

1 Introduction

In this lecture we discuss:

- State
- `setjmp`, `longjmp`, `callcc`, and `throw`
- Exceptions

2 State

In the previous lecture we introduced Continuation Passing Style (*CPS*). For the translation of an expression we used the following notation:

$$\llbracket e \rrbracket \rho k$$

Semantically, this means that we send the result of the evaluation of e in environment ρ to the continuation k , where e is an expression, ρ is the naming context, and k is the control context.

In order to have references with *CPS*, we need to add state, σ , which will take a location and return a value. Intuitively, state is threaded throughout computation and must be passed to the continuation, so we model the continuation as a function taking both value and state:

$$k = \lambda v. \lambda \sigma. \dots$$

Translation is denoted as:

$$\llbracket e \rrbracket \rho k \sigma$$

We have the following functions, which we introduced when we first discussed references:

$$\text{update} : (\sigma, l, v) \rightarrow \sigma'$$

$$\text{lookup} : (\sigma, l) \rightarrow v$$

$$\text{malloc} : \sigma \rightarrow (l, \sigma')$$

The new translations are:

$$\llbracket \text{ref } e \rrbracket \rho k = \llbracket e \rrbracket \rho (\lambda v. \lambda \sigma. \text{let } (l, \sigma') = \text{malloc}(\sigma) \text{ in } k(\text{LOC } l) (\text{update } \sigma' l v))$$

where $\text{LOC} = \lambda l. (4, l)$ serves to *tag* locations, in the same way as the *bool*, *integer*, *tuple*, and *function* tags we saw previously. We will use corresponding functions *BOOL*, *INT*, etc., which in conjunction with the tag-checking functions *check-bool*, *check-int* effectively abstract over whether we are doing tagging and checking.

$$\llbracket ! e \rrbracket \rho k = \llbracket e \rrbracket \rho (\text{check-loc}(\lambda l. \lambda \sigma'. k(\text{lookup } \sigma' l) \sigma'))$$

$$\llbracket e_1 := e_2 \rrbracket \rho k = \llbracket e_1 \rrbracket \rho (\text{check-loc}(\lambda l. \lambda \sigma. \llbracket e_2 \rrbracket \rho (\lambda v. \lambda \sigma'. k(\text{UNIT } 0) (\text{update } \sigma' l v)) \sigma))$$

It is important to note that, though we have introduced state, we do not need new translation rules for the expressions we considered before because of η -equivalence. For example:

$$\llbracket x \rrbracket \rho k = \lambda \sigma. k(\rho x) \sigma =_{\eta} k(\rho x)$$

3 setjmp and longjmp

`setjmp` is a C function which takes a pointer to a buffer. Its operation is to save the state of all registers (including the program counter) into the specified buffer and return 0. `longjmp`, also a C function, takes as an argument a pointer to a buffer (which is presumed to be a buffer which has already been filled with a previous call to `setjmp`) and a value. When invoked, it restores all of the registers to the values saved in the buffer and returns the value passed in to the point in the program where `setjmp` was called (in effect, the program resumes executing right where `setjmp` was called, except the call will return the value passed in to `longjmp`). These functions can be used for error handling. `setjmp` is called before code that may result in an error. If an error occurs in computation, `longjmp` is called in order to restore initial state and handle the error. For instance:

```
if (setjmp(&jmpbuf))
    // error handling code goes here
else
    do_computation();
    // if error occurs in compute(), call
    // longjmp(jmpbuf, e), where e is the error code
```

These functions can be translated using continuations into:

$$\begin{aligned} \llbracket \text{setjmp } e \rrbracket \rho k &= \llbracket e \rrbracket \rho (\text{check-loc } (\lambda l. \lambda \sigma. k (\text{INT } 0) (\text{update } \sigma \ l \ (CONT k)))) \\ \llbracket \text{longjmp } e \ e' \rrbracket \rho k &= \llbracket e \rrbracket \rho (\text{check-loc } (\lambda l. \llbracket e \rrbracket \rho (\text{check-loc } (\lambda l. \llbracket e' \rrbracket \rho \\ &\quad (\lambda v. \lambda \sigma. \text{check-cont } (\text{lookup } \sigma \ l) (\lambda k'. k' \ v \ \sigma)))))) \end{aligned}$$

The translation of `setjmp` stores a continuation at a new location, while the translation of `longjmp` restores a continuation from a program location. This is roughly equivalent to restoring the registers and program state of the executing program.

4 Continuations in some programming languages

4.1 SML/NJ

The basis library of SML/NJ includes several functions designed to work with continuations. These functions are stored in the module `SMLofNJ.Cont` and include:

$$\begin{aligned} \text{throw} &: \alpha \text{ cont} \rightarrow \alpha \rightarrow \beta \\ \text{callcc} &: (\alpha \text{ cont} \rightarrow \alpha) \rightarrow \alpha \end{aligned}$$

The expression `throw c v` passes `v` to the continuation `c`, while `callcc f` invokes `f` and passes it the current continuation. The semantics for these two expressions are as follows:

$$\llbracket \text{callcc } e \rrbracket \rho k = \llbracket e \rrbracket \rho (\text{check-fun } (\lambda f. f (CONT k) k))$$

$$\llbracket \text{throw } e_c \ e_v \rrbracket \rho k = \llbracket e_c \rrbracket \rho (\text{check-cont } (\lambda k'. \llbracket e_v \rrbracket \rho (\lambda v. k' \ v)))$$

Note: `k` is not needed in the second translation because `throw` doesn't actually return control to its context. This also explains why the type definition of `throw` specifies that the call returns an arbitrary type β , since it doesn't actually return a value at all

An example of using `callcc` and `throw` for exception handling is below:

```

open SMLofNJ.Cont;

fun processList' itemList eHandler let
  val listlen = length itemList
in
  (* If the list is of zero length, we can't process it.
   * Throw an exception to our error handler *)
  if listlen = 0 then
    throw eHandler 42
  else
    (* Do some processing on the list here *)
  end

fun processList(itemList) if (callcc (processList' itemList)) = 42
  then (* Try to handle the error somewhat gracefully *)
  else (* Success! Continue as normal *)

```

4.2 Scheme

The language Scheme also includes support for explicit continuations. Using `callcc`, the current continuation is turned into a function that can be called directly, but when called it will never return (as expected, given that it is equivalent to restoring the continuation)

4.3 Threads

Concurrent ML (CML) implements threads as continuations. Each thread has access to a scheduler continuation, which when invoked switched to another scheduled continuation.

Aside: this speaks to the difficulty of implementing continuations, since threading can be implemented using continuations and it is nontrivial to implement threading correctly.

5 Exceptions

Exceptions are another modern language feature which can be explained (and implemented) using continuations. Exceptions are useful for error handling within applications, and for conveniently returning results for unusual cases without cluttering up the “main-line” code.

5.1 Termination semantics

Most languages supporting exceptions support termination-style exceptions, in which an exception terminates the computation in which it occurs. Usually there are constructs similar to the following:

- **raise** $s\ e$ — throws/raises/signals an exception s with value e
- **try** e_1 **catch** $(s\ x)$ e_2 — Attempt to execute e_1 . If an exception s is raised with value x , then execute e_2 .

Handling an exception is similar to `setjmp` and `longjmp` from above, except that we need to be able to look up the continuation (if any) that can handle a raised exception. To accomplish this, we create a new handling environment $h : \text{exceptionName} \rightarrow \text{continuation}$ and pass that to our translation functions along with ρ and k . One other notable distinction is that the handling environment applies where a given function is used, not where it is lexically defined. These considerations lead to the following semantic translations:

$$\begin{aligned}
\llbracket \text{raise } s \ e \rrbracket \rho k h &= \llbracket e \rrbracket \rho (\lambda v. (\text{lookup-handler } h \ s) v) h \\
\llbracket \text{try } e_1 \ \text{catch } (s \ x) \ e_2 \rrbracket \rho k h &= \llbracket e_1 \rrbracket \rho k (\text{extend-handler } h \ s \ (\lambda v. \llbracket e_2 \rrbracket (\text{extend } \rho \ x \ v) \ k \ h)) \\
\llbracket \lambda x. e \rrbracket \rho k h &= k (\lambda v. \lambda k'. \lambda h'. \llbracket e \rrbracket (\text{extend } \rho \ x \ v) k' h') \\
\llbracket e_1 \ e_2 \rrbracket \rho k h &= \llbracket e_1 \rrbracket \rho (\text{check-fun}(\lambda f. \llbracket e_2 \rrbracket \rho (\lambda v. f \ v \ k \ h)))
\end{aligned}$$

The semantics for exceptions is very similar to the semantics for dynamic scoping, since in both cases the meanings of objects in the language (variables in the case of dynamic scoping, and handlers in the case of exceptions) depend on where they are used in the run-time environment and not just in the lexical environment. In particular, the translation of a function expression is a function that explicitly takes a handler as a dynamic argument. In fact, if a language has dynamic scoping and first-class continuations, an exception handling mechanism can be encoded by simply storing continuations into variables that record the location of the appropriate exception handler.

We can extend this translation to explain other exception mechanisms such as `try...finally`, an exercise left to the reader.

5.2 Resumption semantics

An alternate model is exceptions that permit the computation in which the exception occurred to be resumed after the exception is handled. For example, operating system interrupts have roughly this behavior. To allow resumption, the handler must receive a continuation to be resumed. In fact, the handler becomes a essentially a function that is called from the point where the exception is raised:

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

This semantics exposes some interesting semantic issues. For example, in the `interrupt` (“trap”) handler, the code e_2 uses the handler environment h' explicitly passed up from the `interrupt`, rather than the lexical handler environment h . This means that exceptions raised in the handler will be sent back down into code that raised the exception.