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 $\text{true}$ and $\text{false}$, and a $\text{letrec}$ 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:

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

where $n$ is strictly positive in projections $\# n e$, $\lambda$-abstractions $\lambda x_1 \ldots x_n.e$, and the $\text{letrec}$ 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 $\text{let}$ and $\text{letrec}$ 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 $\text{let } x = e_1 \text{ in } e_2$ is $e_2$ (but not $e_1$!), and the scope of $f_i$ in $\text{letrec } f_1 = \lambda x_1. e_1 \text{ and } \ldots \text{ and } f_n = \lambda x_n. e_n \text{ in } e$ 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 irreducible. There will be other irreducible terms that are not values; this will be the stuck values.

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

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.

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

1
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:

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

2.2 Reductions

$$
(\lambda x_1 \ldots x_n . e) v \rightarrow (\lambda x_2 \ldots x_n . e) \{v/x_1\}, \ n \geq 2
$$

$$
(\lambda x . e) v \rightarrow e \{v/x\}
$$

$$
\#n (v_1, \ldots , v_m) \rightarrow v_n, \text{ where } n \leq m
$$

$$
\text{if } \text{true} \text{ then } e_1 \text{ else } e_2 \rightarrow e_1
$$

$$
\text{if } \text{false} \text{ then } e_1 \text{ else } e_2 \rightarrow e_2
$$

$$
\text{let } x = v \text{ in } e \rightarrow e \{v/x\}
$$

$$
\text{letrec } \ldots \rightarrow (\text{to be continued})
$$

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 (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,

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

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

$$
[\lambda x_1 \ldots x_n . e] \triangleq \lambda x_1 \ldots \lambda x_n . [e], \ n \geq 2
$$

$$
[\lambda x . e] \triangleq \lambda x . [e]
$$

$$
[e_0 \text{ if } e_1 \text{ then } e_2] \triangleq [e_0] (\lambda z . [e_1]) (\lambda z . [e_2])
$$

$$
[\text{let } x = e_1 \text{ in } e_2] \triangleq (\lambda x . [e_2]) [e_1]
$$

$$
[\text{letrec } \ldots] \triangleq (\text{to be continued})
$$
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:

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

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 \texttt{letrec} construct

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

This construct allows us to define mutually recursive functions, each of which is able to call itself and other functions defined in the same \texttt{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 \texttt{letrec}, we will consider only the case $n = 1$ for simplicity of the presentation. The operational semantics is given by the following reduction rule:

\[
\texttt{letrec} \ f = \lambda x. e_1 \ \text{in} \ e \ \rightarrow \ e \{ (\lambda x. e_1) \{ \texttt{letrec} \ f = \lambda x. e_1 \ \text{in} \ f/f \} / f \}.
\]

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

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

The substitution of $\texttt{letrec} \ f = \lambda x. e_1 \ \text{in} \ f$ 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

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

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 \texttt{letrec} construct is equivalent to the ordinary nonrecursive \texttt{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

$$\text{letrec } f = \lambda x. e_1 \text{ in } e \triangleq (\lambda f. [e]) (Y(\lambda f. [\lambda x. e_1])).$$

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

### 5 Strong Typing

The translation of $\text{FL}$ to $\lambda$-CBV presented in Lecture 11 is not sound, because there are many stuck terms in $\text{FL}$ that translate to terms of $\lambda$-CBV that are not stuck. For example, consider the stuck $\text{FL}$ expression $\text{if } 3 \text{ then } 1 \text{ else } 0$. It is stuck because there is no rule of the small-step semantics of $\text{FL}$ that applies. However, its image $\llbracket \text{if } 3 \text{ then } 1 \text{ else } 0 \rrbracket$ is not stuck—it reduces to a value under the CBV rules. In fact, there is no way for a closed term to get stuck in the CBV or CBN $\lambda$-calculus. However, this value does not correspond to the stuck non-value if $3 \text{ then } 1 \text{ else } 0$ in the $\text{FL}$ language. It is meaningless gibberish.

All reasonably powerful languages confront this problem in one way or another, but there is more than one approach to dealing with it. A language in which no term can get stuck during evaluation is said to be **strongly typed**. There is no way to apply an operation to a value of the wrong type. Note that strong typing and static typing are not the same property. For example, the language C is statically typed (the compiler figures out types for all expressions), but it is possible to write code that gets stuck, such as the following:

```c
int a[4]; a[4] = 2;
```

What this code does depends on what machine it is compiled on and what compiler options are used. For example, it might result in the variable $x$ holding the value 2, or perhaps some other variable or even the return address register containing that value. The program may compute the wrong results, crash, or do something completely unpredictable, such as jumping to memory address 2 and executing code.

In C, when an expression is evaluated whose results are not defined by the semantics, either the outcome is “implementation-defined” or else the program is an incorrect C program. Experience has shown that this is not necessarily a good idea, especially when it comes to building secure systems. One might assert that a good programmer would never write such code, but that is of little consolation if the system is successfully attacked by a buffer overrun that exploits implementation-defined behavior to jump to code controlled by the attacker.

Some statically typed languages are strongly typed. Examples include Java and the various ML languages. And some languages that are not statically typed are strongly typed, such as Scheme. And finally, some languages, such as Forth and assembly code, are neither strongly nor statically typed.

Even in languages like OCaml that are statically typed, there are terms that are stuck unless we define some kind of runtime type checking. For example, the expression $0/0$ causes a runtime error. Runtime checking is needed to provide well-defined behavior in these cases.

### 5.1 Runtime Type Checking

As defined, $\text{FL}$ is not explicitly a strongly typed language. We can solve this problem by extending the operational semantics with rules that reduce all stuck expressions to a special error value $\text{error}$. The new
term **error** represents a runtime error. This term cannot occur in a well-formed program, but may arise during evaluation whenever an otherwise stuck expression occurs.

We implement runtime type checking for FL by building a translation from FL to itself. The effect will be that when this new translation is layered on top of the translation above, the resulting target \(\lambda\)-CBV program will faithfully and soundly represent evaluation of the original FL program. And the work done in the translated code arguably does a better job of showing what happens in such a language than the operational semantics does.

To build a sound translation, we will need a representation of the **error** value. More generally, we will need to be able to tell what kind of value we have when an operation is to be applied, so we can catch values of the wrong type. The idea is to tag each value with an integer representing its type. We could use 0 to tag the error value, 1 to tag null, etc. The actual values do not matter, as long as they are distinct. Let us give them symbolic names:

\[
\begin{align*}
\text{Err} &\triangleq 0 \\
\text{Null} &\triangleq 1 \\
\text{Bool} &\triangleq 2 \\
\text{Num} &\triangleq 3 \\
\text{Tuple} &\triangleq 4 \\
\text{Fun} &\triangleq 5
\end{align*}
\]

We use tags to check that we are getting the right kind of values where they are expected. For example, we could check that we have a Boolean value for the test in a conditional if-then-else construct by testing that the value’s tag is 2.

Let us call the new translation \(E[e]\), where the \(E\) stands for “error”. Define translations of the various constructor forms as follows, tagging values appropriately:

\[
\begin{align*}
E[t] &\triangleq (\text{Bool}, t), \quad t \in \{\text{true}, \text{false}\} \\
E[\text{Null}] &\triangleq (\text{Null}, \text{nil}) \\
E[n] &\triangleq (\text{Num}, n), \quad n \in \mathbb{N} \\
E[(e_1, \ldots, e_n)] &\triangleq (\text{Tuple}, n, (E[e_1], \ldots, E[e_n])), \quad n \geq 1 \\
E[\text{error}] &\triangleq (\text{Err}, \text{error}) \\
E[\lambda x_1, \ldots, x_n. e] &\triangleq (\text{Fun}, x_1, \ldots, x_n. E[e])
\end{align*}
\]

Each value is paired with a tag denoting its runtime type. In addition, tuples are tagged with their length so that when a projection \(#n\) is applied, it can be checked that \(n\) is no larger than the length of the tuple.

The translation of other terms needs to check tags. For example, we can translate a conditional as follows, checking the value of the test to make sure it is a Boolean:

\[
E[\text{if } e_0 \text{ then } e_1 \text{ else } e_2] \triangleq \begin{cases} \\
\text{let } z = E[e_0] \text{ in} \\
\quad \text{if } #1 z = \text{Bool} \\
\quad \quad \text{then if } #2 z \text{ then } E[e_1] \text{ else } E[e_2] \\
\quad \text{else } E[\text{error}] \\
\end{cases}
\]

where \(z \notin \text{FV}(e_1) \cup \text{FV}(e_2)\).

If we had arithmetic operators, we could do the same thing for arithmetic:

\[
E[e_1 + e_2] \triangleq \begin{cases} \\
\text{let } z_1 = E[e_1] \text{ in} \\
\text{let } z_2 = E[e_2] \text{ in} \\
\quad \text{if } #1 z_1 = \text{Num} \\
\quad \quad \text{then if } #1 z_2 = \text{Num} \\
\quad \quad \quad \text{then } (\#2 z_1) + (\#2 z_2) \\
\quad \quad \text{else } E[\text{error}] \\
\quad \text{else } E[\text{error}] \\
\end{cases}
\]

where \(z_1 \notin \text{FV}(e_2)\).
The rule for function application checks that the entity being applied as a function is actually a function:

\[
\mathcal{E}[e_0 e_1] \triangleq \begin{cases} 
\text{let } z = \mathcal{E}[e_0] \text{ in } \\
\quad \text{if } \#1 z = \text{Fun} \text{ then } \#2 z \mathcal{E}[e_1] \text{ else } \mathcal{E}[\text{error}] 
\end{cases}
\]

where \( z \not\in \text{FV}(e_1) \).

Of course, we will need more translation rules for the various other constructs. The rule for projection checks that it is in bounds:

\[
\mathcal{E}[\#n e] \triangleq \begin{cases} 
\text{let } z = \mathcal{E}[e] \text{ in } \\
\quad \text{if } \#1 z = \text{Tuple} \text{ then } \#n (\#3 z) \\
\quad \text{else } \mathcal{E}[\text{error}] 
\end{cases}
\]

We do not need to check that \( \#2 z \) is a number, because that is true whenever the first component is \( \text{Tuple} \), as guaranteed by the translation. Likewise, we do not need to check \( n \geq 1 \) because that is guaranteed by the syntax of FL.

6 Summary

We have made FL strongly typed using runtime type checking. However, this does not really solve the problem of unexpected values arising at runtime; it merely converts unpredictable behavior into a predictable error value.

We can further improve the situation by introducing an exception mechanism that allows a program to catch error conditions and handle them in some graceful way. In general, however, it is difficult for programs to handle errors effectively, even with an exception mechanism.

Another approach is to use static (compile-time) reasoning supported by a type system that rules out certain stuck expressions. This reduces the cost associated with runtime type checking and ensures that certain errors cannot occur. However, type systems can never be expressive enough to rule out all unexpected expressions, because it is impossible in general to predict the values of expressions at compile time. We will have more to say about type systems later in the course.

References