510: Programming Languages

First-Class Continuations

David Walker

Fall, 2002

First-class continuations

Seize the **current continuation**: $\texttt{letcc}\,x\,\texttt{in}\,e$.

- Reify the current control stack (current continuation) K.
- Bind x to K.
- Evaluate e.

Pass control to a **reified** continuation: $\texttt{throw}\,e_1\,\texttt{to}\,e_2$.

- Evaluate e_1 to a value v_1.
- Evaluate e_2 to a continuation (stack) K.
- Pass v_1 to K.

Static Semantics

Typing rules:

$$\frac{\Gamma[x{:}\tau\,\texttt{cont}] \vdash e : \tau}{\Gamma \vdash \texttt{letcc}\,x\,\texttt{in}\,e : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_1\,\texttt{cont}}{\Gamma \vdash \texttt{throw}\,e_1\,\texttt{to}\,e_2 : \tau'}$$

Result type of `throw` is arbitrary because it doesn't return.

Type of reified continuation is the type of the body of `letcc`.

Example: Composition

Problem: composing a continuation with a function.

- Given: a function f of type $\tau' \to \tau$ and a continuation k of type $\tau\,\mathtt{cont}$;

- Return: a continuation k' of type $\tau'\,\mathtt{cont}$ that, when thrown a value v' of type τ', throws $f(v')$ to k.

Example: Composition

Steps of the solution:

- Visualize the continuation we want.
- Find a way to construct it using `letcc`.
- Find a way to return it using the "early return" trick.

Example: Composition

The continuation we want: $\texttt{throw}\ (f\ \square)\ \texttt{to}\ k$.

If we fill the hole with v', then

- f is applied to v'
- the result is thrown to k

Example: Composition

How do we obtain that continuation?

```
fun compose (f, k) =
    ... throw (f ...) to k ...
```

We want the continuation at the argument to f.

Example: Composition

Idea: seize the continuation using `letcc`:

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

The variable `r` is bound to the desired continuation.

Example: Composition

Return the continuation using short-circuit return:

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

Question: what is the type of `ret`?
Answer: τ' **cont cont**.

Semantics of Continuations

Extend MinML with types $\tau\,\texttt{cont}$ and these expressions:

$$e \;::=\; \ldots \mid \texttt{letcc}\,x\,\texttt{in}\,e \mid \texttt{throw}\,e_1\,\texttt{to}\,e_2 \mid K$$

Note: `letcc` binds x in e.

Control stacks are values.

Dynamic Semantics

Specify evalation order:

$$\frac{}{(K, \texttt{throw}\, e_1\, \texttt{to}\, e_2) \mapsto (\texttt{throw}\, \Box\, \texttt{to}\, e_2 \triangleright K, e_1)}$$

$$\frac{}{(\texttt{throw}\, \Box\, \texttt{to}\, e_2 \triangleright K, v_1) \mapsto (\texttt{throw}\, v_1\, \texttt{to}\, \Box \triangleright K, e_2)}$$

Dynamic Semantics

`letcc` duplicates control stack:

$$\overline{(K, \texttt{letcc}\, x\, \texttt{in}\, e) \mapsto (K, \{K/x\}e)}$$

`throw` abandons current control stack:

$$\overline{(\texttt{throw}\, v\, \texttt{to}\, \Box \triangleright K, K') \mapsto (K', v)}$$

Example

Let's trace the execution of $e = \texttt{compose(}F\texttt{,}K\texttt{)}$, where $F : \tau' \to \tau$ and $K : \tau\,\texttt{cont}$.

$$
\begin{array}{ll}
& (K_0, e) \\
\mapsto^*_{\mathsf{C}} & (K_0, \texttt{letcc}\, r\, \texttt{in}\, \texttt{throw}\, \texttt{(}F\texttt{(letcc}\, x\, \texttt{in}\, \texttt{throw}\, x\, \texttt{to}\, r\texttt{))}\, \texttt{to}\, K) \\
\mapsto^*_{\mathsf{C}} & (K_0, \texttt{throw}\, \texttt{(}F\texttt{(letcc}\, x\, \texttt{in}\, \texttt{throw}\, x\, \texttt{to}\, K_0\texttt{))}\, \texttt{to}\, K) \\
\mapsto^*_{\mathsf{C}} & (\underbrace{F\texttt{(}\square\texttt{)} \triangleright \texttt{throw}\, \square\, \texttt{to}\, K \triangleright K_0}_{K'}, \texttt{letcc}\, x\, \texttt{in}\, \texttt{throw}\, x\, \texttt{to}\, K_0) \\
\mapsto^*_{\mathsf{C}} & (K', \texttt{throw}\, K'\, \texttt{to}\, K_0) \\
\mapsto^*_{\mathsf{C}} & (K_0, K')
\end{array}
$$

So the continuation K' is returned to K_0, as desired. But is K' the desired continuation?

Example

Let $F = \texttt{fun}\, f\ \texttt{(}x\texttt{:}\tau\texttt{)}\texttt{:}\tau' = e$.

$$
\begin{array}{ll}
\multicolumn{2}{l}{(K_0, \texttt{throw}\, V\, \texttt{to}\, K')} \\
\mapsto_{\mathsf{C}} & (K', V) \\
= & (F\,\texttt{(}\Box\texttt{)} \triangleright \texttt{throw}\, \Box\, \texttt{to}\, K \triangleright K_0, V) \\
\mapsto_{\mathsf{C}} & (\texttt{throw}\, \Box\, \texttt{to}\, K \triangleright K_0, \{F, V/f, x\}e) \\
\mapsto_{\mathsf{C}}^{*} & (\texttt{throw}\, \Box\, \texttt{to}\, K \triangleright K_0, V') \\
\mapsto_{\mathsf{C}} & (K, V')
\end{array}
$$

This is the desired behavior!

Safety

Well-formed states:

$$\frac{\vdash K : \tau\,\texttt{stack} \qquad \vdash e : \tau}{\vdash (K, e)\ \mathsf{ok}}$$

Theorem 1 (Preservation)
If (K, e) ok *and* $(K, e) \mapsto (K', e')$, *then* (K', e') ok.

Safety

Lemma 2 (Canonical Forms)
If $\vdash v : \tau\,\texttt{cont}$*, then* $v = K$ *for some control stack* K *such that* $\vdash K : \tau\,\texttt{stack}$*.*

This is easily proved by induction on typing.

Theorem 3 (Progress)
If (K, e) ok *then either* $K = \bullet$ *and* e value*, or there exists* K' *and* e' *such that* $(K, e) \mapsto (K', e')$*.*

Left as an exercise!

User-Level Threads

Create new threads dynamically using **fork**.

- Create a new child thread.
- Parent thread continues.

Kill threads using **exit**.

- Can be called explicitly.
- Implicit **exit** upon termination.

User-Level Threads

Cede control to another thread by calling **yield**.

- A coroutine hand-off to the scheduler itself.

- No need to specify next routine to run.

Scheduler resumes some ready thread:

- Maintain a **ready** queue of threads available to run.

- Select a routine from the ready queue and resume it.

A Simple Thread Interface

A simple interface for user-level threads:

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

Implementing User-Level Threads

- A thread is a **unit cont**.
- Maintain a ready queue of threads.
- Yield saves current continuation and dispatches to another one in the queue.
- Exit throws away current continuation and dispatches.
- Fork runs a new thread with given code; suspends parent.

User-Level Threads

```
structure Threads :> THREADS = struct

    open SMLofNJ.Cont

    exception NoRunnableThreads

    type thread = unit cont

    val readyQueue : thread Queue.queue =
        Queue.mkQueue()

    fun dispatch () =
        let
            val t = Queue.dequeue readyQueue
                    handle Queue.Dequeue =>
                      raise NoRunnableThreads
        in
            throw t ()
        end

    ⋮

end
```

Implementing User-Level Threads

```
structure Threads :> THREADS = struct
    ⋮

    fun exit () = dispatch()

    fun enqueue t = Queue.enqueue (readyQueue, t)

    fun fork f =
        callcc (fn parent =>
            (enqueue parent; f (); exit()))

    fun yield () =
        callcc (fn parent =>
            (enqueue parent; dispatch()))
end
```

Producer-Consumer Threads

Naïve implementation:

```
structure Client = struct

    open Threads

    val buffer : int ref = ref (~1)

    fun producer (n) =
        (buffer := n ; yield () ; producer (n+1))

    fun consumer () =
        (print (Int.toString (!buffer));
         yield (); consumer())

    fun run () =
        (fork (consumer); producer 0)

end
```

Producer-Consumer Threads

This **only** works because:

- Scheduler is a strict FIFO.
- There are only two threads that alternate execution on `yield`.
- There are no asynchronous interrupts.

Producer-Consumer Threads

Avoiding reliance on some scheduling assumptions:

```
structure Client = struct

    open Threads

    val buffer : int option ref = ref NONE

    (* avoid trampling on our own feet *)
    fun producer (n) =
        (case !buffer
           of NONE =>
              (buffer := SOME n ;
               yield() ;
               producer (n+1))
            | SOME _ =>
              (yield ();
               producer (n)))

    ⋮

end
```

Producer-Consumer Threads

```
structure Client = struct

    ⋮

    (* avoid repeatedly printing the same value *)
    fun consumer () =
        (case !buffer
           of NONE => (yield (); consumer())
            | SOME n =>
              (print (Int.toString n);
               buffer := NONE;
               yield();
               consumer()))

    fun run () =
        (fork (consumer); producer 0)

end
```

Asynchronous Events and Locking

- Should not rely on strict alternation of threads, or any predictable order of execution.

- Must assume demonic behavior by the other threads: a context switch will happen at the worst possible moment.

To limit possible interactions we need a **synchronization** mechanism.

- Ensure atomicity of **transactions**, sequences of operations on shared mutable storage.

- Preclude context switches within **critical regions**.

CML to the rescue ....

Rolling Your Own Continuations

We can avoid **letcc** and **throw** by keeping a "copy" of the control stack!

- **Maintain** a representation of the stack as a **function**.

- **Pass control** to the stack by **application**.

- **Seize** the stack by ordinary binding.

Requires a **systematic** and **global** programming style, called **continuation-passing style**.

Rolling Your Own Continuations

Example: multiplying elements of a list:

```
fun cps_mult nil k = k 1
  | cps_mult (n::l) k =
      cps_mult l (fn r => k (n * r))
fun mult l = cps_mult l (fn r => r)
```

Rolling Your Own Continuations

Correspondences:

- Frames corresponds to functions. For example, the frame `n *` $\Box$ corresponds to `fn r => n * r`.

- Pushing frames corresponds to composition: For example, the stack `n *` $\Box \triangleright K$ corresponds to the function `k o (fn r => n * r)`.

Rolling Your Own Continuations

With explicit continuations you no longer need `letcc` or `throw`.

```
fun cps_mult_list l k =
    let fun cps_mult nil k0 k = k 1
      | fun cps_mult (0::_) k0 k = k0 0
      | fun cps_mult (n::l) k0 k =
          cps_mult k0 l (fn p => k (n*p))
    in cps_mult l k k end
```

Notice that `k0` never changes; it is always the return continuation for `cps_mult_list`!

Continuation-Passing Style

The SML/NJ compiler works by systematic conversion into CPS form.

- All functions take a continuation argument.

- No function ever "returns", instead it calls its continuation.

- `letcc` and `throw` are "compiled away".

Continuation-Passing Style

An **expression** of type τ is transformed into an expression of type

$$\tau^\dagger = (\tau^* \to \alpha) \to \alpha.$$

Here α is a chosen **answer** type for the ultimate value of a complete program.

A **value** of type τ is transformed into a value of type τ^*:

$$\begin{array}{rcl} \mathtt{int}^* & = & \mathtt{int} \\ \mathtt{bool}^* & = & \mathtt{bool} \\ (\tau_1 \to \tau_2)^* & = & \tau_1^* \to \tau_2^\dagger \\ & = & \tau_1^* \to (\tau_2^* \to \alpha) \to \alpha \end{array}$$

Continuation-Passing Style

For example, the first version of `cps_mult` has type

$$\texttt{int list -> (int ->}\ \alpha\texttt{) ->}\ \alpha$$

The second version has type

$$\texttt{int list -> (int ->}\ \alpha\texttt{) -> (int ->}\ \alpha\texttt{) ->}\ \alpha$$

which takes both a “success” and a “fail” continuation.

Summary

Continuations are fairly cool:

- Provide a **user-level** notion of concurrency.
- No need for heavyweight OS processes, context switch, or system calls.
- And, of course, they have an elegant semantics.