Supplementary Notes on Continuations

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 10
September 26, 2002

In this lecture 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.[1]

`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.

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

(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.

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

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

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).

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

At this point, we define

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

and continue

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

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/`