To this point, we have only talked about small languages that felt a bit like toys. Imp had some basic programming constructs we expect, but was conspicuously missing functions. On the other hand λ-calculus was conspicuously missing everything but functions. We saw how to encode a variety of useful constructs in λ-calculus, but we will now go a step farther and augment the language itself with more constructs.

This new functional language Fl is a richer language than anything we have seen and is something we might actually be willing 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 primitives:

- natural number constants n,
- primitive booleans true and false, and
- a letrec construct for constructing recursive functions.

All of these will be language primitives. That is, they are given as part of the syntax, not encoded by other constructs. We could also include arithmetic and boolean operators as before, but for simplicity of exposition, we will include only conditional if statements. Note that these are functional-style if statements which means they return whatever value the chosen branch returns.

Expressions. Fl is an expression language, so there is only one kind of expression. The syntax is as follows.

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

Here n must be strictly positive in projections $\#n\ e$, λ-abstractions $\lambda x_1 \ldots x_n.\, e$, and letrec constructs.

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, so 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. That is, letrec allows us to define arbitrary *mutually recursive* functions.

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, which we will call *stuck* terms. The grammar for values is as follows.

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

2 Operational Semantics of Fl

We will define our operational semantics by a set of evaluation order rules and a set of reduction rules. With all of these extra programming constructs, the power of evaluation contexts to concisely specify evaluation order rules becomes readily apparent.

For evaluation order rules, we will define our evaluation contexts so that evaluation is left-to-right, in applicative order (like CBV), and deterministic

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

Note that there are no holes in the branches of if expressions. We want to delay evaluation of the branches until we finish evaluating the condition, and then discard the incorrect branch without ever evaluating it.

The operational semantic rule using these evaluation contexts is the standard structural congruence rule.

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}$$

Our reduction rules are as follows. Note that multi-argument functions expect one argument at a time and are implicitly curried. That allows for a simpler semantics as well as partial evaluation, though it prevents the semantics from checking that the correct number of arguments were applied.

$$[\textsc{AppN}]\ \frac{n \geq 2}{(\lambda x_1 \ldots x_n.\, e)\ v \longrightarrow (\lambda x_2 \ldots x_n.\, e)[x_1 \mapsto v]} \qquad [\textsc{App1}]\ \frac{}{(\lambda x.\, e)\ v \longrightarrow e[x \mapsto v]}$$

$$[\textsc{IfT}]\ \frac{}{\mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 \longrightarrow e_1} \qquad [\textsc{IfF}]\ \frac{}{\mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 \longrightarrow e_2}$$

$$[\textsc{Proj}]\ \frac{1 \leq n \leq m}{\#n\ (v_1, \ldots, v_m) \longrightarrow v_n} \qquad [\textsc{Let}]\ \frac{}{\mathsf{let}\ x = v\ \mathsf{in}\ e \longrightarrow e[x \mapsto v]}$$

$$[\textsc{LetRec}]\ \frac{}{\mathsf{letrec} \ldots \longrightarrow \textit{(to be continued)}}$$

We can already see the distinction between a value and an irreducible term. For example, what happens with the expression if 3 then 1 else 5 or #3 (true, false, 0)? Those expressions are not values, but they also cannot be reduced further. They are *stuck*. Unlike in λ-calculus, not all expressions work in all contexts.

In a real programming language, these examples might produce a *runtime type error*. For that, we would need a notion of types, which wee will see later in the course.

3 Translation to λ-calculus

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 we will use Church numerals and Church booleans, modified to thunk and apply their arguments—as we did in when translating CBN to CBV—to avoid evaluating the branches of if statements early.

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

We can project multi-argument functions by making the currying explicit, single-argument functions by translating their bodies, function application remains function application, if translates based on the encoding of true and false above, and let is simply a desugaring operation.

$$[\![\lambda x_1 \ldots x_n.\, e]\!] \triangleq \lambda x_1.\, [\![\lambda x_2 \ldots x_n.\, e]\!] \text{ for } n \geq 2 \qquad [\![\lambda x.\, e]\!] \triangleq \lambda x.\, [\![e]\!] \qquad [\![e_1\ e_2]\!] \triangleq [\![e_1]\!]\ [\![e_2]\!]$$

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

Tuples. To project arbitrary length tuples, we will rely on the our existing knowledge of pairs, and how to extend that to encoding lists. Specifically, we will project the tuple $(e_1, \ldots, e_n)$ to the list $[e_1; \ldots; e_n]$. To do so, we will use the list constructs in the homework: empty, cons, head, tail, and get.

$$[\![()]\!] \triangleq \mathsf{empty} \qquad [\![(e_1, \ldots, e_n)]\!] \triangleq \mathsf{cons}\ [\![e_1]\!]\ [\![(e_2, \ldots, e_n)]\!] \text{ for } n \geq 1 \qquad [\![\#n\ e]\!] \triangleq \mathsf{get}\ [\![n]\!]\ [\![e]\!]$$

We again leave recursive definitions for later.

Adequacy. It would be great if this translation were adequate. Unfortunately, the presence of stuck terms in FL and the lack of a runtime type system mean it is not. For instance, #1 () is stuck in the FL operational syntax, but not in the λ-calculus translation. That translated term may not behave in a reasonable way, but the fact that it can step at all makes the translation unsound.

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. That is, we need to know the semantics of

$$\mathsf{letrec}\ f = \lambda x.\, e_1\ \mathsf{in}\ e_2.$$

4.1 Operational Semantics

We would like to substitute $\lambda x.\, e_1$ into e_2 for each occurrence of f, but the whole point of the recursive definition is that f may itself be free in e_1 (and thus $\lambda x.\, e_1$). We therefore need to retain the definition of f somehow in the substitution.

To retain the definition of f, we substitute every instance of f in e_1 with something to retain its definition. In particular, we use $\mathsf{letrec}\ f = \lambda x.\, e_1\ \mathsf{in}\ f$. This expression defines the recursive function f and then simply returns it. This approach allows us to define the semantics of letrec as follows.

$$[\textsc{LetRec}]\ \frac{}{\mathsf{letrec}\ f = \lambda x.\, e_1\ \mathsf{in}\ e_2 \longrightarrow e_2\big[f \mapsto (\lambda x.\, e_1)[f \mapsto \mathsf{letrec}\ f = \lambda x.\, e_1\ \mathsf{in}\ f]\big]}$$

With this semantic rule, we note something interesting about the body of the substitution:

$$\mathsf{letrec}\ f = \lambda x.\, e_1\ \mathsf{in}\ f \longrightarrow (\lambda x.\, e_1)[f \mapsto \mathsf{letrec}\ f = \lambda x.\, e_1\ \mathsf{in}\ f].$$

Looking back, this gives us confidence that the semantic rule is doing the right thing. The inner substitution is replacing every instance of f in e_1 with something that will behave identically to the outer substitution when it is used, exactly the behavior we want from a recursive function.

Note also that if $f = x$, then $\lambda x.\, e_1$ has no free occurrences of f, so the inner substitution does nothing and this definition devolves into a regular (non-recursive) let expression.

Expanding this to multiple mutually-recursive functions requires a parallel substitution of all of the recursive function names, and each inner substitution must include the entire letrec, not just the relevant function. This is a conceptually simple extension, but one that is extremely long to write down formally.

4.2 Translation to λ-calculus

For the translation, recall the fixed point combinators from the previous lecture. Note that, instead of doing this nested substitution, if we could define f directly as a recursive function, we could substitute that into e_2 immediately. Using the Z combinator, as it is call-by-value friendly, can accomplish this goal. That is, $Z(\lambda f.\, \lambda x.\, e_1)$ will be the desired fixed point. From here, we can use the translation of a non-recursive let expression to get

$$[\![\mathsf{letrec}\ f = \lambda x.\, e_1\ \mathsf{in}\ e_2]\!] \triangleq (\lambda f.\, [\![e_2]\!])\ (Z\ (\lambda f.\, [\![\lambda x.\, e_1]\!])).$$

Extending this to multiple mutually-recursive functions requires a mutual recursion combinator that is considerably more complicated than Y or Z, but does exist.