In this lecture we discuss advanced control constructs, formalizing exceptions and continuations in the C-machine. We begin with exceptions by introducing a new machine state

\[ k \leftarrow \text{raise } v \]

which signals that we are propagation an exception upwards in the control stack \( k \), looking for a handler or stopping at the empty stack. This “uncaught exception” is a particularly common form of implementing runtime errors. The value \( v \) has some type \( \tau_{\text{exn}} \), and carries some information about what caused the exception. For more other variations of exceptions, see [Ch. 13] and Assignment 4.

We have two new forms of expressions \( \text{raise}(\tau, e) \) (with concrete syntax \( \text{raise}[\tau] e \))\(^1\) and \( \text{try}(e_1, x.e_2) \) (with concrete syntax \( \text{try } e_1 \text{ catch } x = e_2 \)). Informally, \( \text{try}(e_1, x.e_2) \) evaluates \( e_1 \) and returns its value. If an exception \( \text{raise } v \) is raised during the evaluation of \( e_1 \), then we evaluate \( \{v/x\}e_2 \) instead and returns its value (or propagate its exception). These rules are formalized in the C-machine as follows.

---

\(^1\)The type is written here in order to preserve the property that every well-typed expression has a unique type.
In order to verify that these rules are sensible, we should prove appropriate progress and preservation theorems. In order to do this, we need to introduce some typing judgments for machine states and the new forms of expressions. First, expressions:

\[
\begin{align*}
\Gamma \vdash e : \tau_{\text{exn}} & \quad \Gamma \vdash \text{raise}(\tau, e) : \tau \\
\Gamma, x : \tau_{\text{exn}} \vdash e_2 : \tau & \quad \Gamma \vdash \text{try}(e_1, x.e_2) : \tau
\end{align*}
\]

In these rules, the exception value must be of some specified type \(\tau_{\text{exn}}\).

The rule for typing \text{try}, therefore, checks that \(e_2\) has the same type as \(e_1\) assuming that \(x\) has the exception type.

We have several choices for the exception type. \(\tau_{\text{exn}}\) could be \text{int}, in which case the exception value acts like a error code returned by a C library function. If we make \(\tau_{\text{exn}}\) a string type, then an error message can be included in the exception. The advantage of binding the exception value to a variable \(x\) is that the exception handler can react to the exception in a more flexible way than would be possible without this information.

The ML language allows users to declare their own exception types which can carry exception-specific information. A catch clause can choose to handle all exceptions, or only a particular exception type. Java takes this a step further by organizing exceptions into a hierarchy; a clause that catches exception class \(E\) will also catch any of the subclasses of \(E\). In Assignment 4, you will implement an exception mechanism that allows the user to catch exceptions of a particular user-declared type.

As with environments, typing exceptions will depend on specifying typing rules for C-machine states. We can add a judgement for the exception raising state to our rules from last lecture:

\[
\begin{align*}
k : \tau \text{ stack} & \quad \Gamma \vdash \text{raise}(\tau, e) : \tau \\
k \ll \text{raise} v & \quad \text{OK}
\end{align*}
\]

We can now state (without proof) the preservation and progress properties. The proofs follow previous patterns (see [Ch. 13]) and Lecture 5 on Type Safety.
1. (Preservation) If $s$ OK and $s \rightarrow s'$ then $s'$ OK.

2. (Progress) If $s$ OK then either
   
   (i) $s \rightarrow s'$ for some $s'$, or
   
   (ii) $s = \bullet < v$ with $v$ value, or
   
   (iii) $s = \bullet \ll raise v$.

The manner in which the C-machine operates with respect to exceptions may seem a bit unrealistic, since the stack is unwound frame by frame. However, in languages like Java this is not an unusual implementation method. In ML, there is more frequently a second stack containing only handlers for exceptions. The handler at the top of the stack is innermost and a `raise` expression can jump to it directly.

Overall, this machine should be equivalent to the specification of exceptions above, but potentially more efficient. Often, we want to describe several aspects of execution behavior of a language constructs in several different machines, keeping the first as high-level as possible.

In our simple language, the handler stack $h$ contains only frames `catch(k, x.e_2)` while the control stack contains the usual frames, and `try(□)` (the “catch” clause has moved to the handler stack). All the usual rules are augmented to carry a control stack and a handler stack, and leave the handler unchanged.

$$
(h, k) > apply(e_1, e_2) \quad \Rightarrow_c (h, k \triangleright apply(\square, e_2)) > e_1
$$

$$
\cdots
$$

$$
(h, k) > try(e_1, x.e_2) \quad \Rightarrow_c (h \triangleright try(\square)) > e_1
$$

$$
(h \triangleright catch(k', x.e_2), k \triangleright try(\square)) < v \quad \Rightarrow_c (h, k) < v_1
$$

$$
(h, k) > raise(\tau, e) \quad \Rightarrow_c (h, k \triangleright raise(\tau, \square)) > e
$$

$$
(h, k \triangleright raise(\tau, \square)) < v \quad \Rightarrow_c (h, k) \ll raise v
$$

$$
(h \triangleright catch(k', x.e_2), k) \ll raise v \quad \Rightarrow_c (h, k') > \{v/x\} e_2
$$

Note that we do not unwind the control stack explicitly, but jump directly to the handler when an exception is raised. This handler must store a copy of the control stack in effect at the time the `try` expression was executed. Fortunately, this can be implemented without the apparent copying of the stack in the rule for `try`, because we can just keep a pointer to the right frame in the control stack [Ch. 13].

Note also in case of a regular return for the subject of a `try` expression, we need to pop the corresponding handler off the handler stack.
Exception Scope  In ML, exceptions are declared explicitly in the source

text, and can be referred to anywhere in the scope of the declaration. For

example, one can declare an exception with the syntax:

```ml
exception E1
```

We can raise an exception using the syntax:

```ml
raise E1
```

Finally, we can handle exceptions using pattern matching, either using

a particular exception type in the match or a wildcard:

```ml
e1 handle E => e1
    | _ => e2
    ...
```

Consider what happens in the following code:

```ml
fun f x =
  let exception E in
   raise E;
   x+1
  end
(f 5) handle E => 0
```

At the point where the exception handler occurs, E is out of scope, so E
acts as a wildcard. The program will catch the exception and return 0. In
fact, there is no way to catch just exception E when E is out of scope–either
you catch all exceptions or you can’t catch E.

Having exceptions propagate outside the scope where they are declared
is misguided at best–it means that users of a library may be confronted with
internal exception types that are not declared in the library’s public inter-
face, for example. One solution to this problem is to require the interface
of a function to declare the exceptions it may throw, and use a type sys-

tem to guarantee that no “invisible” exceptions are thrown. This solution,
used by Java, has the advantage that the type of a function documents what
happens in the error case as well as in the normal case. The disadvantage
is that signatures become larger and more unwieldy. However, it is easy
to infer exception declarations, so that the programmer only has to specify
exceptions at module boundaries.

SUPPLEMENTARY NOTES  SEPTEMBER 25, 2003
Continuations

In this section we introduce continuations, an advanced control construct available in some functional languages [Ch. 12]. Most notably, they are part of the definition of Scheme and are implemented as a library in Standard ML of New Jersey, even though they are not part of the definition of Standard ML. Continuations have been described as the goto of functional languages, since they allow non-local transfer of control. While they are powerful, programs that exploit continuations can difficult to reason about and their gratuitous use should therefore be avoided.

There are two basic constructs, given here with concrete and abstract syntax. We ignore issues of type-checking in the concrete syntax.

\[ \text{letcc } x \text{ in } e \text{ end} \]
\[ \text{letcc} (\tau, x.e) \]
\[ \text{throw } e_1 \text{ to } e_2 \]
\[ \text{throw} (\tau, e_1, e_2) \]

In brief, letcc \text{ in } e \text{ end} captures the stack (= continuation) \( k \) in effect at the time the letcc is executed and substitutes \( \text{cont}(k) \) for \( x \) in \( e \). We can later transfer control to \( k \) by throwing a value \( v \) to \( k \) with \( \text{throw} v \text{ to } \text{cont}(k) \). Note that the stack \( k \) we capture can be returned passed point in which it was in effect. As a result, \( \text{throw} \) can effect a kind of “time travel”. While this can lead to programs that are very difficult to understand, it has multiple legitimate uses. One pattern of usage is an alternative to exceptions, another is to implement co-routines or thread. Another use is to affect backtracking.

As a starting example we consider simple arithmetic expressions.

(a) \( 1 + \text{letcc } x \text{ in } 2 + (\text{throw } 3 \text{ to } x) \text{ end} \Rightarrow^* 4 \)
(b) \( 1 + \text{letcc } x \text{ in } 2 \text{ end} \Rightarrow^* 3 \)
(c) \( 1 + \text{letcc } x \text{ in } \text{if} (\text{throw } 2 \text{ to } x) \text{ then } 3 \text{ else } 4 \text{ fi} \text{ end} \Rightarrow^* 3 \)

Example (a) shows an upward use of continuations similar to exceptions, where the addition of \( 2 + \square \) is bypassed and discarded when we throw to \( x \).

Example (b) illustrates that captured continuations need not be used in which case the normal control flow remains in effect.

Example (c) demonstrates that a throw expression can occur anywhere; its type does not need to be tied to the type of the surrounding expression. This is because a throw expression never returns normally—it always passes control to its continuation argument.
With this intuition we can describe the operational semantics, followed by the typing rules.

\[ k > \text{letcc}(\tau, x.e) \mapsto_c k > \{\text{cont}(k) / x\}e \]

\[ k > \text{throw}(\tau, e_1, e_2) \mapsto_c k \triangleright \text{throw}(\tau, \Box, e_2) > e_1 \]

\[ k \triangleright \text{throw}(\tau, v_1, \Box) < \text{cont}(k_2) \mapsto_c k_2 < v_1 \]

\[ k > \text{cont}(k') \mapsto_c k < \text{cont}(k') \]

The typing rules can be derived from the need to make sure both preservation and progress to hold. First, the constructs that can appear in the source.

\[
\frac{\Gamma, x: \tau \vdash e: \tau}{\Gamma \vdash \text{letcc}(\tau, x.e): \tau}
\]

\[
\frac{\Gamma \vdash e_1: \tau_1 \quad \Gamma \vdash e_2: \tau_1 \text{ cont}}{\Gamma \vdash \text{throw}(\tau, e_1, e_2): \tau}
\]

Finally, the rules for continuation values that can only arise during computation. They are needed to check the machine state, even though they are not needed to type-check the input.

\[
\frac{k: \tau \text{ stack}}{\Gamma \vdash \text{cont}(k): \tau \text{ cont}}
\]

As a more advanced example, consider the problem of composing a function with a continuation. This can also be viewed as explicitly pushing a frame onto a stack, represented by a continuation. Even though we have not yet discussed polymorphism, we will phrase it as a generic problem:

Write a function

\[
\text{compose} : (\tau \to \sigma) \to \sigma \text{ cont} \to \tau \text{ cont}
\]

so that \text{compose} \( F \) \( K \) returns a continuation \( K_1 \). Throwing a value \( v \) to \( K_1 \) should first compute \( F v \) and then throw the resulting value \( v' \) to \( K \).

To understand the solution, we analyze the intended behavior of \( K_1 \). When given a value \( v \), it first applies \( F \) to \( v \). So

\[
K_1 = K_2 \triangleright \text{apply}(F, \Box)
\]
for some $K_2$. Then, it needs to throw the result to $K$. So

$$K_2 = K_3 \triangleright \text{throw}(., \square, K)$$

and therefore

$$K_1 = K_3 \triangleright \text{throw}(., \square, K) \triangleright \text{apply}(F, \square)$$

for some $K_3$.
How can we create such a continuation? The expression

$$\text{throw } (F \ldots) \text{ to } K$$

will create a continuation of the form above. This continuation will be the stack precisely when the hole "\ldots" is reached. So we need to capture it there:

$$\text{throw } (F \text{(letcc } k_1 \text{ in } \ldots \text{ end})) \text{ to } K$$

The next conundrum is how to return $k_1$ as the result of the compose function, now that we have captured it. Certainly, we can not just replace $\ldots$ by $k_1$, because the $F$ would be applied (which is not only wrong, but also not type-correct). Instead we have to throw $k_1$ out of the local context! In order to throw it to the right place, we have to name the continuation in effect when the compose is called.

$$\text{letcc } r \text{ in } \text{throw } (F \text{ (letcc } k_1 \text{ in } \text{throw } k_1 \text{ to } r \text{ end})) \text{ to } K \text{ end}$$

Now it only remains to abstract over $F$ and $K$, where we take the liberty of writing a curried function directly in our language.

$$\text{fun compose } (f:\text{‘a} \rightarrow \text{‘b}) \text{ (k:\text{‘b cont}) : \text{‘a cont is}}$$

$$\text{letcc } r \text{ in }$$

$$\text{throw } (f \text{ (letcc } k_1 \text{ in } \text{throw } k_1 \text{ to } r \text{ end})) \text{ to } k \text{ end}$$

end
In order to verify the correctness of this function, we can just calculate, using the operational semantics, what happens when \( \text{compose} \) is applied to two values \( F \) and \( K \) under some stack \( K_0 \). This is a very useful exercise, because the correctness of many opaque functions can be verified in this way (and many incorrect functions discovered).

\[
K_0 > \text{apply}(\text{apply}(\text{compose}, F), K)
\]

\[
\Rightarrow^*_c \quad K_0 > \text{letcc}(\_ \cdot r. \text{throw}(\_, \text{apply}(F, \text{letcc}(\_ \cdot k_1. \text{throw}(\_, k_1, r)))), K)
\]

\[
\Rightarrow_c \quad K_0 > \text{throw}(\_, \text{apply}(F, \text{letcc}(\_ \cdot k_1. \text{throw}(\_, k_1, \text{cont}(K_0)))), K)
\]

\[
\Rightarrow_c \quad K_0 \triangleright \text{throw}(\_, \Box, K) > \text{apply}(F, \text{letcc}(\_ \cdot k_1. \text{throw}(\_, k_1, \text{cont}(K_0))))
\]

\[
\Rightarrow^*_c \quad K_0 \triangleright \text{throw}(\_, \Box, K) \triangleright \text{apply}(F, \Box) > \text{letcc}(\_ \cdot k_1. \text{throw}(\_, k_1, \text{cont}(K_0)))
\]

At this point, we define

\[
K_1 = K_0 \triangleright \text{apply}(\_, \Box, K) \triangleright \text{apply}(F, \Box)
\]

and continue

\[
\Rightarrow_c \quad K_1 > \text{throw}(\_, K_1, \text{cont}(K_0))
\]

\[
\Rightarrow_c \quad K_0 < K_1
\]

By looking at \( K_1 \) we can see that it exactly satisfies our specification. Interestingly, \( K_3 \) from our earlier motivation turns out to be \( K_0 \), the continuation in effect at the evaluation of \( \text{compose} \). Note that if \( F \) terminates normally, then that part of the continuation is discarded because \( K \) is installed instead as specified. However, if \( F \) raises an exception, control is returned back to the point where the \( \text{compose} \) was called, rather than to the place where the resulting continuation was invoked (at least in our semantics). This is an example of the rather unpleasant interactions that can take place between exceptions and continuations.

See the code\(^2\) for a rendering of this in Standard ML of New Jersey, where we have slightly different primitives. The translations are as given below. Note that, in particular, the arguments to \( \text{throw} \) are reversed which may be significant in some circumstances because of the left-to-right evaluation order.

\[
\begin{array}{lll}
\text{Concrete MinML} & \text{Abstract MinML} & \text{SML of NJ} \\
\text{letcc} \colon x \in e \text{ end} & \text{letcc}(\tau, x, e) & \text{callcc} \ (\text{fn} \ x \Rightarrow e) \\
\text{throw} e_1 \text{ to } e_2 & \text{throw}(\tau, e_1, e_2) & \text{throw} e_2 \ e_1
\end{array}
\]

For a simpler and quite practical example for the use of continuation refer to the implementation of threads given in the textbook [Ch. 12.3]. A runnable version of this code can be found at the same location as the example above.

\(^2\)http://www.cs.cmu.edu/~fp/courses/312/code/10-continuations/
Lecture 10 Addendum:
Implementation of the C-machine

This material was included in Fall '02. We show how to implement the
C-machine for the fragment containing integers, booleans, and functions
using higher-order functions.

The implementation of the C-machine is to represent the stack as a continuation
that encapsulates the rest of the computation to be performed.\footnote{We give some code excerpts here; the full code can be found at
http://www.cs.cmu.edu/˜fp/courses/312/code/10-exceptions/}

First, in our implementation, both expressions and value have type \texttt{exp}.
We nonetheless use different names to track our intuition, even though the
type system of ML does not help use verify the correctness of this intuition.

\begin{verbatim}
type value = exp
e : exp
v : value
\end{verbatim}

Next, the stack \(k\) is represented by an ML function

\begin{verbatim}
k : value -> value
\end{verbatim}

Applying this function to a value \(v\) will carry out the rest of the compu-
tation of the machine, returning the final answer. Finally, we have two
functions

\begin{verbatim}
eval : exp -> (value -> value) -> value
return : value -> (value -> value) -> value
\end{verbatim}

satisfying the specification:

(i) \(\texttt{eval } e \ k \downarrow a \iff k > e \mapsto c \bullet < a\)

(ii) \(\texttt{return } v \ k \downarrow a \iff k < v \mapsto c \bullet < a\)

In order to implement stacks as ML functions, it is useful to introduce
some new auxiliary functions to represent the frames. We give in the table
below the association between forms of the stack and the corresponding
ML function. We omit only the case for primops which requires a simple
treatment of lists.
The case of the empty stack corresponds to the initial continuation, which simply returns the value passed to it as the result of the overall computation.

Now we can piece together the whole code elegantly, as advertised. We have elided only the case for primitive operations, which can be found with the complete code at the address given above.

fun eval (v as Int _) k = return v k (* elided primops *)
| eval (v as Bool _) k = return v k
| eval (If(e1, e2, e3)) k =
  eval e1 (fn v1 => ifFrame (v1, e2, e3) k)
| eval (v as Fun _) k = return v k
| eval (Apply(e1, e2)) k =
  eval e1 (fn v1 => applyFrame1 (v1, e2) k)
| eval (Let(e1, ((), e2))) k =
  eval e1 (fn v1 => letFrame (v1, ((), e2)) k)
(* eval (Var _) k impossible by MinML typing *)
and ifFrame (Bool(true), e2, e3) k = eval e2 k
| ifFrame (Bool(false), e2, e3) k = eval e3 k (* other expressions impossible by MinML typing *)
and applyFrame1 (v1, e2) k =
  eval e2 (fn v2 => applyFrame2 (v1, v2) k)
and applyFrame2 (v1 as Fun(_, _, ((), ()), e1'), v2) k =
  eval (Subst.subst (v1, 2, Subst.subst (v2, 1, el'))) k
(* other expressions impossible by MinML typing *)
and letFrame (v1, ((), e2)) k = eval (Subst.subst (v1, 1, e2)) k
and return v k = k v

The overall evaluation just starts with the initial continuation which corresponds to the empty stack.

fun evaluate e = eval e (fn v => v)

This style of writing an interpreter is also referred to as continuation-passing style. It is quite flexible and elegant.