1 Strong Typing

The translation of FL to λ-CBV presented in Lecture 11 is not sound, because there are many stuck terms in FL that translate to terms of λ-CBV that are not stuck. For example, consider the stuck FL expression if 3 then 1 else 0. It is stuck because there is no rule of the small-step semantics of FL that applies. However, its image $[\![$if 3 then 1 else 0$]\!]$ 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 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 thing. 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 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.

1.1 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:

$$\mathsf{Err} \triangleq 0 \qquad \mathsf{Null} \triangleq 1 \qquad \mathsf{Bool} \triangleq 2 \qquad \mathsf{Num} \triangleq 3 \qquad \mathsf{Tuple} \triangleq 4 \qquad \mathsf{Fun} \triangleq 5$$

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:

$$\begin{array}{rcl@{\qquad}rcl}
\mathcal{E}[\![t]\!] & \triangleq & (\mathsf{Bool}, t),\ t \in \{\mathsf{true}, \mathsf{false}\} & \mathcal{E}[\![()]\!] & \triangleq & (\mathsf{Null}, \mathsf{nil}) \\
\mathcal{E}[\![n]\!] & \triangleq & (\mathsf{Num}, n),\ n \in \mathbb{N} & \mathcal{E}[\![(e_1, \ldots, e_n)]\!] & \triangleq & (\mathsf{Tuple}, n, (\mathcal{E}[\![e_1]\!], \ldots, \mathcal{E}[\![e_n]\!])),\ n \geq 1 \\
\mathcal{E}[\![\mathsf{error}]\!] & \triangleq & (\mathsf{Err}, \mathsf{error}) & \mathcal{E}[\![\lambda x_1, \ldots, x_n . e]\!] & \triangleq & (\mathsf{Fun}, \lambda x_1, \ldots, x_n . \mathcal{E}[\![e]\!])
\end{array}$$

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:

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

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

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

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

where $z_1 \notin FV(e_2)$.

The rule for function application checks that the entity being applied as a function is actually a function:

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

where $z \notin FV(e_1)$.

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

$$
\begin{array}{rcl}
\mathcal{E}[\![\#n\, e]\!] & \triangleq & \mathsf{let}\ z = \mathcal{E}[\![e]\!]\ \mathsf{in} \\
& & \mathsf{if}\ \#1\, z = \mathsf{Tuple} \\
& & \quad \mathsf{then\ if}\ n \leq \#2\, z \\
& & \quad\quad \mathsf{then}\ \#n\, (\#3\, z) \\
& & \quad\quad \mathsf{else}\ \mathcal{E}[\![\mathsf{error}]\!] \\
& & \quad \mathsf{else}\ \mathcal{E}[\![\mathsf{error}]\!]
\end{array}
$$

We do not need to check that $\#2\, z$ is a number, because that is true whenever the first component is 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.

2 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.