Continuations

Offered as language construct of libraries by some functional languages (Scheme, SML of New Jersey)

- allow non-local transfer of control
- make a snapshot of control stack and restore it at some later point in time
- "goto" of functional languages

We need a language construct

- to make a "snapshot" of the current stack, and one
- to restore a given stack

Syntax:

letcc:: \( x \) in \( e \) endletcc(
\( ;x:e \))

throw:: \( e_1 \) in \( e_2 \) endthrow(\( ;e_1;e_2 \))

- letcc captures the current stack \( k \) as \( \text{continuation} \) in \( \text{cont}(k) \), and binds it to \( x \) in \( e \)
- throw evaluates both arguments and throws the value of \( e_1 \) to stack of continuation \( e_2 = \text{cont}(k) \)

Examples:

1 + letcc \( x \) in 2 + (throw 3 to \( x \)) end \( ^{\text{c}} \) 4
1 + letcc \( x \) in 2 end \( ^{\text{c}} \) 3
1 + letcc \( x \) in if (throw 7 to \( x \)) then 3 else 4 fi end \( ^{\text{c}} \) 8

Given the following definition (MinHs Plus):

\[
\text{mult} \[] = 1 \\
\text{mult} \ (x:xs) = x \ast (\text{mult} \ xs)
\]

- if there is one 0 in the list, the result will be 0
- rest of the list does not have to be considered
- how can we use continuations to terminate the execution and return 0 straight away?

\[
\begin{aligned}
\text{mult} \ xs &= \\
&= \text{letcc} \ k \\
in \text{let} \\
\text{mult'} \ [] &= 1 \\
\text{mult'} \ (x:xs) &= \\
in \text{if} \ (x = 0) \\
\text{then} \ &\text{throw} \ 0 \ \text{to} \ k \\
\text{else} \ &x \ast (\text{mult} \ x xs)
\end{aligned}
\]
Alternative implementation:

Continuations are "first class citizens": we can pass them as arguments

\[
\text{mult} \; \text{xs} = \\
\quad \text{let} \quad \text{mult'} \; \text{[]} \quad k = 1 \\
\quad \text{mult'} \; (x:x:s) \quad k = \\
\quad \quad \text{in} \quad \text{if} \; (x == 0) \\
\quad \quad \quad \text{then throw 0 to k} \\
\quad \quad \quad \text{else} \; x * \; (\text{mult'} \; x:s) \\
\quad \quad \text{in letcc ret in mult'} \; \text{xs} \; \text{ret}
\]

Stacks can be "thrown" as well:

- \text{letcc} \; r \; \text{in} \ldots \; \text{letcc} \; s \; \text{in} \ldots \; \text{throw} \; s \; r \; \text{end end}
- makes continuations much harder to implement than exceptions

Application of continuations:

To implement

- exceptions
- co-routines or threads
- backtracking

Types

Definition:

- a stack \( s \) has type \( \tau \) if its top-most frame accepts a value of type \( \tau \)
- a continuation \( \text{cont}(s) \) has type \( \tau \) if the stack \( s \) has type \( \tau \)

Semantics

\[
\begin{align*}
\text{letcc} \; (r,x:e) & \quad \text{cont}(s) : \tau \\
\text{throw} \; (r,e_1,e_2) & \quad \text{cont}(s) : \tau
\end{align*}
\]
Example:

Define a function which, given a continuation \( \text{cont}(k) \) and a function \( f \), returns a continuation \( \text{cont}(k_1) \) which applies \( f \) to its next value, and throws the result to \( s \).

\[
\text{compose}: (a \to b) \to b \text{ cont} \to \text{cont a}
\]

Steps:

1. \( k_1 = \text{apply}(f, \Box) \to \ldots \)
2. \( k_1 = \text{apply}(f, \Box) \to \text{throw}(\Box, k) \to \ldots \)
3. The expression \( \text{throw}(\ldots f(\ldots), k) \) creates a continuation of the above form
4. bind continuation to \( k_1 \): \( \text{throw}(\ldots f(\text{letcc}(k_3, \ldots), k) \) creates a continuation of the above form
5. return \( k_1 \): \( \text{letcc}(r, \text{throw}(\ldots f(\text{letcc}(k_3, \text{throw}(\ldots k_3, r)), k)) \) creates a continuation of the above form

\[
\text{fun compose (f:a->b) (k: b cont) -> a cont is}
\]

\[
\text{letcc r in}
\text{throw (f (letcc k1 in throw k1 to r end)) to k end end}
\]

How can

\[
\rightarrow \text{try}(e_1, x.e_2) \text{ and}
\rightarrow \text{raise}(r)
\]

be expressed in terms of continuations?

\[
\rightarrow \text{letcc}(k, e_1)
\rightarrow \text{throw}(r, \text{compose}(\text{fun}(f.x.e_2), k))
\]