1 Exceptions

An exception mechanism allows non-local transfer of control in exceptional situations. It is typically used to handle abnormal, unexpected, or rarely occurring events. It can simplify code by allowing programmers to factor out these uncommon cases. OCaml also uses them for not-found conditions when searching lists and similar data structures, a questionable design decision; Standard ML uses option for this purpose.

To add an exception handling mechanism to FL, we first extend the syntax:

$$e \ ::= \ \ldots \mid \mathsf{raise}\, s\, e \mid \mathsf{try}\, e_1\, \mathsf{handle}\, s\, e_2$$

Informally, the idea is that handle provides a handler e_2 to be invoked when the exception named s is encountered inside the expression e_1. To raise an exception, the program calls $\mathsf{raise}\, s\, e$, where s is the name of an exception and e is an expression that will be passed as an argument to the handler.

Most languages use a dynamic scoping mechanism to find the handler for a given exception. When an exception is encountered, the language walks up the runtime call stack until a suitable exception handler is found.

1.1 Exceptions in FL

To add support for exceptions to our CPS translation, we add a *handler environment* h, which maps exception names to continuations. We also extend our lookup and update functions to accommodate handler environments. Applied to a handler environment, lookup returns the continuation bound to a given exception name, and update rebinds an exception name to a new continuation.

Now we can add support for exceptions to our translation:

$$\begin{aligned}
\mathcal{E}[\![\mathsf{raise}\, s\, e]\!]\rho k h &\triangleq \mathcal{E}[\![e]\!]\rho(\mathsf{lookup}\ h\ \ulcorner s\urcorner)h \\
\mathcal{E}[\![\mathsf{try}\, e_1\, \mathsf{handle}\, s\, e_2]\!]\rho k h &\triangleq \mathcal{E}[\![e_2]\!]\rho(\lambda f.\, \mathcal{E}[\![e_1]\!]\rho k(\mathsf{update}\ h\ (\lambda v.\, f v k h)\ \ulcorner s\urcorner))h \\
\mathcal{E}[\![\lambda x.\, e]\!]\rho k h &\triangleq k\,(\lambda v.\, \mathcal{E}[\![e]\!](\mathsf{update}\ \rho\ v\ \ulcorner x\urcorner)) \\
&=_\eta \ k\,(\lambda v k' h'.\, \mathcal{E}[\![e]\!](\mathsf{update}\ \rho\ v\ \ulcorner x\urcorner)\, k'\, h') \\
\mathcal{E}[\![e_0\, e_1]\!]\rho k h &\triangleq \mathcal{E}[\![e_0]\!]\rho(\lambda f.\, \mathcal{E}[\![e_1]\!]\rho(\lambda v.\, f v k h)h)h
\end{aligned}$$

The translation of $\mathsf{raise}\, s\, e$ simply evaluates e, then passes this as an argument to the handler bound to s in the handler environment. The translation of $\mathsf{try}\, e_1\, \mathsf{handle}\, s\, e_2$ evaluates the handler e_2 first, producing (the translation of) a function $\lambda x.\, e'$, then makes a continuation $(\lambda v.\, f v k h)$ out of it using the continuation k and handler environment h of the try statement, then binds this to s and evaluates e_1 in this new handler environment.

There are some subtle design decisions captured by this translation. Note that in the translation of $\mathsf{try}\, e_1\, \mathsf{handle}\, s\, e_2$, s is in scope in e_1 but not in e_2. Thus if e_2 attempts to raise the exception s, e_2 will not be invoked again. That is, e_2 cannot be invoked recursively. Another thing to notice is that in the translation of $\mathsf{raise}\, s\, e$, the continuation k disappears completely.

With runtime type checking as described in Lecture **??**, our translation becomes

$$\begin{aligned}
\mathcal{E}[\![\mathsf{raise}\, s\, e]\!]\rho k h &\triangleq \mathcal{E}[\![e]\!]\rho(\mathsf{lookup}\ h\ \ulcorner s\urcorner)\, h \\
\mathcal{E}[\![\mathsf{try}\, e_1\, \mathsf{handle}\, s\, e_2]\!]\rho k h &\triangleq \mathcal{E}[\![e_2]\!]\rho(\mathsf{check\text{-}fun}\,(\lambda f.\, \mathcal{E}[\![e_1]\!]\rho k(\mathsf{update}\ h\ (\lambda v.\, f v k h)\ \ulcorner s\urcorner)))\, h \\
\mathcal{E}[\![\lambda x.\, e]\!]\rho k h &\triangleq k\,(\mathsf{tag\text{-}fun}\,(\lambda v.\, \mathcal{E}[\![e]\!](\mathsf{update}\ \rho\ v\ \ulcorner x\urcorner))) \\
&= k\,(\mathsf{tag\text{-}fun}\,(\lambda v k' h'.\, \mathcal{E}[\![e]\!](\mathsf{update}\ \rho\ v\ \ulcorner x\urcorner) k' h')) \\
\mathcal{E}[\![e_0\, e_1]\!]\rho k h &\triangleq \mathcal{E}[\![e_0]\!]\rho(\mathsf{check\text{-}fun}\,(\lambda f.\, \mathcal{E}[\![e_1]\!]\rho(\lambda v.\, f v k h) h))\, h
\end{aligned}$$

where tag-fun tags a function with its runtime type and check-fun modifies the continuation to check the tag of its argument, then strip off the tag and apply the original continuation to the untagged value (see Lecture **??**, §**??**.

1.2 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. The Cedar/Mesa system supported both kinds of exceptions and found that resumption-style exceptions were almost never used, and often resulted in bugs when they were.

Operating system interrupts are one place where resumption semantics can be seen. When a process receives an interrupt, the interrupt handler is run, and then execution continues at the point in the program where the interrupt happened.

We can give a translation that captures the semantics of resumption-style exceptions. We add two constructs to FL:

$$e \ ::= \ \mathsf{interrupt}\, s\, e \mid \mathsf{try}\, e_1\, \mathsf{handle}\, s\, e_2$$

The translation makes the handler environment h a mapping from exception names to *functions* rather than to continuations:

$$\begin{aligned}
[\![\mathsf{interrupt}\, s\, e]\!]\rho k h &= [\![e]\!]\rho(\lambda v.\, (\mathsf{lookup}\ h\ \ulcorner s\urcorner)\, v\, k) h \\
[\![\mathsf{try}\, e_1\, \mathsf{handle}\, s\, e_2]\!]\rho k h &= [\![e_2]\!]\rho(\lambda f.\, ([\![e_1]\!]\rho k(\mathsf{update}\ h\ (\lambda v k'.\, f v k' h)\ \ulcorner s\urcorner)))\, h
\end{aligned}$$

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.

2 First-Class Continuations

Some languages expose continuations as first-class values. Examples of such languages include Scheme and SML/NJ. In the latter, there is a module that defines a continuation type α cont representing a continuation expecting a value of type α. There are two functions for manipulating continuations:

$\mathsf{callcc} : (\alpha\,\mathsf{cont} \to \alpha) \to \alpha$	$(\mathsf{callcc}\ f)$ passes the current continuation to the function f
$\mathsf{throw} : \alpha\,\mathsf{cont} \to \alpha \to \beta$	$(\mathsf{throw}\ k\ v)$ sends the value v to the continuation k.

The call $(\mathsf{callcc}\, f)$ passes the current continuation corresponding to the evaluation context of the callcc itself to the function f of type $\alpha\,\mathsf{cont} \to \alpha$. The current continuation k is of type $\alpha\,\mathsf{cont}$. When called with this

continuation, f may evaluate to a value of type α, and that is the value of the expression ($\mathsf{callcc}\, f$) that called it. However, the continuation k passed to f may be called with a value v of type α by ($\mathsf{throw}\, k\, v$) with the same effect. It is up to the evaluation context of the callcc to determine which. Thus ($\mathsf{callcc}\ \lambda k.\, 3$) and ($\mathsf{callcc}\ \lambda k.\, \mathsf{throw}\, k\, 3$) have the same effect.

2.1 Semantics of First-Class Continuations

Using the translation approach we introduced earlier, we can easily describe these mechanisms. Suppose we represent a continuation value for the continuation k by tagging it with the integer 7. Then we can translate callcc and throw as follows:

$$\begin{aligned} [\![\mathsf{callcc}\, e]\!]\rho k &= [\![e]\!]\rho(\mathsf{check\text{-}fun}\, (\lambda f.\, f\, (7, k)\, k)) \\ [\![\mathsf{throw}\, e_1\, e_2]\!]\rho k &= [\![e_1]\!]\rho(\mathsf{check\text{-}cont}\, (\lambda k'.\, [\![e_2]\!]\rho k')) \end{aligned}$$

The key to the added power is the non-linear use of k in the callcc rule. This allows k to be reused any number of times.

2.2 Implementing Threads with Continuations

Once we have first-class continuations, we can use them to implement all the different control structures we might want. We can even use them to implement (non-preemptive) threads, as in the following code that explains how concurrency is handled in languages like OCaml and Concurrent ML:

```
type thread = unit cont

let ready : thread queue = new_queue (* a mutable FIFO queue *)
let enqueue t = insert ready t
let dispatch() = throw (dequeue ready) ()

let spawn (f : unit -> unit) : unit =
   callcc (fun k -> (enqueue k; f(); dispatch()))
let yield() : unit = callcc (fun k -> enqueue k; dispatch())
```

The interface to threads consists of the functions `spawn` and `yield`. The `spawn` function expects a function f containing the work to be done in the newly spawned thread. The `yield` function causes the current thread to relinquish control to the next thread on the ready queue. Control also transfers to a new thread when one thread finishes evaluating. To complete the implementation of this thread package, we just need a queue implementation. Concurrent ML has preemptive threads, in which threads implicitly yield automatically after a certain amount of time; this requires just a little help from the operating system.