Supplementary Notes on Continuations

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 10
September 30, 2004

In this lecture we first introduce *exceptions* [Ch. 13] and then *continuations* [Ch. 12], two advanced control construct available in some functional languages.

Exceptions are a standard construct in ML and other languages such as Java. We give here only a particularly simple form; a more elaborate form is pursued in Assignment 4. It is particularly easy to describe now that our abstract machines makes a control stack explicit.

We introduce a new form of state

$$k \ll \mathsf{fail}$$

which signals that we are propagating 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. We do not distinguish different exceptions, only failure.

We have two new forms of expressions `fail`[1] and $\mathtt{try}(e_1, e_2)$ (with concrete syntax $\mathtt{try}\, e_1 \,\mathtt{ow}\, e_2$). Informally, $\mathtt{try}(e_1, e_2)$ evaluates e_1 and returns its value. If the evaluation of e_1 fails, that is, an exception is raised, then we evaluate e_2 instead and returns its value (or propagate *its* exception). These rules are formalized in the C-machine as follows.

$$\begin{array}{lll}
k > \mathtt{try}(e_1, e_2) & \mapsto_{\mathsf{c}} & k \triangleright \mathtt{try}(\Box, e_2) > e_1 \\
k \triangleright \mathtt{try}(\Box, e_2) < v_1 & \mapsto_{\mathsf{c}} & k < v_1 \\
k > \mathtt{fail} & \mapsto_{\mathsf{c}} & k \ll \mathsf{fail} \\
k \triangleright f \ll \mathsf{fail} & \mapsto_{\mathsf{c}} & k \ll \mathsf{fail} \qquad \text{for } f \neq \mathtt{try}(\Box, _) \\
k \triangleright \mathtt{try}(\Box, e_2) \ll \mathsf{fail} & \mapsto_{\mathsf{c}} & k > e_2
\end{array}$$

[1] A type should be included here in order to preserve the property that every well-typed expression has a unique type, but we prefer not to complicate the syntax at this point.

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:

$$\frac{}{\Gamma \vdash \texttt{fail} : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{try}(e_1, e_1) : \tau}$$

We can now state (without proof) the preservation and progress properties. The proofs follow previous patterns (see [Ch. 13]).

1. (Preservation) If $s : \sigma$ and $s \mapsto s'$ then $s' : \sigma$.
2. (Progress) If $s : \sigma$ then either
   (i) $s \mapsto s'$ for some s', or
   (ii) $s = \bullet < v$ with v value, or
   (iii) $s = \bullet \ll$ fail.

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 `fail` expression can jump to it directly.

Overall, such a 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. However, we will not pursue this further, but move on to the discussion of continuations. Continuations are more flexible, but also more dangerous than exceptions.

Continuations 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.[2]

[2]See Assignment 4 for details on concrete syntax.

$\texttt{callcc}\, x \,\texttt{=>}\, e$	$\texttt{callcc}(x.e)$
$\texttt{throw}\, e_1 \,\texttt{to}\, e_2$	$\texttt{throw}(e_1, e_2)$

In brief, $\texttt{callcc}\, x \,\texttt{=>}\, e$ captures the stack (= continuation) k in effect at the time the `callcc` is executed and substitutes $\texttt{cont}(k)$ for x in e. we can later transfer control to k by throwing a value v to k with $\texttt{throw}\, v \,\texttt{to}\, \texttt{cont}(k)$. Note that the stack k we capture can be returned past the 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 as an alternative to exceptions, another is to implement co-routines or threads. Another use is to achieve backtracking.

As a starting example we consider simple arithmetic expressions.

(a) `1 + callcc x => 2 + (throw 3 to x)` $\mapsto_{\mathsf{c}}^{*} 4$
(b) `1 + callcc x => 2` $\mapsto_{\mathsf{c}}^{*} 3$
(c) `1 + callcc x => if (throw 2 to x) then 3 else 4 fi` $\mapsto_{\mathsf{c}}^{*} 3$

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.

$$
\begin{array}{lll}
k > \texttt{callcc}(x.e) & \mapsto_{\mathsf{c}} & k > \{\texttt{cont}(k)/x\}e \\[1ex]
k > \texttt{throw}(e_1, e_2) & \mapsto_{\mathsf{c}} & k \rhd \texttt{throw}(\Box, e_2) > e_1 \\
k \rhd \texttt{throw}(\Box, e_2) < v_1 & \mapsto_{\mathsf{c}} & k \rhd \texttt{throw}(v_1, \Box) > e_2 \\
k \rhd \texttt{throw}(v_1, \Box) < \texttt{cont}(k_2) & \mapsto_{\mathsf{c}} & k_2 < v_1 \\[1ex]
k > \texttt{cont}(k') & \mapsto_{\mathsf{c}} & k < \texttt{cont}(k')
\end{array}
$$

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.

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

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

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.

$$\frac{k : \tau \Rightarrow \sigma}{\Gamma \vdash \mathtt{cont}(k) : \tau\ \mathsf{cont}}$$

It looks like there could be a problem here, because σ, the final answer type of the continuation, does not appear in the conclusion. Fortunately, it works, but only because the final answer type σ of all continuations that may occur in a computation will be equal. To be precise, if we want to be talk about typing intermediate states of the computation, we would need to pass along the final answer type σ through the typing judgments.

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

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

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

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

and therefore

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

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 (callcc k1 => ...)) 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.

```
callcc r =>
   throw (F (callcc k1 => throw k1 to r)) to K
```

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
  callcc r =>
     throw (f (callcc k1 => throw k1 to r)) to k
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).

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

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[3] 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
`callcc` x `=>` e	$\texttt{callcc}(x.e)$	`callcc (fn x => e)`
`throw` e_1 `to` e_2	$\texttt{throw}(e_1, e_2)$	`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.

[3] `http://www.cs.cmu.edu/~fp/courses/312/code/10-continuations/`