}] \triangleq \lambda x y. x \text{id} \quad [\text{false}] \triangleq \lambda x y. y \text{id}
\]

The compound constructs other than tuples, projections, and letrec are translated as follows:

\[
[\lambda x_1 \ldots x_n. e] \triangleq \lambda x_1. [\lambda x_2 \ldots x_n. e], \quad n \geq 2
\]
\[
[\lambda x. e] \triangleq \lambda x. [e]
\]
\[
[e_0 e_1] \triangleq [e_0] [e_1]
\]
\[
[\text{if } e_0 \text{ then } e_1 \text{ else } e_2] \triangleq [e_0] (\lambda z. [e_1]) (\lambda z. [e_2])
\]
\[
[\text{let } x = e_1 \text{ in } e_2] \triangleq (\lambda x. [e_2]) [e_1]
\]
\[
[\text{letrec . . . } \triangleq \text{(to be continued)}
\]

2
3.2 Tuples

Let us consider the translation of tuples. We have already seen how to represent pairs in the \(\lambda\)-calculus. Using these constructs, we can define the translation from tuples to \(\lambda\)-CBV as follows:

\[
\begin{align*}
\llbracket ( ) \rrbracket & \triangleq \lambda x y. x \\
\llbracket (e_1, e_2, \ldots, e_n) \rrbracket & \triangleq (\lambda x y b. b x y) \llbracket e_1 \rrbracket \llbracket (e_2, \ldots, e_n) \rrbracket \\
\llbracket \#1 e \rrbracket & \triangleq \llbracket e \rrbracket (\lambda x y. \llbracket (e) \rrbracket (n - 1) y) \\
\llbracket \#n e \rrbracket & \triangleq \llbracket e \rrbracket (\lambda x y. \llbracket (e) \rrbracket (n - 1) y)
\end{align*}
\]

The translation is not sound, because there are stuck \(FL\) expressions whose translations are not stuck; for example, \(\#1()\).

4 Recursive Functions

Recursion in \(FL\) is implemented with the \(letrec\) construct

\[
letrec\ f_1 = \lambda x_1. e_1 \text{ and } \ldots \text{ and } f_n = \lambda x_n. e_n \text{ in } e.
\]

This construct allows us to define mutually recursive functions, each of which is able to call itself and other functions defined in the same \(letrec\) block. Note that all the variables \(f_i\) are in scope in the entire expression; thus any \(f_i\) may occur in \(e\) and in any of the bodies \(e_j\) of the functions being defined. The latter occurrences represent recursive calls.

For the semantics of \(letrec\), we will consider only the case \(n = 1\) for simplicity of the presentation. The operational semantics is given by the following reduction rule:

\[
\begin{align*}
letrec\ f = \lambda x \cdot e_1 \text{ in } e & \rightarrow e \{ (\lambda x \cdot e_1) \{ letrec \ f = \lambda x \cdot e_1 \text{ in } f/f \} / f \}. \\
(1)
\end{align*}
\]

Some explanation of this rule is in order. First, let us look at the two subexpressions

\[
\begin{align*}
letrec\ f = \lambda x \cdot e_1 \text{ in } f & \rightarrow f \{ (\lambda x \cdot e_1) \{ letrec \ f = \lambda x \cdot e_1 \text{ in } f/f \} / f \} \\
& = (\lambda x \cdot e_1) \{ letrec \ f = \lambda x \cdot e_1 \text{ in } f/f \}.
\end{align*}
\]

appearing on the right-hand side of the rule (3). Both expressions represent the recursive function being defined, and the latter is a value. The former reduces to the latter under the rule (3):

\[
\begin{align*}
letrec\ f = \lambda x \cdot e_1 \text{ in } f & \rightarrow (\lambda x \cdot e_1) \{ letrec \ f = \lambda x \cdot e_1 \text{ in } f/f \} / f
\end{align*}
\]

The substitution of \(letrec\ f = \lambda x \cdot e_1 \text{ in } f\) for free occurrences of \(f\) in \(\lambda x \cdot e_1\) is what makes the function recursive. Later on in the computation, when this expression is again exposed and applied to a value \(v\), we will have

\[
\begin{align*}
(\lambda x \cdot e_1) \{ letrec \ f = \lambda x \cdot e_1 \text{ in } f/f \} / f & \rightarrow (\lambda x \cdot e_1) \{ letrec \ f = \lambda x \cdot e_1 \text{ in } f/f \} / v
\end{align*}
\]

If \(f = x\), then \(f\) has no free occurrences in \(\lambda x \cdot e_1\), so the expressions (4) both reduce to \(\lambda x \cdot e_1\). In this case the \(letrec\) construct is equivalent to the ordinary nonrecursive \(let\).
For the translation to $\lambda$-CBV, recall from Lecture 5 that, using the $Y$-combinator, we can produce a fixpoint $Y(\lambda f. \lambda x. e)$ of $\lambda f. \lambda x. e$. We can think of $Y(\lambda f. \lambda x. e)$ as a recursively-defined function $f$ such that $f = \lambda x. e$, where the body $e$ can refer to $f$. Then we define

$$\llbracket \text{letrec } f = \lambda x. e_1 \text{ in } e \rrbracket \Vdash (\lambda f. \llbracket e \rrbracket) (Y(\lambda f. \llbracket \lambda x. e_1 \rrbracket)).$$

We should not use the original $Y$ combinator, but the more CBV-friendly combinator $Y_{CBV}$ as defined in Lecture 5.

References