1 Untyped ML (uML)

Let's use the idea of semantics by translation for a richer language, one we might actually like to program in. We augment the λ-calculus with some more conventional programming constructs We call this language uML since it resembles ML, with the "u" standing for "untyped"[1]. We'll give semantics for this language in two ways, first through a structural operational semantics, and second through a translation to the CBV λ-calculus.

In addition to lambda abstractions, we also introduce pairs (e_1, e_2), numbers n, booleans, and a value **null** corresponding to ML () or Java **null**.

1.1 Expressions

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

1.2 Values

$$v \quad ::= \quad \lambda x_1 \ldots x_n . e \mid n \mid \textbf{true} \mid \textbf{false} \mid \textbf{null} \mid (v_1, v_2)$$

1.3 Evaluation Contexts

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

$$\begin{aligned} E \quad ::= \quad & [\,\cdot\,] \mid v_0 \ \ldots \ v_m \ E \ e_{m+2} \ \ldots \ e_n \mid \textbf{let } (x, y) = E \textbf{ in } e \\ & \mid \textbf{if } E \textbf{ then } e_1 \textbf{ else } e_2 \\ & \mid \textbf{let } x = E \textbf{ in } e \\ & \mid (E, e) \mid (v, E) \end{aligned}$$

Note that there are no holes on the right-hand side of **if** because it evaluates the consequent and alternative expressions lazily. Even in an eager, call-by-value language, we want *some* laziness.

The structural congruence rule takes the usual form:

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

[1] uML is not to be confused with UML, the Unified Modeling Language

1.4 Reductions

$$
\begin{array}{rcll}
(\lambda x_1 \ldots x_n.\, e)\ v_1\ \ldots\ v_n & \longrightarrow & e\{v_1/x_1\}\{v_2/x_2\}\ldots\{v_n/x_n\} & \\
v_1 \oplus v_2 & \longrightarrow & v_3 & \text{if } n_3 = n_1 \oplus n_2, \text{ where } \oplus \text{ represents the corresponding mathematical operation.} \\
\#1\ (v_1, v_2) & \longrightarrow & v_1 & \\
\#2\ (v_1, v_2) & \longrightarrow & v_2 & \\
\textbf{if true then } e_1 \textbf{ else } e_2 & \longrightarrow & e_1 & \\
\textbf{if false then } e_1 \textbf{ else } e_2 & \longrightarrow & e_2 & \\
\textbf{let } x = v \textbf{ in } e & \longrightarrow & e\{v/x\} & \\
\textbf{letrec } \ldots & \longrightarrow & \textit{to be continued} &
\end{array}
$$

We can already see that there will be problems with establishing soundness. For example, what happens with the expression $\textbf{if } 3 \textbf{ then } 1 \textbf{ else } 0$? The evaluation is stuck, because there is no reduction rule that applies to this term. Unlike in the lambda calculus, not all terms work in all contexts. We don't 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 *run-time type error*.

1.5 Translating uML to the CBV λ-Calculus

To capture the semantics of uML, we can also translate it to the call-by-value lambda calculus. We define some of the translation rules:

$$
\begin{array}{rcl}
[\![\lambda x_1 \ldots x_n.\, e]\!] & = & \lambda x_1 \ldots x_n.\, [\![e]\!] \\
[\![e_0\ \ldots\ e_n]\!] & = & [\![e_0]\!]\ [\![e_1]\!]\ [\![e_2]\!]\ \ldots\ [\![e_n]\!] \\
[\![x]\!] & = & x \\
[\![n]\!] & = & \overline{n} = \lambda f x.\, f^n x \\
[\![\textbf{null}]\!] & = & I \\
[\![\textbf{true}]\!] & = & \lambda x y.\, x\ I \\
[\![\textbf{false}]\!] & = & \lambda x y.\, y\ I \\
[\![\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2]\!] & = & [\![e_0]\!]\ (\lambda z.\, [\![e_1]\!])\ (\lambda z.\, [\![e_2]\!]).
\end{array}
$$

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

Let us consider the translation of pairs. We have already seen how to represent pairs in the λ-calculus using functions *PAIR*, *FIRST*, and *SECOND*, with the following properties:

$$
\begin{array}{rcl}
\textit{FIRST}\ (\textit{PAIR}\ e_1\ e_2) & = & e_1 \\
\textit{SECOND}\ (\textit{PAIR}\ e_1\ e_2) & = & e_2 \\
\textit{PAIR}\ (\textit{FIRST}\ e)\ (\textit{SECOND}\ e) & = & e
\end{array}
$$

Using these constructs, we can define the translation from pairs to λ_{CBV} straightforwardly as follows:

$$
\begin{array}{rcll}
[\![(e_1, e_2)]\!] & = & \textit{PAIR}\ [\![e_1]\!]\ [\![e_2]\!] = \lambda f.\, f\ [\![e_1]\!]\ [\![e_2]\!] & \\
[\![\textbf{let } (x, y) = e_1 \textbf{ in } e_2]\!] & = & e_1\ (\lambda x y.\, e_2) & \text{(using the properties of } \textit{PAIR}\text{)}
\end{array}
$$

For **let** expressions, we define

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

Now comes the last of our uML constructs, **letrec**:

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

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. We consider only case $n = 1$, saving the general case for later when we have a translation with better control over naming. 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

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

2 Strong Typing

Revisiting our earlier example, $\mathbf{if}\ 3\ \mathbf{then}\ 1\ \mathbf{else}\ 0$, we see that the translation to CBV is not sound, because its image $[\![\mathbf{if}\ 3\ \mathbf{then}\ 1\ \mathbf{else}\ 0]\!]$ 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, as we proved previously. However, this value does not correspond to the stuck non-value $\mathbf{if}\ 3\ \mathbf{then}\ 1\ \mathbf{else}\ 0$ in the uML language. It is 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. Notice 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's 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 a program results in evaluation expressions whose possible 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 ML that are statically typed, there are terms that are stuck unless we define some kind of run-time type checking. For example, the expression 0/0 causes a run-time error. Run-time checking is needed to provide well-defined behavior in these cases.

2.1 Run-time type checking

As defined, uML 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 $\mathbf{error}$. Writing these rules explicitly captures the need to do *run-time type checking*. The new term $\mathbf{error}$ represents a run-time error. This term cannot occur in a well-formed program, but may arise during evaluation whenever an otherwise stuck expression occurs.

Let us examine run-time type checking by building a translation from strongly-typed uML (call it uML$_{ST}$) to the uML just defined. The effect will be that when this new translation is layered on top of the translation above, the resulting λ_{CBV} program will faithfully and soundly represent evaluation of the original uML$_{ST}$ 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 basic idea is to represent uML$_{ST}$ values as a pair of a *tag* and a uML value. The tag is an integer representing the type. We could use 0 to tag the error value, 1 to tag **null**, 2 to tag booleans, 3 for numbers, 4 for pairs, and 5 for functions, for example. It doesn't really matter what values we choose, as long as they are distinct, so let's give them symbolic names:

$$\begin{array}{rcl@{\qquad}rcl} \mathit{Err} & \triangleq & 0 & \mathit{Null} & \triangleq & 1 \\ \mathit{Bool} & \triangleq & 2 & \mathit{Num} & \triangleq & 3 \\ \mathit{Pair} & \triangleq & 4 & \mathit{Func} & \triangleq & 5 \end{array}$$

We use tags to check that we are getting the right kind of values where they are expected. For example, we would 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 subscript $\mathcal{E}$ stands for "error". Define translations of the various constructor forms as follows, tagging values appropriately:

$$\begin{array}{rcl} \mathcal{E}[\![\mathbf{null}]\!] & = & (\mathit{Null}, \mathbf{null}) = (1, \mathbf{null}) \\ \mathcal{E}[\![\mathbf{true}]\!] & = & (\mathit{Bool}, \mathbf{true}) = (2, \mathbf{true}) \\ \mathcal{E}[\![\mathbf{false}]\!] & = & (\mathit{Bool}, \mathbf{false}) \\ \mathcal{E}[\![n]\!] & = & (\mathit{Num}, n) \\ \mathcal{E}[\![(e_1, e_2)]\!] & = & (\mathit{Pair}, (\mathcal{E}[\![e_1]\!], \mathcal{E}[\![e_2]\!])) \\ \mathcal{E}[\![\lambda x_1, \ldots, x_n.\, e]\!] & = & (\mathit{Func}, (n, \lambda x_1, \ldots, x_n.\, \mathcal{E}[\![e]\!])) \end{array}$$

Each value is paired with an indication of its run-time type. In addition, lambda 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:

$$\begin{array}{rcl} \mathcal{E}[\![\mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]\!] & = & \mathbf{let}\ (t, z) = \mathcal{E}[\![e_0]\!] \\ & & \quad \mathbf{in\ if}\ t = \mathit{Bool} \\ & & \quad\quad \mathbf{then}\ (\mathbf{if}\ z\ \mathbf{then}\ \mathcal{E}[\![e_1]\!]\ \mathbf{else}\ \mathcal{E}[\![e_2]\!]) \\ & & \quad\quad \mathbf{else\ error} \end{array}$$

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

Similarly, for pair operations and arithmetic:

$$\begin{aligned}
\mathcal{E}[\![\mathbf{let}\ x, y = e_1\ \mathbf{in}\ e_2]\!] = \mathbf{let}\ (t, z) = \mathcal{E}[\![e_1]\!]\ \mathbf{in} \quad & \mathbf{if}\ t = \textit{Pair} \\
& \quad \mathbf{then\ let}\ (x, y) = z\ \mathbf{in}\ e_2 \\
& \quad \mathbf{else\ error} \qquad\qquad \text{((where } z \notin FV(e)\text{;))} \\
\mathcal{E}[\![e_1 + e_2]\!] = & \mathbf{let}\ (t_1, z_1) = \mathcal{E}[\![e_1]\!]\ \mathbf{in} \\
& \quad \mathbf{let}\ (t_2, z_2) = \mathcal{E}[\![e_2]\!]\ \mathbf{in} \\
& \quad\ \mathbf{if}\ (t_1 = \textit{Num}) \wedge (t_2 = \textit{Num}) \\
& \quad\ \mathbf{then}\ z_1 + z_2 \\
& \quad\ \mathbf{else\ error} \\
& \text{(where } z \notin FV(e_1) \cup FV(e_2)\text{)}
\end{aligned}$$

Of course, we'll need more translation rules for the various arithmetic and logical operators. The rule for function application is more complicated, because it checks that the number of actual parameters matches the number of formal parameters:

$$\begin{aligned}
\mathcal{E}[\![e_0\ e_1 \ldots e_n]\!] = & \mathbf{let}\ (t_f, p_f) = \mathcal{E}[\![e_0]\!]\ \mathbf{in} \\
& \quad \mathbf{if}\ t_f = \textit{Func}\ \mathbf{then} \\
& \quad \mathbf{let}(n_f, z_f) = p_f \mathbf{in} \\
& \quad\ \mathbf{if}\ n_f = n \\
& \quad\ \mathbf{then}\ z_f\ \mathcal{E}[\![e_1]\!]\ \ldots\ \mathcal{E}[\![e_n]\!] \\
& \quad\ \mathbf{else\ error} \\
& \quad \mathbf{else\ error} \\
& \text{(where } z_f, p_f, n_f \notin FV(e_i) \text{ for } i \in 0..n\text{)}
\end{aligned}$$

3 Summary

We can see that for uML to be strongly typed, a translation to a lower-level language must insert some kind of type information for run-time type checking. However, run-time type checking doesn't really solve the problem of unexpected values cropping up at run time; it merely converts unpredictable behavior into a predictable error value. This can make systems easier to reason about and perhaps more secure, but it doesn't guarantee that they work.

One way to improve the situation is to introduce an *exception* mechanism that allows a program to catch error conditions and handle them in some way. In general, though, it's difficult for programs to handle errors effectively, even with an exception mechanism. We'll see how exceptions and exception handling work in the next few lectures.

Another approach is to use static (compile-time) reasoning, perhaps supported by a static type system that rules out some classes of stuck expressions. This reduces the cost associated with run-time type checking, and more importantly, ensures that certain errors cannot occur. However, type systems can never be expressive enough to rule out all unexpected expressions, and they come at some cost in expressiveness, because they must be conservative. We'll talk about type systems later in the course.