CONTINUATIONS

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

- allow non-local transfer of control
- make a snap-shot of control stack and restore it at some later point in time
- “goto” of functional languages
CONTINUATIONS

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

- allow non-local transfer of control
- make a snap-shot of control stack and restore it at some later point in time
- “goto” of functional languages

We need a language construct

- to make a “snap-shot” of the current stack, and one
- to restore a given stack
Syntax:

letcc:: \( \tau \ x \ in \ e \ end \) \hspace{1cm} \text{letcc}(\tau, x.e)

throw:: \( \tau \ e_1 \ in \ e_2 \ end \) \hspace{1cm} \text{throw}(\tau, e_1, e_2)

\( \rightarrow \) letcc captures the current stack \( k \) as \textit{continuation} in \text{cont}(k), and binds it to \( x \) in \( e \)

\( \rightarrow \) throw evaluates both arguments and throws the value of \( e_1 \) to stack of continuation \( e_2 = \text{cont}(k) \)
Examples:

$\Rightarrow 1 + \text{letcc } x \text{ in } 2 + (\text{throw 3 to } x) \text{ end}$

$\Rightarrow 1 + \text{letcc } x \text{ in } 2 \text{ end}$

$\Rightarrow 1 + \text{letcc } x \text{ in if (throw 7 to } x) \text{ then 3 else 4 fi end}$
Examples:

$\Rightarrow 1 + \text{letcc } x \text{ in } 2 + \text{(throw 3 to } x) \text{ end } \xrightarrow{\ast} 4$

$\Rightarrow 1 + \text{letcc } x \text{ in } 2 \text{ end}$

$\Rightarrow 1 + \text{letcc } x \text{ in } \text{if (throw 7 to } x) \text{ then 3 else 4 fi end}$
Examples:

\[ \Rightarrow 1 + \text{letcc } x \text{ in } 2 + (\text{throw 3 to } x) \text{ end } \xrightarrow{\ast} c 4 \]

\[ \Rightarrow 1 + \text{letcc } x \text{ in } 2 \text{ end } \xrightarrow{\ast} c 3 \]

\[ \Rightarrow 1 + \text{letcc } x \text{ in if } (\text{throw 7 to } x) \text{ then 3 else 4 fi end} \]
Examples:

\[ \begin{align*}
&\rightarrow 1 + \text{letcc } x \text{ in } 2 + (\text{throw } 3 \text{ to } x) \text{ end } \overset{\ast}{\rightarrow}_c 4 \\
&\rightarrow 1 + \text{letcc } x \text{ in } 2 \text{ end } \overset{\ast}{\rightarrow}_c 3 \\
&\rightarrow 1 + \text{letcc } x \text{ in } \text{if } (\text{throw } 7 \text{ to } x) \text{ then } 3 \text{ else } 4 \text{ fi end } \overset{\ast}{\rightarrow}_c 8
\end{align*} \]
Given the following definition (MinHs Plus):

\[
\begin{align*}
mult \, [] & = 1 \\
mult \, (x:xs) & = x \ast (mult \, xs)
\end{align*}
\]

→ 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?
Given the following definition (MinHs Plus):

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

- 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?

\[
\text{mult} \ xs = \\
\text{let} \text{cc} \ k \\
\text{in} \ \text{let} \\
\quad \text{mult'} \ [\ ] = 1 \\
\quad \text{mult'} \ (x:xs) = \\
\quad \quad \text{in} \ \text{if} \ (x == 0) \\
\quad \quad \quad \text{then} \ \text{throw} \ 0 \ \text{to} \ k \\
\quad \quad \quad \text{else} \ x \times (\text{mult'} \ xs)
\]
Alternative implementation:

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

\[
\text{mult } \text{xs } = \\
\text{let } \\
\text{mult’ } \text{[]} \text{ } k = 1 \\
\text{mult’ } \text{(x:xs) } k = \\
\text{in if (x } \text{== } 0) \\
\text{then throw 0 to } k \\
\text{else x } \ast \text{ (mult’ } \text{xs)} \\
\text{in letcc ret in mult’ } \text{xs } \text{ret}
\]
Stackscan be "thrown" as well:

letccrin....letccsin.....throwsrendend

makes continuations much harder to implement than exceptions

Application of continuations:
To implement exceptions, co-routines or threads, backtracking
Stacks can be “thrown” as well:

- letcc r in .... letcc s in ..... throw s r end end
- makes continuations much harder to implement than exceptions
Stacks can be “thrown” as well:

→ letcc r in .... letcc s in ..... throw s r end end
→ makes continuations much harder to implement than exceptions

Application of continuations:

To implement
→ exceptions
→ co-routines or threads
→ backtracking
Definition:

- a stack $s$ has type $\tau$ if it’s top-most frame accepts a value of type $\tau$

- a continuation $\text{cont}(s)$ has type $\tau$ if the stack $s$ has type $\tau$

\[
\frac{\text{stack}}{s : \tau \quad \text{stack}}\quad \text{cont}(s) : \tau \quad \text{cont}
\]

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

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

\[
\begin{align*}
    s \triangleright \text{letcc}(\tau, x.e) & \implies_c s \triangleright \{\text{cont}(s)/x\}e \\
    s \triangleright \text{throw}(\tau, e_1, e_2) & \implies_c \text{throw}(\tau, \Box, e_2) \triangleright s \triangleright e_1 \\
    \text{throw}(\tau, \Box, e_2) & \triangleleft v_1 \implies_c \text{throw}(\tau, v_1, \Box) \triangleright s \triangleright e_2 \\
    \text{throw}(\tau, v_1, \Box) \triangleright s \triangleright \text{cont}(s_2) & \implies_c s_2 \triangleleft v_1 \\
    s \triangleright \text{cont}(s') & \implies_c s \triangleleft \text{cont}(s')
\end{align*}
\]
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 it’s next value, and throws the result to \( s \).

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

$\Rightarrow k_1 = \text{apply}(f, \square) \triangleright \ldots$
Steps:

\[ k_1 = \text{apply}(f, \square) \triangleright \ldots \]

\[ k_1 = \text{apply}(f, \square) \triangleright \text{throw}(-, \square, k) \triangleright \ldots \]
Steps:

$\rightarrow k_1 = \text{apply}(f, \square) \triangleright \ldots$

$\rightarrow k_1 = \text{apply}(f, \square) \triangleright \text{throw}(\_, \square, k) \triangleright \ldots$

$\rightarrow$ The expression $\text{throw}(\_, f(\ldots), k)$ creates a continuation of the above form
Steps:

$\Rightarrow \ k_1 = \text{apply}(f, \Box) \triangleright \ldots$

$\Rightarrow \ k_1 = \text{apply}(f, \Box) \triangleleft \text{throw}(\_, \Box, k) \triangleright \ldots$

$\Rightarrow \ \text{The expression throw}(\_, f(\ldots), k) \text{ creates a continuation of the above form}$

$\Rightarrow \ \text{bind continuation to } k_1: \ \text{throw}(\_, f(\text{letcc}(k_1 \ldots), k) \text{ creates a continuation of the above form}$
Steps:

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

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

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
How can
\[ \text{try}(e_1, x.e_2) \text{ and } \text{raise}(v) \]
be expressed in terms of continuations?
How can $\rightarrow \text{try}(e_1, x.e_2)$ and $\rightarrow \text{raise}(v)$ be expressed in terms of continuations?

$\rightarrow \text{letcc}(k. e_1)$
How can

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

be expressed in terms of continuations?

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