Supplementary Notes on Exceptions and Continuations

15-312: Foundations of Programming Languages
Frank Pfenning
Modified by Jonathan Aldrich

Lecture 10
September 25, 2003

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

$$k \ll \texttt{raise}\, v$$

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 τ_{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 $\texttt{raise}(\tau, e)$ (with concrete syntax $\texttt{raise}[\tau]\, e$)[1] and $\texttt{try}(e_1, x.e_2)$ (with concrete syntax $\texttt{try}\, e_1\, \texttt{catch}\, x\ =>\ e_2$). Informally, $\texttt{try}(e_1, x.e_2)$ evaluates e_1 and returns its value. If an exception $\texttt{raise}\, v$ 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.

$$
\begin{array}{lcll}
k > \texttt{try}(e_1, x.e_2) & \mapsto_{\mathsf{c}} & k \triangleright \texttt{try}(\Box, x.e_2) > e_1 & \\
k \triangleright \texttt{try}(\Box, x.e_2) < v_1 & \mapsto_{\mathsf{c}} & k < v_1 & \\
k > \texttt{raise}(\tau, e) & \mapsto_{\mathsf{c}} & k \triangleright \texttt{raise}(\tau, \Box) > e & \\
k \triangleright \texttt{raise}(\tau, \Box) < v & \mapsto_{\mathsf{c}} & k \ll \texttt{raise}\, v & \\
k \triangleright f \ll \texttt{raise}\, v & \mapsto_{\mathsf{c}} & k \ll \texttt{raise}\, v & \text{for } f \neq \texttt{try}(\Box, \_.\_) \\
k \triangleright \texttt{try}(\Box, x.e_2) \ll \texttt{raise}\, v & \mapsto_{\mathsf{c}} & k > \{v/x\}e_2 &
\end{array}
$$

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:

$$\frac{\Gamma \vdash e : \tau_{exn}}{\Gamma \vdash \texttt{raise}(\tau, e) : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma, x : \tau_{exn} \vdash e_2 : \tau}{\Gamma \vdash \texttt{try}(e_1, x.e_1) : \tau}$$

In these rules, the exception value must be of some specified type τ_{exn}. The rule for typing 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. τ_{exn} could be `int`, in which case the exception value acts like a error code returned by a C library function. If we make τ_{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:

$$\frac{k \ : \tau \ \mathsf{stack}}{k \ll \texttt{raise}\, v \ \ \mathsf{OK}}$$

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

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 $\texttt{catch}(k, x.e_2)$ while the control stack contains the usual frames, and $\texttt{try}(\Box)$ (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.

$$
\begin{array}{lcl}
(h,k) > \texttt{apply}(e_1, e_2) & \mapsto_{\mathsf{c}} & (h, k \triangleright \texttt{apply}(\Box, e_2)) > e_1 \\
\ldots & & \\
(h,k) > \texttt{try}(e_1, x.e_2) & \mapsto_{\mathsf{c}} & (h \triangleright \texttt{catch}(k, x.e_2), k \triangleright \texttt{try}(\Box)) > e_1 \\
(h \triangleright \texttt{catch}(k', x.e_2), k \triangleright \texttt{try}(\Box)) < v_1 & \mapsto_{\mathsf{c}} & (h,k) < v_1 \\
(h,k) > \texttt{raise}(\tau, e) & \mapsto_{\mathsf{c}} & (h, k \triangleright \texttt{raise}(\tau, \Box)) > e \\
(h, k \triangleright \texttt{raise}(\tau, \Box)) < v & \mapsto_{\mathsf{c}} & (h,k) \ll \texttt{raise}\ v \\
(h \triangleright \texttt{catch}(k', x.e_2), k) \ll \texttt{raise}\ v & \mapsto_{\mathsf{c}} & (h, k') > \{v/x\}e_2
\end{array}
$$

Note that we do not unwind the control stack explicilty, 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:

```
exception E1
```

We can raise an exception using the syntax:

```
raise E1
```

Finally, we can handle exceptions using pattern matching, either using a particular exception type in the match or a wildcard:

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

Consider what happens in the following code:

```
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 interface, 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 system 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.

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.

`letcc` x `in` e `end`	`letcc`$(\tau, x.e)$
`throw` e_1 `to` e_2	`throw`(τ, e_1, e_2)

In brief, `letcc` x `in` e `end` captures the stack (= continuation) k in effect at the time the `letcc` is executed and substitutes `cont`(k) for x in e. we can later transfer control to k by throwing a value v to k with `throw` v `to` `cont`(k). Note that the stack k we capture can be returned passed point in which it was in effect. As a result, `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 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 + letcc x in 2 + (throw 3 to x) end` $\mapsto_{\mathsf{c}}^{*} 4$
(b) `1 + letcc x in 2 end` $\mapsto_{\mathsf{c}}^{*} 3$
(c) `1 + letcc x in if (throw 2 to x) then 3 else 4 fi end`
$\mapsto_{\mathsf{c}}^{*} 3$

Example (a) shows an upward use of continuations similar to exceptions, where the addition of $2 + \Box$ 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.

$$
\begin{array}{lcl}
k > \texttt{letcc}(\tau, x.e) & \mapsto_{\mathsf{c}} & k > \{\texttt{cont}(k)/x\}e \\
k > \texttt{throw}(\tau, e_1, e_2) & \mapsto_{\mathsf{c}} & k \triangleright \texttt{throw}(\tau, \Box, e_2) > e_1 \\
k \triangleright \texttt{throw}(\tau, \Box, e_2) < v_1 & \mapsto_{\mathsf{c}} & k \triangleright \texttt{throw}(\tau, v_1, \Box) > e_2 \\
k \triangleright \texttt{throw}(\tau, v_1, \Box) < \texttt{cont}(k_2) & \mapsto_{\mathsf{c}} & k_2 < v_1 \\
k > \texttt{cont}(k') & \mapsto_{\mathsf{c}} & k < \texttt{cont}(k')
\end{array}
$$

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.

$$\frac{\Gamma, x{:}\tau\ \mathsf{cont} \vdash e : \tau}{\Gamma \vdash \texttt{letcc}(\tau, x.e) : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_1\ \mathsf{cont}}{\Gamma \vdash \texttt{throw}(\tau, e_1, e_2) : \tau}$$

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.

$$\frac{k : \tau\ \mathsf{stack}}{\Gamma \vdash \texttt{cont}(k) : \tau\ \mathsf{cont}}$$

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
>
> ```
> compose : ('a -> 'b) -> 'b cont -> 'a cont
> ```
>
> so that `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

$$K_1 = K_2 \triangleright \texttt{apply}(F, \Box)$$

for some K_2. Then, it needs to throw the result to K. So

$$K_2 = K_3 \triangleright \texttt{throw}(\_, \Box, K)$$

and therefore

$$K_1 = K_3 \triangleright \texttt{throw}(\_, \Box, K) \triangleright \texttt{apply}(F, \Box)$$

for some K_3.

How can we create such a continuation? The expression

```
throw (F ...) to K
```

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

```
throw (F (letcc k1 in ... end)) to K
```

The next conundrum is how to return `k1` as the result of the `compose` function, now that we have captured it. Certainly, we can *not* just replace ... by `k1`, because the F would be applied (which is not only wrong, but also not type-correct). Instead we have to throw `k1` 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.

```
letcc r
in
   throw (F (letcc k1 in throw k1 to r end)) to K
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.

```
fun compose (f:'a -> 'b) (k:'b cont) : 'a cont is
  letcc r
  in
     throw (f (letcc k1 in throw k1 to r end)) to k
  end
end
```

In order to verify the correctness of this function, we can just calculate, using the operational semantics, what happens when `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).

$$\begin{array}{ll} & K_0 > \texttt{apply}(\texttt{apply}(\mathsf{compose}, F), K) \\ \mapsto_{\mathsf{c}}^* & K_0 > \texttt{letcc}(\_, r.\texttt{throw}(\_, \texttt{apply}(F, \texttt{letcc}(\_, k_1.\texttt{throw}(\_, k_1, r))), K)) \\ \mapsto_{\mathsf{c}} & K_0 > \texttt{throw}(\_, \texttt{apply}(F, \texttt{letcc}(\_, k_1.\texttt{throw}(\_, k_1, \texttt{cont}(K_0)))), K) \\ \mapsto_{\mathsf{c}} & K_0 \triangleright \texttt{throw}(\_, \Box, K) > \texttt{apply}(F, \texttt{letcc}(\_, k_1.\texttt{throw}(\_, k_1, \texttt{cont}(K_0)))) \\ \mapsto_{\mathsf{c}}^* & K_0 \triangleright \texttt{throw}(\_, \Box, K) \triangleright \texttt{apply}(F, \Box) > \texttt{letcc}(\_, k_1.\texttt{throw}(\_, k_1, \texttt{cont}(K_0))) \end{array}$$

At this point, we define

$$K_1 = K_0 \triangleright \texttt{throw}(\_, \Box, K) \triangleright \texttt{apply}(F, \Box)$$

and continue

$$\begin{array}{ll} \mapsto_{\mathsf{c}} & K_1 > \texttt{throw}(\_, K_1, \texttt{cont}(K_0)) \\ \mapsto_{\mathsf{c}} & K_0 < K_1 \end{array}$$

By looking at K_1 we can see hat 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 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 `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 `throw` are reversed which may be significant in some circumstances because of the left-to-right evaluation order.

Concrete MinML	Abstract MinML	SML of NJ
`letcc` x `in` e `end`	`letcc`$(\tau, x.e)$	`callcc (fn x => e)`
`throw` e_1 `to` e_2	`throw`(τ, e_1, e_2)	`throw e2 e1`

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.[3]

First, in our implementation, both expressions and value have type `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.

```
type value = exp
e : exp
v : value
```

Next, the stack k is represented by an ML function

```
k : value -> value
```

Applying this function to a value v will carry out the rest of the computation of the machine, returning the final answer. Finally, we have two functions

```
eval : exp -> (value -> value) -> value
return : value -> (value -> value) -> value
```

satisfying the specification:

(i) `eval` e k $\Downarrow$ a iff $k > e \mapsto_{\mathsf{c}}^* \bullet < a$

(ii) `return` v k $\Downarrow$ a iff $k < v \mapsto_{\mathsf{c}}^* \bullet < a$

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.

[3] We give some code excerpts here; the full code can be found at `http://www.cs.cmu.edu/~fp/courses/312/code/10-exceptions/`.

$k \triangleright$ `if`$(\Box, e_2, e_3)$	`(fn v1 => ifFrame (v1, e2, e3) k)`
$k \triangleright$ `apply`$(\Box, e_2)$	`(fn v1 => applyFrame1 (v1, e2) k)`
$k \triangleright$ `apply`$(v_1, \Box)$	`(fn v2 => applyFrame2 (v1, v2) k)`
$k \triangleright$ `let`$(\Box, x.e_2)$	`(fn v1 => letFrame (v1, ((), e2)) k)`
$\bullet$	`(fn v => v)`

The case of the empty stack corresponds to the initial continuation, which simply returns the value passed to it as the result of the overal 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, e1'))) 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 refered to as *continuation-passing style*. It is quite flexible and elegant.