Continuations
COS 441
Princeton University
Fall 2004

Webster Says…
Main Entry: re-ify
Pronunciation: ’rA-&#39;&ndash;”fl, ’rE-
Function: transitive verb
Inflected Form(s): re-ified; re-ify-ing
Etymology: Latin res thing -- to regard
(something abstract) as a material or
concrete thing

First-Class Continuations
• Form of structured GOTO
• Can be used to implement exception
• Can be used to build co-routines or threads
• Available in Scheme, Ruby, and SML/NJ but not Standard ML
• Also useful as a programming abstraction for web-services!

Some Informal Examples
(throw e<sub>1</sub> to e<sub>2</sub>:τ)
Invoke continuation (e<sub>2</sub>:τ cont) with the
value of e<sub>1</sub>:τ
(letcc x in e:τ)
Evaluate e with x bound to a value of type τ cont. Expression evaluates to the value of e or result of thrown to x. Can throw to x multiple times and “rewind” to this control point

Example
Compute product of list elements with short circuit when 0 is encountered

fun mult.list (l:int list):int =
  letcc ret in
  let fun mult nil = 1
   | mult (0:::) = throw 0 to ret
   | mult (n::l) = n * mult l
  in mult l end

Example
Compute product of list elements with short circuit when 0 is encountered

fun mult.list l =
  let fun mult nil ret = 1
   | mult (0:::) ret = throw 0 to ret
   | mult (n::l) ret = n * mult l ret
  in letcc ret in (mult l) ret end
Trickier Example

fun compose (f, k) =
    letcc ret in
    throw (f (letcc r in throw r to ret)) to k

Compose a function with a continuation

Continuation Passing Style

• Can apply a *global* translation of a program and remove all uses of `throw` and `letcc`
• Won’t study in detail in class
  – CPS transform used in compilation of functional languages to machine code
• Understanding how continuations can be used is harder than understanding their semantics!

First-Class Continuations

| Numbers  | $n \in \mathbb{N}$ |
| Types    | $\tau ::= \text{int} \mid \tau \text{cont} \mid \ldots$ |
| Expr’s   | $e ::= n \mid +(e_1, e_2)$ |
|          | | $\mid \text{throw } e_1 \text{ to } e_2$ |
|          | | $\mid \text{letcc } x \text{ in } e$ |
|          | | $\mid K$ (* not user visible *) |
| Values   | $v ::= n \mid K$ |

Frames and Stacks

| Frames   | $f ::= +(v_1, \Box) \mid +(\Box, e_2)$ |
|          | $\mid \text{throw } v_1 \text{ to } \Box$ |
|          | $\mid \text{throw } \Box \text{ to } e_2$ |
| Stack    | $K ::= \bullet \mid f \triangleright K$ |

Dynamic Semantics

\[
(K, \text{letcc } x \text{ in } e) \mapsto (K, \{K/x\} e)
\]

\[
(\text{throw } v \text{ to } \Box \triangleright K, K') \mapsto (K', v)
\]

\[
(K, \text{throw } e_1 \text{ to } e_2) \mapsto (\text{throw } \Box \triangleright e_2 \triangleright K, e_1)
\]

\[
(\text{throw } \Box \triangleright e_2 \triangleright K, v_1) \mapsto (\text{throw } v_1 \text{ to } \Box \triangleright K, e_2)
\]

Dynamic Semantics

\[
(K, \text{letcc } x \text{ in } e) \mapsto (K, \{K/x\} e)
\]

• Capturing a continuation can be a linear time operation if not implemented cleverly
• Clever implementations make continuation capture constant time
• Compare this to two stack exception handler approach
Type Safety

- Must define when a machine state is well-formed
- Introduced the notion of types for stacks, frames, and also choose one arbitrary but fixed type for the empty stack \( \tau_{\text{ans}} \)

\[ \vdash K : \tau \text{ stack} \vdash e : \tau \]

(K, e) ok

Programmer Visible Rules

\[ \Gamma [x : \tau \text{ cont}] \vdash e : \tau \]
\[ \Gamma \vdash \text{letcc } x \text{ in } e : \tau \]
\[ \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_1 \text{ cont} \]
\[ \Gamma \vdash \text{throw } e_1 \text{ to } e_2 : \tau' \]

Auxiliary Types

Types \( \tau ::= \text{int} | \tau \text{ cont} \]
\[ | \tau \text{ stack} \]
\[ | (\tau, \tau) \text{ frame} \]

\[ \vdash K : \tau \text{ stack} \]
\[ \Gamma \vdash K : \tau \text{ cont} \]

Internal Type-Checking Rules

\[ \vdash \bullet : \tau_{\text{ans}} \text{ stack} \]
\[ \vdash F : (\tau, \tau') \text{ frame} \vdash K : \tau' \text{ stack} \]
\[ \vdash F \triangleright K : \tau \text{ stack} \]

Rules For Frames

\[ \vdash e_2 : \text{int} \]
\[ \vdash + (\square, e_2) : (\text{int}, \text{int}) \text{ frame} \]
\[ v_1 \text{ value} \vdash v_1 : \text{int} \]
\[ \vdash + (v_1, \square) : (\text{int}, \text{int}) \text{ frame} \]

Frame which expects a value of type \( \tau \) that can be used to fill the hole to produce an expression of type \( \tau' \)

Frame Rules (cont.)

\[ v, \text{ value} \vdash v : \tau \]
\[ \vdash \text{throw } v_1 \text{ to } \square : (\tau \text{ cont}, \tau) \text{ frame} \]
\[ \vdash e_1 : \tau \text{ cont} \]
\[ \vdash \text{throw } \square \text{ to } e_2 : (\tau, \tau') \text{ frame} \]

Harper’s text has a small bug rules above are correct.
Example: Continuation Capture

\[
\begin{align*}
\bullet & \to (+\langle 1, 2 \rangle, \bullet, \text{letcc } x \text{ in } 1) \\
+ & \to (+\langle 2 \rangle, \bullet, 1) \\
\bullet & \to (+\langle 1, \bullet \rangle, 2) \\
+ & \to (\bullet, 3)
\end{align*}
\]

Example: Thrown Continuation

\[
\begin{align*}
\bullet & \to (+\langle \text{letcc } x \text{ in } (+1, \text{throw } 2 \text{ to } x), 3 \rangle) \\
+ & \to (+\langle 3 \rangle, \bullet, \text{letcc } x \text{ in } (+1, \text{throw } 2 \text{ to } x)) \\
\bullet & \to (+\langle 1, \bullet \rangle, (+1, \text{throw } 2 \text{ to } (+\langle 3 \rangle, \bullet, 1)), \text{throw } 2 \text{ to } (+\langle 3 \rangle, \bullet, 1)) \\
+ & \to (+\langle 3 \rangle, \bullet, (+1, \bullet), (+1, \bullet)) \\
\bullet & \to (+\langle \text{throw } \bullet \text{ to } (+\langle 3 \rangle, \bullet), (+1, \bullet), (+1, \bullet)) \\
(\text{throw } \bullet \text{ to } (+\langle 3 \rangle, \bullet)) & \to (+\langle \text{throw } \bullet \text{ to } (+\langle 3 \rangle, \bullet), (+1, \bullet), (+1, \bullet)) \\
\langle 2, \bullet \rangle & \to (\bullet, 3) \\
+ & \to (\bullet, 3)
\end{align*}
\]

Stacks and Frames as HOF

One big insight of the CPS transform is that you can represent stacks as HOF:

\[
\begin{align*}
\text{rep}(\bullet) & \equiv \lambda . x . x \\
\text{rep}(+(v_1, \bullet) : K) & \equiv \lambda . x . (\text{rep}(K) + (v_1, x)) \\
\text{rep}(+(\bullet, e_2) : K) & \equiv \lambda . x . (\text{rep}(K) + (x, e_2)) \\
\text{rep}(\text{throw } v_1 \text{ to } \bullet : K) & \equiv \lambda . x . (v v) \\
\text{rep}(\text{throw } \bullet \text{ to } e_2 : K) & \equiv \lambda . x . (e_2 x)
\end{align*}
\]

Producer Consumer

```
fun p(n) = (put(n); p(n+1))
funs c() = let
val n = get()
in printInt(n);
c()
end
```

Push Model

```
fun p(n) = (c(n); p(n+1))
fun c(n) = printInt(n)
```

Pull Model

```
fun p(n,s) = (n, n+1)
fun c(n,s) = let
val (n, s) = p(s)
in printInt(n);
c(s)
end
```
Coroutine Model

```ml
fun p(n, s) = let
    val s = put(n, s)
  in p(n+1, s) end

fun c(s) = let
    val (n, s) = get(s)
    in printInt(n); c(s) end
```

Implementation

```ml
datatype state = S of state cont
val buf = ref 0

val resume(S k) = callcc(fn k' => throw k (S k'))

fun put(n, s) = (buf := n; resume s)

fun get(s) = (!buf, resume s)
```

Threads

```ml
signature THREADS = sig
  exception NoMoreThreads
  val fork : (unit -> unit) -> unit
  val yield : unit -> unit
  val exit : unit -> 'a
end
```

User-level threads implemented with `callcc`

Summary

- First-Class continuations very powerful construct
- “Simple” semantics but allows powerful control constructs
  - Non-local exits
  - Coroutines
  - User-level threads