Let us construct a functional language FL by augmenting the λ-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 λ-calculus.

1 Syntax of FL

In addition to λ-abstractions, we introduce some new primitive constructs: tuples $(e_1, \ldots, e_n)$, natural number constants n, Boolean constants true and false, and a 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:

$$\begin{aligned} e \quad ::= \quad & \lambda x_1 \ldots x_n . e \mid e_0\, e_1 \mid x \mid n \mid \mathsf{true} \mid \mathsf{false} \\ & \mid (e_1, \ldots, e_n) \mid \#n\, e \mid \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \\ & \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{letrec}\ f_1 = \lambda x_1 . e_1\ \mathsf{and}\ \ldots\ \mathsf{and}\ f_n = \lambda x_n . e_n\ \mathsf{in}\ e \end{aligned}$$

where n is strictly positive in projections $\#n\, e$, λ-abstractions $\lambda x_1 \ldots x_n . e$, and the letrec construct.

Computation will be performed on closed terms only. We have said what we mean by *closed* in the case of λ-terms, but there are also variable bindings in the let and 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 $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ is e_2 (but *not* e_1!), and the scope of f_i in $\mathsf{letrec}\ f_1 = \lambda x_1 . e_1\ \mathsf{and}\ \ldots\ \mathsf{and}\ f_n = \lambda x_n . e_n\ \mathsf{in}\ e$ 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 *irreducible*. There will be other irreducible terms that are not values; this will be the *stuck* values.

$$v \ ::= \ \lambda x_1 \ldots x_n . e \mid n \mid \mathsf{true} \mid \mathsf{false} \mid (v_1, \ldots, v_n)$$

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.

$$\begin{aligned} E \quad ::= \quad & [\cdot] \mid E\, e \mid v\, E \mid \#n\, E \mid \mathsf{if}\ E\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \\ & \mid \mathsf{let}\ x = E\ \mathsf{in}\ e \mid (v_1, \ldots, v_m, E, e_{m+2}, \ldots, e_n) \end{aligned}$$

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:

$$\frac{e \xrightarrow{1} e'}{E[e] \xrightarrow{1} E[e']}$$

2.2 Reductions

$$\begin{array}{rcl}
(\lambda x_1 \ldots x_n . e)\, v & \to & (\lambda x_2 \ldots x_n . e)\{v/x_1\},\ n \geq 2 \\
(\lambda x . e)\, v & \to & e\{v/x\} \\
\#n\,(v_1, \ldots, v_m) & \to & v_n, \text{ where } n \leq m \\
\mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 & \to & e_1 \\
\mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 & \to & e_2 \\
\mathsf{let}\ x = v\ \mathsf{in}\ e & \to & e\{v/x\} \\
\mathsf{letrec} \ldots & \to & (\textit{to be continued})
\end{array}$$

We can already see that there will be problems with soundness. For example, what happens with the expressions if 3 then 1 else 0 or $\#5\,(\mathsf{true}, \mathsf{false}, \mathsf{true})$? In these cases, the evaluation is *stuck*, because there is no reduction rule that applies, but the expression is not a value. Unlike the λ-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 λ-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 λ-calculus. The translation is defined by structural induction on the syntax of the expression. For the basis of the induction,

$$[\![x]\!] \triangleq x \qquad [\![n]\!] \triangleq \lambda f x . f^n x \qquad [\![\mathsf{true}]\!] \triangleq \lambda x y . x\,\mathsf{id} \qquad [\![\mathsf{false}]\!] \triangleq \lambda x y . y\,\mathsf{id}$$

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

$$[\![\lambda x_1 \ldots x_n . e]\!] \triangleq \lambda x_1 . [\![\lambda x_2 \ldots x_n . e]\!],\ n \geq 2 \qquad [\![\lambda x . e]\!] \triangleq \lambda x . [\![e]\!] \qquad [\![e_0\, e_1]\!] \triangleq [\![e_0]\!]\,[\![e_1]\!]$$

$$[\![\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!] \triangleq [\![e_0]\!]\,(\lambda z . [\![e_1]\!])\,(\lambda z . [\![e_2]\!]) \qquad [\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!] \triangleq (\lambda x . [\![e_2]\!])\,[\![e_1]\!]$$

$$[\![\mathsf{letrec} \ldots]\!] \triangleq (\textit{to be continued})$$

3.2 Tuples

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

$$\begin{aligned}
[\![()]\!] &\triangleq \lambda xy.\, x \\
[\![(e_1, e_2, \ldots, e_n)]\!] &\triangleq (\lambda xyb.\, b\ x\ y)\ [\![e_1]\!]\, [\![(e_2, \ldots, e_n)]\!] \\
[\![\#1\, e]\!] &\triangleq [\![e]\!]\ (\lambda xy.\, x) \\
[\![\#n\, e]\!] &\triangleq [\![e]\!]\ (\lambda xy.\, [\![\#(n-1)\ y]\!]) \qquad\qquad n > 1
\end{aligned}$$

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

$$\mathsf{letrec}\ f_1 = \lambda x_1.\, e_1\ \mathsf{and}\ \ldots\ \mathsf{and}\ f_n = \lambda x_n.\, e_n\ \mathsf{in}\ e.$$

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:

$$\mathsf{letrec}\ f = \lambda x.\, e_1\ \mathsf{in}\ e \quad \to \quad e\{(\lambda x.\, e_1)\{\mathsf{letrec}\ f = \lambda x.\, e_1\ \mathsf{in}\ f/f\}/f\}. \tag{1}$$

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

$$\mathsf{letrec}\ f = \lambda x.\, e_1\ \mathsf{in}\ f \qquad\qquad (\lambda x.\, e_1)\{\mathsf{letrec}\ f = \lambda x.\, e_1\ \mathsf{in}\ f/f\} \tag{2}$$

appearing on the right-hand side of the rule (1). Both expressions represent the recursive function being defined, and the latter is a value. The former reduces to the latter under the rule (1):

$$\begin{aligned}
\mathsf{letrec}\ f = \lambda x.\, e_1\ \mathsf{in}\ f \quad &\xrightarrow{1} \quad f\{(\lambda x.\, e_1)\{\mathsf{letrec}\ f = \lambda x.\, e_1\ \mathsf{in}\ f/f\}/f\} \\
&= \quad (\lambda x.\, e_1)\{\mathsf{letrec}\ f = \lambda x.\, e_1\ \mathsf{in}\ f/f\}.
\end{aligned}$$

The substitution of $\mathsf{letrec}\ f = \lambda x.\, e_1\ \mathsf{in}\ f$ for free occurrences of f in $\lambda x.\, e_1$ 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

$$\begin{aligned}
(\mathsf{letrec}\ f = \lambda x.\, e_1\ \mathsf{in}\ f)\ v \quad &\xrightarrow{1} \quad ((\lambda x.\, e_1)\{\mathsf{letrec}\ f = \lambda x.\, e_1\ \mathsf{in}\ f/f\})\ v \\
&= \quad (\lambda x.\, (e_1\{\mathsf{letrec}\ f = \lambda x.\, e_1\ \mathsf{in}\ f/f\}))\ v \quad (\text{assuming } f \neq x) \\
&\xrightarrow{1} \quad e_1\{\mathsf{letrec}\ f = \lambda x.\, e_1\ \mathsf{in}\ f/f\}\{v/x\}.
\end{aligned}$$

If $f = x$, then f has no free occurrences in $\lambda x.\, e_1$, so the expressions (2) both reduce to $\lambda x.\, e_1$. In this case the letrec construct is equivalent to the ordinary nonrecursive let.

For the translation to λ-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

$$[\![\mathsf{letrec}\ f = \lambda x.\, e_1\ \mathsf{in}\ e]\!] \quad \stackrel{\triangle}{=} \quad (\lambda f.\, [\![e]\!])\,(Y(\lambda f.\, [\![\lambda x.\, e_1]\!])).$$

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

References