

## 1 CPS and Strong Typing

Now let us use CPS semantics to augment our previously defined FL language translation so that it supports runtime type checking. This time our translated expressions will be functions of  $\rho$  and  $k$  denoting an environment and a continuation, respectively. The term  $\mathcal{E}\llbracket e \rrbracket \rho k$  represents a program that evaluates  $e$  in the environment  $\rho$  and sends the resulting value to the continuation  $k$ .

As before, assume that we have an encoding of variable names  $x$  and a representation of environments  $\rho$  along with lookup and update functions  $\text{lookup } \rho x$  and  $\text{update } \rho x v$ .

In addition, we want to catch type errors that may occur during evaluation. As before, we use integer tags to keep track of types:

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

A *tagged value* is a value paired with its type tag; for example,  $(0, \text{true})$ . Using these tagged values, we can now define a translation that incorporates runtime type checking:

$$\begin{aligned} \mathcal{E}\llbracket x \rrbracket \rho k &\triangleq k(\text{lookup } \rho x) \\ \mathcal{E}\llbracket b \rrbracket \rho k &\triangleq k(\text{Bool}, b) \\ \mathcal{E}\llbracket n \rrbracket \rho k &\triangleq k(\text{Num}, n) \\ \mathcal{E}\llbracket \text{nil} \rrbracket \rho k &\triangleq k(\text{Null}, 0) \\ \mathcal{E}\llbracket (e_1, \dots, e_n) \rrbracket \rho k &\triangleq \mathcal{E}\llbracket e_1 \rrbracket \rho(\lambda v_1. \dots \mathcal{E}\llbracket e_2 \rrbracket \rho(\lambda v_2. \dots \mathcal{E}\llbracket e_n \rrbracket \rho(\lambda v_n. k(\text{Tuple}, n, (v_1, \dots, v_n)))))) \\ \mathcal{E}\llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket \rho k &\triangleq \mathcal{E}\llbracket e_1 \rrbracket \rho(\lambda p. \mathcal{E}\llbracket e_2 \rrbracket (\text{update } \rho x p) k) \\ \mathcal{E}\llbracket \lambda x. e \rrbracket \rho k &\triangleq k(\text{Func}, \lambda x k. \mathcal{E}\llbracket e \rrbracket (\text{update } \rho x x) k) \\ \mathcal{E}\llbracket \text{error} \rrbracket \rho k &\triangleq k(\text{Err}, \text{error}). \end{aligned}$$

Now a function application can check that it is actually applying a function:

$$\mathcal{E}\llbracket e_0 e_1 \rrbracket \rho k \triangleq \mathcal{E}\llbracket e_0 \rrbracket \rho(\lambda p. \text{let } (t, f) = p \text{ in if } t \neq \text{Func} \text{ then error else } \mathcal{E}\llbracket e_1 \rrbracket \rho(\lambda v. f v k))$$

We can simplify this by defining a helper function `check-fn`:

$$\text{check-fn} \triangleq \lambda k p. \text{let } (t, f) = p \text{ in if } t \neq \text{Func} \text{ then error else } k f.$$

The helper function takes in a continuation and a tagged value, checks the type, strips off the tag, and passes the raw (untagged) value to the continuation. Then

$$\mathcal{E}\llbracket e_0 e_1 \rrbracket \rho k \triangleq \mathcal{E}\llbracket e_0 \rrbracket \rho(\text{check-fn } (\lambda f. \mathcal{E}\llbracket e_1 \rrbracket \rho(\lambda v. f v k))).$$

Similarly,

$$\begin{aligned} \mathcal{E}\llbracket \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \rrbracket \rho k &\triangleq \mathcal{E}\llbracket e_0 \rrbracket \rho(\text{check-bool } (\lambda b. \text{if } b \text{ then } \mathcal{E}\llbracket e_1 \rrbracket \rho k \text{ else } \mathcal{E}\llbracket e_2 \rrbracket \rho k)) \\ \mathcal{E}\llbracket \#n e \rrbracket \rho k &\triangleq \mathcal{E}\llbracket e \rrbracket \rho(\text{check-pair } (\lambda t. k(\#n t))). \end{aligned}$$

## 2 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:

$$e ::= \dots \mid \text{raise } s e \mid \text{try } e_1 \text{ handle } (s x) e_2$$

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 `raise s e`, where  $s$  is the name of an exception and  $e$  is an expression that will be passed to the handler as its argument  $x$ .

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.

### 2.1 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 exception support to our translation:

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

where `tag-fun` tags a function value with its runtime type.

There are some subtle design decisions captured by this translation. For example, if  $e_2$  raises exception  $s$  in `try  $e_1$  handle  $(s x) e_2$` , in this translation  $e_2$  will not be invoked again. That is,  $e_2$  cannot be invoked recursively.

### 2.2 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:

$$e ::= \text{interrupt } s e \mid \text{try } e_1 \text{ handle } (s x) e_2$$

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

$$\begin{aligned} \llbracket \text{interrupt } s e \rrbracket \rho k h &= \llbracket e \rrbracket \rho (\lambda v. (\text{lookup } h s) v k) h \\ \llbracket \text{try } e_1 \text{ handle } (s x) e_2 \rrbracket \rho k h &= \llbracket e_1 \rrbracket \rho k (\text{update } h s (\lambda v k'. \llbracket e_2 \rrbracket \rho k' h)) \end{aligned}$$

This translation shows that with resumption semantics, the exception handler is really a dynamically bound function that is invoked at the point where the exception happens.

### 3 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  $\alpha \text{ cont}$  representing a continuation expecting a value of type  $\alpha$ . There are two functions for manipulating continuations:

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

The call ( $\text{callcc } f$ ) passes the current continuation corresponding to the evaluation context of the  $\text{callcc}$  itself to the function  $f$  of type  $\alpha \text{ cont} \rightarrow \alpha$ . The current continuation  $k$  is of type  $\alpha \text{ cont}$ . When called with this continuation,  $f$  may evaluate to a value of type  $\alpha$ , and that is the value of the expression ( $\text{callcc } f$ ) that called it. However, the continuation  $k$  passed to  $f$  may be called with a value  $v$  of type  $\alpha$  by ( $\text{throw } k v$ ) with the same effect. It is up to the evaluation context of the  $\text{callcc}$  to determine which. Thus ( $\text{callcc } \lambda k. 3$ ) and ( $\text{callcc } \lambda k. \text{throw } k 3$ ) have the same effect.

#### 3.1 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  $\text{callcc}$  and  $\text{throw}$  as follows:

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

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

#### 3.2 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. CML 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.