1 Definition of FL

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.

In addition to λ-abstractions, we also introduce tuples $(e_1, \ldots, e_n)$, numbers n, booleans, and a value null corresponding to OCaml () or Java null.

1.1 Expressions

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

1.2 Values

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

1.3 Evaluation Contexts

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

$$\begin{array}{rcl} E & ::= & [\cdot] \mid v_0 \cdots v_m\, E\, e_{m+2} \cdots e_n \mid \#n\, E \mid \mathsf{if}\ E\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \\ & & \mid \mathsf{let}\ x = E\ \mathsf{in}\ e \mid (v_1, \ldots, v_m, E, e_{m+2}, \ldots, e_n) \end{array}$$

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:

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

1.4 Reductions

$$\begin{array}{rcl} (\lambda x_1 \ldots x_n . e)\, v_1 \cdots v_n & \to & e\{v_1/x_1\}\{v_2/x_2\} \cdots \{v_n/x_n\} \\ n_1 \oplus n_2 & \to & n_3 \quad \text{if } n_1 \oplus n_2 = n_3 \text{ under the corresponding arithmetic operation } \oplus \\ \#n\,(v_1, \ldots, v_m) & \to & v_n, \quad \text{where } 1 \le n \le m \\ \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 & \to & e_1 \\ \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 & \to & e_2 \\ \mathsf{let}\ x = v\ \mathsf{in}\ e & \to & e\{v/x\} \\ \mathsf{letrec} \ldots & \to & \textit{to be continued} \end{array}$$

We can already see that there will be problems with soundness. For example, what happens with the expression $\mathsf{if}\ 3\ \mathsf{then}\ 1\ \mathsf{else}\ 0$? The evaluation is stuck, because there is no reduction rule that applies to this term, but it is not a value. Unlike the λ-calculus, not all terms work in all contexts. We do not have an explicit notion of *type* in this language; the types are simply the different kinds of expression forms that can appear on the left-hand side of the various reduction rules. Nevertheless, we consider expressions that are stuck to contain a *runtime type error*.

2 Translating FL to λ-CBV

2.1 Application and Abstraction, Numbers and Booleans

To capture the semantics of FL, we can also translate it to the call-by-value λ-calculus. Here are some of the basic translation rules:

$$
\begin{aligned}
\llbracket \lambda x_1 \ldots x_n .\, e \rrbracket &\triangleq \lambda x_1 \ldots x_n .\, \llbracket e \rrbracket \\
\llbracket e_0 \cdots e_n \rrbracket &\triangleq \llbracket e_0 \rrbracket\, \llbracket e_1 \rrbracket\, \llbracket e_2 \rrbracket \cdots \llbracket e_n \rrbracket \\
\llbracket x \rrbracket &\triangleq x \\
\llbracket n \rrbracket &\triangleq \lambda f x .\, f^n x \\
\llbracket \mathsf{null} \rrbracket &\triangleq \mathsf{id} \\
\llbracket \mathsf{true} \rrbracket &\triangleq \lambda x y .\, x\, \mathsf{id} \\
\llbracket \mathsf{false} \rrbracket &\triangleq \lambda x y .\, y\, \mathsf{id} \\
\llbracket \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \rrbracket &\triangleq \llbracket e_0 \rrbracket\, (\lambda z .\, \llbracket e_1 \rrbracket)\, (\lambda z .\, \llbracket e_2 \rrbracket).
\end{aligned}
$$

Notice that we implement booleans by combining our earlier λ-calculus implementation of booleans with the delayed-evaluation trick employed in the translation from CBN to CBV λ-calculus.

2.2 Tuples

Let us consider the translation of tuples. We have already seen how to represent lists in the λ-calculus using the functions list, head, tail, nil, and empty with the following properties:

$$
\begin{aligned}
\mathsf{head}\,(\mathsf{list}\, e_1\, e_2) &= e_1 \\
\mathsf{tail}\,(\mathsf{list}\, e_1\, e_2) &= e_2 \\
\mathsf{empty}\,(\mathsf{list}\, e_1\, e_2) &= \mathsf{false} \\
\mathsf{empty}\, \mathsf{nil} &= \mathsf{true}.
\end{aligned}
$$

Using these constructs, we can define the translation from tuples to λ-CBV as follows:

$$
\begin{aligned}
\llbracket (\,) \rrbracket &\triangleq \mathsf{nil} \\
\llbracket (e_1, e_2, \ldots, e_n) \rrbracket &\triangleq \mathsf{list}\, \llbracket e_1 \rrbracket\, \llbracket (e_2, \ldots, e_n) \rrbracket \\
\llbracket \# n\, e \rrbracket &\triangleq \mathsf{head}\, (\mathsf{tail}^{n-1}\, \llbracket e \rrbracket).
\end{aligned}
$$

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

2.3 Let and Letrec

Now come the last of our FL constructs, let and letrec. For let expressions, we define

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

The letrec construct

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

allows us to define mutually recursive functions, each of which is able to call itself and other functions defined in the same letrec block. We will consider only the case $n = 1$. Recall 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

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

3 Strong Typing

Revisiting our earlier example, if 3 then 1 else 0, we see that the translation to λ-CBV is not sound, because its image $[\![\mathsf{if}\ 3\ \mathsf{then}\ 1\ \mathsf{else}\ 0]\!]$ reduces to a value under the CBV rules—there is no way for a closed term to get stuck in the CBV or CBN λ-calculus. However, this value does not correspond to the stuck non-value if 3 then 1 else 0 in the 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:

```
int x = 1;
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. That the buck has been passed to the programmer 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.

3.1 Runtime Type Checking

As defined, 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 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 λ-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:

$$\begin{array}{lll} \mathsf{Err} \triangleq 0 & \mathsf{Null} \triangleq 1 & \mathsf{Bool} \triangleq 2 \\ \mathsf{Num} \triangleq 3 & \mathsf{Tuple} \triangleq 4 & \mathsf{Func} \triangleq 5 \end{array}$$

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 $\mathcal{E}[\![e]\!]$, where the $\mathcal{E}$ stands for "error". Define translations of the various constructor forms as follows, tagging values appropriately:

$$\begin{array}{rcl@{\qquad}rcl} \mathcal{E}[\![\mathsf{null}]\!] & \triangleq & (\mathsf{Null}, \mathsf{null}) & \mathcal{E}[\![t]\!] & \triangleq & (\mathsf{Bool}, t),\ t \in \{\mathsf{true}, \mathsf{false}\} \\ \mathcal{E}[\![n]\!] & \triangleq & (\mathsf{Num}, n) & \mathcal{E}[\![(e_1, \ldots, e_n)]\!] & \triangleq & (\mathsf{Tuple}, n, (\mathcal{E}[\![e_1]\!], \ldots, \mathcal{E}[\![e_n]\!])) \\ \mathcal{E}[\![\mathsf{error}]\!] & \triangleq & (\mathsf{Err}, \mathsf{error}) & \mathcal{E}[\![\lambda x_1, \ldots, x_n \,.\, e]\!] & \triangleq & (\mathsf{Func}, n, \lambda x_1, \ldots, x_n \,.\, \mathcal{E}[\![e]\!]) \end{array}$$

Each value is paired with an indication of its runtime type. In addition, λ-abstractions are tagged with the number of their arguments, so that the argument count can be checked when the abstraction is applied. The translation of other terms needs to check tags. For example, we can translate if as follows, checking the value of the conditional to make sure it has the boolean tag 2:

$$\begin{array}{rcl} \mathcal{E}[\![\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!] & \triangleq & \mathsf{let}\ z = \mathcal{E}[\![e_0]\!]\ \mathsf{in} \\ & & \mathsf{if}\ \#1\, z = \mathsf{Bool} \\ & & \quad \mathsf{then}\ (\mathsf{if}\ \#2\, z\ \mathsf{then}\ \mathcal{E}[\![e_1]\!]\ \mathsf{else}\ \mathcal{E}[\![e_2]\!]) \\ & & \quad \mathsf{else}\ \mathcal{E}[\![\mathsf{error}]\!] \end{array}$$

where $z \notin FV(e_1) \cup FV(e_2)$.

Similarly, for arithmetic:

$$\begin{array}{rcl} \mathcal{E}[\![e_1 + e_2]\!] & \triangleq & \mathsf{let}\ z_1 = \mathcal{E}[\![e_1]\!]\ \mathsf{in} \\ & & \mathsf{let}\ z_2 = \mathcal{E}[\![e_2]\!]\ \mathsf{in} \\ & & \mathsf{if}\ (\#1\, z_1 = \mathsf{Num}) \wedge (\#1\, z_2 = \mathsf{Num}) \\ & & \quad \mathsf{then}\ (\#2\, z_1) + (\#2\, z_2) \\ & & \quad \mathsf{else}\ \mathcal{E}[\![\mathsf{error}]\!] \end{array}$$

where $z_1, z_2 \notin FV(e_1) \cup FV(e_2)$.

Of course, we will need more translation rules for the various other constructs. The rule for function application checks that the number of actual parameters matches the number of formal parameters:

$$\begin{array}{rcl} \mathcal{E}[\![e_0\, e_1\, \cdots\, e_n]\!] & \triangleq & \mathsf{let}\ z = \mathcal{E}[\![e_0]\!]\ \mathsf{in} \\ & & \mathsf{if}\ (\#1\, z = \mathsf{Func}) \wedge (\#2\, z = n) \\ & & \quad \mathsf{then}\ \#3\, z\, \mathcal{E}[\![e_1]\!] \cdots \mathcal{E}[\![e_n]\!] \\ & & \quad \mathsf{else}\ \mathcal{E}[\![\mathsf{error}]\!] \end{array}$$

where $z \notin FV(e_i)$ for $i \in \{0, \ldots, n\}$.

4 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.