Exceptions are a language feature that provide for non-local control flow in exceptional situations. They are generally considered a double-edged sword from a software engineering and maintenance perspective: exceptional control flow is harder to understand and reason about (and thus maintain), but factoring out some control flow into an exceptional path often makes it easier to understand and maintain the other control flow paths. As a result, exceptions are typically used to signify and handle abnormal, unexpected, or rarely occurring events and simplify code for the common cases.

To add exceptions to FL, we extend the syntax with two new constructs: raise and try–catch.

$$e \quad ::= \quad \cdots \mid \mathsf{raise}\ s\ e \mid \mathsf{try}\ e_1\ \mathsf{catch}\ s\ (\lambda x.\, e_2)$$

Informally, $\mathsf{raise}\ s\ e$ throws an exception named s with value e. Meanwhile, $\mathsf{try}\ e_1\ \mathsf{catch}\ s\ (\lambda x.\, e_2)$ provides the handler $\lambda x.\, e_2$ for any exception named s raised while executing e_1. In other words, if e_1 terminates normally with value v, the result of $\mathsf{try}\ e_1\ \mathsf{catch}\ s\ (\lambda x.\, e_2)$ will also be v. If it raises an exception named s, the handler $\lambda x.\, e_2$ will be invoked and provided the value of that exception. If it raises an exception with a different name, the exception will propagate through.

Most languages use a dynamic scoping mechanism to find the handler for a given exception. When an exception occurs, the language walks up the runtime call stack until it finds a suitable exception handler. We will take the same approach in FL and see multiple ways to define it.

1 Operational Semantics of Exceptions

One way to formalize the definition of exceptions is to define a small-step operational semantics. Here we will extend the semantics for FL to define the semantics for our new exception terms.

First, we extend our evaluation contexts as follows:

$$E \quad ::= \quad \cdots \mid \mathsf{raise}\ s\ E.$$

Note that we do not include try–catch statements in our evaluation contexts. This is because we want to use our evaluation contexts to succinctly define how exceptions propagate. In particular, we can define the following exception propagation rule.

$$[\textsc{Raise}]\ \frac{E \neq [\cdot]}{E[\mathsf{raise}\ s\ v] \longrightarrow \mathsf{raise}\ s\ v}$$

This rule is what creates the non-local control flow described earlier. Any evaluation context surrounding an exception is simply destroyed without executing any pending computations it specifies. The side condition that $E \neq [\cdot]$ is a technical requirement to prevent and infinite sequence of reduction steps that do nothing by reducing $\mathsf{raise}\ s\ v$ to itself.

Note that, if we included the body of a try–catch statement as an evaluation context, RAISE would bypass the handlers and propagate exceptions in ways it should not. Instead, we include an explicit evaluation order rule along with the other semantic rules for try–catch.

$$[\textsc{TryE}]\ \frac{e_1 \longrightarrow e_1'}{\mathsf{try}\ e_1\ \mathsf{catch}\ s\ (\lambda x.\, e_2) \longrightarrow \mathsf{try}\ e_1'\ \mathsf{catch}\ s\ (\lambda x.\, e_2)} \qquad [\textsc{TryV}]\ \frac{}{\mathsf{try}\ v\ \mathsf{catch}\ s\ (\lambda x.\, e) \longrightarrow v}$$

$$[\textsc{Catch}]\ \frac{}{\mathsf{try}\ (\mathsf{raise}\ s\ v)\ \mathsf{catch}\ s\ (\lambda x.\, e) \longrightarrow e[x \mapsto v]} \qquad [\textsc{NCatch}]\ \frac{s \neq s'}{\mathsf{try}\ (\mathsf{raise}\ s'\ v)\ \mathsf{catch}\ s\ (\lambda x.\, e) \longrightarrow \mathsf{raise}\ s'\ v}$$

The first rule (TryE) is a simple evaluation order rule. The second rule (TryV) says that if the body of the try–catch block terminates normally with a value, there is no catching to be done and the block should return the same value. The other two rules address the case where the body e_1 raises an uncaught exception.

If the name of the raised exception matches the name of the exception this block catches—here s—then Catch applies. Catch applies the provided handler function to the value of in the exception. If the name of the raised exception and the name of the caught exception do *not* match, then NCatch applies and this block simply propagates the exception outward.

There is a slightly subtle decision hiding in the Catch rule. Because Catch discards the try–catch and steps to only the body of the handler, if the handler body itself throws an exception named s, it will not recursively handle itself (though a different handler in a larger try–catch block might).

Note also that we did not change our definition of values, so, in particular raise s v is *not* a value. However, there is no semantic rule for what to do with a top-level raise expression. These globally uncaught exceptions are considered errors, so the semantics simply gets stuck.

2 CPS and Exception Handlers

Another way to cleanly define the semantics of raise and try–catch is to extend the CPS conversion with exception handlers. To do that, we extend the CPS conversion definition from the previous lecture with a *handler environment* h that maps exception names to continuations. The continuation associated with s in h should run the handler specified by the inner-most try–catch block that handles s and pass its result to the outer continuation passed to that try–catch block.

We will again use lookup and update for an environment, this time lookup $h\ \ulcorner s \urcorner$ should return the continuation associated with s in h, and update $h\ \ulcorner s \urcorner\ k$ should rebind s in h to continuation k.

We can now add support for exceptions to the CPS translation. We write this extended translation $\mathcal{E}[\![e]\!]$, and note that it takes both a continuation, as before, and a new exception handler environment. For most of the expressions in Fl, this new translation looks the same as standard CPS conversion, but with h being passed in to nearly everything. Here are the cases for some representative expressions.

$$
\begin{aligned}
\mathcal{E}[\![x]\!]\ k\ h &\triangleq k\ x \\
\mathcal{E}[\![n]\!]\ k\ h &\triangleq k\ n \\
\mathcal{E}[\![e_1 \otimes e_2]\!]\ k\ h &\triangleq \mathcal{E}[\![e_1]\!]\ (\lambda x_1.\, \mathcal{E}[\![e_2]\!]\ (\lambda x_2.\, k\ (x_1 \otimes x_2))\ h)\ h \\
\mathcal{E}[\![\lambda x.\, e]\!]\ k\ h &\triangleq k\ (\lambda x.\, \mathcal{E}[\![e]\!]) \\
&=_\eta k\ (\lambda x k' h'.\, \mathcal{E}[\![e]\!]\ k'\ h') \\
\mathcal{E}[\![e_1\ e_2]\!]\ k\ h &\triangleq \mathcal{E}[\![e_1]\!]\ (\lambda f.\, \mathcal{E}[\![e_2]\!]\ (\lambda x.\, f\ x\ k\ h)\ h)\ h \\
\mathcal{E}[\![\mathsf{true}]\!]\ k\ h &\triangleq k\ \mathsf{true} \\
\mathcal{E}[\![\mathsf{false}]\!]\ k\ h &\triangleq k\ \mathsf{false} \\
\mathcal{E}[\![\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]\ k\ h &\triangleq \mathcal{E}[\![e_0]\!]\ (\lambda b.\, \mathsf{if}\ b\ \mathsf{then}\ (\mathcal{E}[\![e_1]\!]\ k\ h)\ \mathsf{else}\ (\mathcal{E}[\![e_2]\!]\ k\ h))\ h
\end{aligned}
$$

The cases for raise and try–catch make interesting use of h.

$$
\begin{aligned}
\mathcal{E}[\![\mathsf{raise}\ s\ e]\!]\ k\ h &\triangleq \mathcal{E}[\![e]\!]\ (\mathsf{lookup}\ h\ \ulcorner s \urcorner)\ h \\
\mathcal{E}[\![\mathsf{try}\ e_1\ \mathsf{catch}\ s\ (\lambda x.\, e_2)]\!]\ k\ h &\triangleq \mathcal{E}[\![e_1]\!]\ k\ (\mathsf{update}\ h\ \ulcorner s \urcorner\ (\lambda x.\, \mathcal{E}[\![e_2]\!]\ k\ h))
\end{aligned}
$$

The translation of raise s e simply evaluates e and passes the result as an argument to the handler bound to s in the handler environment. The translation of try e_1 catch s $(\lambda x.\, e_2)$ makes a new continuation by

translating the handler $\lambda x.\, e_2$ with the current continuation k and handler environment h, and then updates h to map the specified name s to that new continuation. It then runs the body e_1 with the same continuation passed to the try–catch block and this new updated handler environment.

There are some subtle decisions captured by this translation. First, in the translation of $\mathsf{try}\ e_1\ \mathsf{catch}\ s\ (\lambda x.\, e_2)$, s is in scope in e_1 but not in e_2. Thus, if e_2 raises an exception s, it will not be invoked again, matching the decision made in the operational semantics in Section 1. Second, in the translation of $\mathsf{raise}\ s\ e$, the continuation k disappears completely. That means that whatever computation is pending will simply be ignored and we will instead execute an exception handler—which will include the continuation from outside of the corresponding try–catch block. This behavior also matches something we saw in Section 1: the RAISE rule destroyed the evaluation context $E[\cdot]$, which corresponds to the continuation k.

3 Exceptions with Resumption

The exception mechanism above has the property that raising an exception terminates execution of the evaluation context. Most modern programming languages have exceptions with this *termination semantics*. A different approach to exceptions is to allow execution to continue at the point where the exception was raised, after the exception handler gets a chance to repair the damage. This approach is known as exceptions with *resumption semantics*. In practice it seems to be difficult to use these mechanisms usefully, though a few kinds of systems (and research languages) do support them.

Operating system interrupts are one practical instance of resumption semantics. When a process receives an interrupt, the interrupt handler runs and only then does execution continue from the program point where the interrupt happened.

One reason to use the CPS conversion in Section 2 is that it is far easier to modify it to implement resumption semantics than the operational semantics from Section 1. Operational semantics would need to keep track of the evaluation context $E[\cdot]$ that RAISE currently removes and be able to reestablish it after handling the exception. With the CPS-style semantics, however, the only change is considerably more simple.

To see how this works, we can give resumption-style exceptions to FL with the constructs

$$e \quad ::= \quad \cdots \mid \mathsf{interrupt}\ s\ e \mid \mathsf{run}\ e_1\ \mathsf{handle}\ s\ (\lambda x.\, e_2).$$

The new translation of these constructs treats the handler environment h as a mapping of names to functions that take a value *and a continuation*, not just a continuation. Note that only the exception cases need to change from the translation described in Section 2.

$$\begin{aligned} [\![\mathsf{interrupt}\ s\ e]\!]\ k\ h &\triangleq [\![e]\!]\ (\lambda x.\, (\mathsf{lookup}\ h\ \ulcorner s \urcorner)\ x\ k)\ h \\ [\![\mathsf{run}\ e_1\ \mathsf{handle}\ s\ (\lambda x.\, e_2)]\!]\ k\ h &\triangleq [\![e_1]\!]\ k\ (\mathsf{update}\ h\ \ulcorner s \urcorner\ (\lambda x k'.\, [\![e_2]\!]\ k'\ h)) \end{aligned}$$

The main difference between termination semantics and resumption semantics is that with the former, the continuation to be invoked when the handler is finished is the continuation at site of the handler definition, whereas with the latter, it is the continuation at the site of the interrupt.