510: Programming Languages

First-Class Continuations

David Walker

Fall, 2002

Stack Reification

The semantics for exceptions **reifies** the control stack.

- Saves control stack on the handler stack.
- Replaces the control stack with a saved stack.

Stack Reification

This is cheap because every saved stack is a **prefix** of the control stack.

- Save a “finger” on the stack. Pop back to the finger on restore.

- Similar to `setjmp` and `longjmp` in C.

Stack Reification

But `setjmp` and `longjmp` are not safe!

- Can `setjmp` inside a procedure, then return.
- Subsequent `longjmp` returns to a defunct (overwritten) stack!

These primitives promise more than they can deliver!

First-Class Continuations

Can we **safely** reify control stacks without worrying about whether they'll expire?

- Yes, because that's what Unix does internally to switch processes.

- Yes, and we can do it at the language level, rather than the OS level.

Key idea: use a **persistent** representation of the control stack.

Persistent and Ephemeral Data Structures

Data structures in conventional imperative languages are **ephemeral**.

- Insertion into a linked list mutates the list. The old version is lost.

- Pushing onto a stack modifies the stack pointer and writes on the underlying memory. Popping writes the stack pointer.

It is difficult to avoid ephemeral data structures in these languages.

Persistent and Ephemeral Data Structures

Data structures in functional languages are **persistent**.

- Inserting an element into a list yields a new list. The old version is still available.
- Stacks can be implemented so that pushing yields a new stack, leaving the old stack still available.

ML supports **both** persistent **and** ephemeral data structures.

- Reference cells are the fundamental ephemeral structure.

Ephemeral Stack Representations

Conventional run-time systems use an ephemeral representation of the stack.

- There's only one stack active at a time.
- Push and pop destructively update the stack.

These representations prevent efficient reification of the stack!

Persistent Stack Representations

But we can use a persistent representation instead!

- For example, can represent a stack as a linked list of frames.

- Persistent push and pop operations admit multiple copies of a stack.

- Rely on GC to collect unused copies.

Using this we can implement first-class continuations safely.

Informal Overview

Introduce a type τ **`cont`** of continuations.

- Values are reified control stacks.
- No "constants" for this type (*cf.*locations).
- Two operations: **`letcc`** and **`throw`**.

Informal Overview

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

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

Grab the current control point (continuation) for use elsewhere.

Informal Overview

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.

"Jump" to a given continuation, passing a value.

Continuations in SML/NJ

To seize a continuation use `callcc`:

- `callcc : ('a cont -> 'a) -> 'a`

- For `letcc` x `in` e, write `callcc (fn` x `=>` e`)`.

To throw a value to a continuation:

- `throw : 'a cont -> 'a -> 'b`

- For `throw` e_1 `to` e_2, write `throw` e_2 e_1.

Informal Overview

Crucial intuition: the **current continuation** is the current control stack at the point of evaluation.

- Evaluation builds up the stack incrementally.

- The stack "unwinds" to an expression.

Remember: continuations **only** arise as reified control stacks!

Example: Early Return

Problem: multiply the integers in a list, stopping early on zero.

Solution: bind an "escape" point for the return.

```
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: Early Return

A slicker formulation:

```
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
```

Example: Composition

Problem: composing a continuation with a function.

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

- Return: a continuation k' of type $\tau'\,\texttt{cont}$ 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: **`throw`** $(f\ \Box)$ **`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 r)) to k
```

Question: what is the type of `return`?
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.

Static Semantics

Typing rules:

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

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

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

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:

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

`throw` abandons current control stack:

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

Example

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

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

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

Example

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

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

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.

Proof of Preservation

Suppose that $(K, \texttt{letcc}\, x\, \texttt{in}\, e) \mapsto (K, \{K/x\}e)$ and that $(K, \texttt{letcc}\, x\, \texttt{in}\, e)$ ok.

Then there exists τ such that $K : \tau\, \texttt{stack}$ and $\texttt{letcc}\, x\, \texttt{in}\, e : \tau$.

Hence $x{:}\tau\, \texttt{cont} \vdash e : \tau$.

Hence $\{K/x\}e : \tau$.

Hence $(K, \{K/x\}e)$ ok.

Proof of Preservation

Suppose that $(K, \texttt{throw}\, v' \,\texttt{to}\, K') \mapsto (K', v')$ and that $(K, \texttt{throw}\, v' \,\texttt{to}\, K')$ ok.

Then there exists τ such that $K : \tau\, \texttt{stack}$ and $\texttt{throw}\, v' \,\texttt{to}\, K' : \tau$.

Hence there exists τ' such that $v' : \tau'$ and $K' : \tau' \,\texttt{cont}$.

Hence (K', v') 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!

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

```
int list -> (int -> α) -> α
```

The second version has type

```
int list -> (int -> α) -> (int -> α) -> α
```

which takes both a "success" and a "fail" continuation.

Summary

Continuations are **reified** control stacks.

- Seized by `letcc`, activated by `throw`.
- Values of type τ `cont` are continuations accepting values of type τ.

Continuations are a powerful programming mechanism!