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.

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.
Informal Overview

Seize the current continuation: \texttt{letcc \texttt{x 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.

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.

Persistent and Ephemeral Data Structures

Data structures in functional languages are \textbf{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 \textbf{both} persistent and ephemeral data structures.

- Reference cells are the fundamental ephemeral structure.
Continuations in SML/NJ

To seize a continuation use callcc:

- \texttt{callcc : ('a cont -> 'a) -> 'a}
- For letcc \texttt{x in e}, write callcc (fn \texttt{x => e}).

To throw a value to a continuation:

- \texttt{throw : 'a cont -> 'a -> 'b}
- For \texttt{throw \ e1 to \ e2}, write \texttt{throw \ e1 \ e2}.

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.

\begin{verbatim}
fun mult_list (l:int list):int = 
  letcc ret:int cont in
    let fun mult nil ret = 1
      | mult (0::[]) ret = throw 0 to ret
      | mult (n::l) ret = n * mult l ret
    in
      mult l ret end
  end
\end{verbatim}

Example: Composition

Problem: composing a continuation with a function.

- Given: a function \( f \) of type \( 	au' \rightarrow \tau \) and a continuation \( k \) of type \( \tau \cont \);
- Return: a continuation \( k' \) of type \( \tau' \cont \) that, when thrown a value \( v' \) of type \( \tau' \), throws \( f(v') \) to \( k \).

Steps of the solution:

- Visualize the continuation we want.
- Find a way to construct it using \texttt{letcc}.
- Find a way to return it using the “early return” trick.
Example: Composition

The continuation we want: \( \text{throw } (f \ □) \) to \( k \).

If we fill the hole with \( v' \), then

- \( f \) is applied to \( v' \)
- the result is thrown to \( k \)

Example: Composition

Idea: seize the continuation using \text{letcc}:

\[
\text{fun compose} \ (f, k) = \\
\text{... throw } (f \ (\text{letcc } r: \tau': \text{cont in } \ldots)) \text{ to } k \ldots
\]

The variable \( r \) is bound to the desired continuation.

Example: Composition

How do we obtain that continuation?

\[
\text{fun compose} \ (f, k) = \\
\text{letcc ret:? in} \\
\text{throw } (f \ (\text{letcc } r: \tau' \text{cont in } \text{throw } r \text{ to ret})) \text{ to } k
\]

We want the continuation at the argument to \( f \).

Example: Composition

Return the continuation using short-circuit return:

\[
\text{fun compose} \ (f, k) = \\
\text{letcc ret:? in} \\
\text{throw } (f \ (\text{letcc } r: \tau' \text{cont in } \text{throw } r \text{ to ret})) \text{ to } k
\]

Question: what is the type of \( \text{ret} \)?

Answer: \( \tau' \text{cont cont} \).

Semantics of Continuations

Extend MinML with types \( \text{cont}(\tau) \) and these expressions:

\[
e ::= \ldots | \text{letcc } x \text{ in } e | \text{throw} _1 \text{ to } e_2 | \text{cont} (k)
\]

Note: \text{letcc} binds \( x \) in \( e \).

Control stacks are values.

Static Semantics

Typing rules:

\[
\frac{\Gamma, x: \text{cont}(\tau) \vdash e : \tau \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \text{cont}(\tau_1)}{
\Gamma \vdash \text{letcc } x \text{ in } e : \tau} \\
\frac{\Gamma \vdash \text{throw} _1 \text{ to } e_2 : \tau'}{
\Gamma \vdash e : \tau'} \\
\frac{k: \text{stack}(\tau)}{
\Gamma \vdash \text{cont}(k) : \text{cont}(\tau)}
\]

Result type of \text{throw} is arbitrary because it doesn’t return.

Type of reified continuation is the type of the body of \text{letcc}.

Stack \( k \) accepting values of type \( \tau \) is a continuation value \( \text{cont}(k) \) of type \( \text{cont}(\tau) \).
Extended C Machine: Stack Frames and Values

Two new stack frames to record pending computations:

- \( e_2 \) expr
- \( e_1 \) value

Typing rules for the new frames:

- \( e_2 : \text{cont}(\tau) \)
- \( e_1 : \tau \)
- \( e_1 \) value

Every reified control stack is a value:

\[
\begin{array}{c}
\text{k stack} \\
\text{cont(k) value}
\end{array}
\]

Dynamic Semantics

letcc duplicates control stack:

\[
k' > \text{letcc } x \text{ in } e \leadsto C k' > \{k/x\}e
\]

throw abandons current control stack:

\[
\text{throw } v \text{ to } \square k' \leadsto C k' < v
\]

Example

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

\[
k_0 > \text{throw } v \text{ to } k' \leadsto C k' < v
\]

This is the desired behavior!

Dynamic Semantics

Specify evaluation order:

- \( k' > \text{throw } e_1 \text{ to } e_2 \leadsto C \text{throw } \square e_2, k \leadsto e_1 \)

- \( \text{throw } \square e_2, k < e_1 \leadsto C \text{throw } e_1 \text{ to } \square, k \leadsto e_2 \)

Safety

Well-formed states:

\[
\vdash k : \text{stack}(\tau) \quad \vdash e : \tau \\
\vdash \langle (k, e) \rangle \text{ ok}
\]

Theorem 1 (Preservation)

If \( (k, e) \text{ ok and } (k, e) \leadsto C (k', e') \), then \( (k', e') \text{ ok.} \)
Proof of Preservation

Suppose that $k > \text{letcc } x \in e \mapsto C \ k < \{k/x\}e$ and that $k > \text{letcc } x \in e \ ok$.

Then there exists $\tau$ such that $k : \text{stack}(\tau)$ and $\text{letcc } x \in e : \tau$.

Hence $x : \text{cont}(\tau) \vdash e : \tau$.

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

Hence $k < \{k/x\}e \ ok$.

Proof of Preservation

Suppose that $k > \text{throw } v' \text{ to } k' \mapsto C \ k' < v'$ and that $k > \text{throw } v' \text{ to } k' \ ok$.

Then there exists $\tau$ such that $k : \text{stack}(\tau)$ and $\text{throw } v' \text{ to } k' : \tau$.

Hence there exists $\tau'$ such that $v' : \tau'$ and $k' : \text{stack}(\tau')$.

Hence $(k', v') \ ok$.

Safety

Lemma 2 (Canonical Forms)

If $\vdash v : \text{cont}(\tau)$, then $v = k$ for some control stack $k$ such that $\vdash k : \text{stack}(\tau)$.

This is easily proved by induction on typing.

Theorem 3 (Progress)

If $(k, e) \ ok$ then either $k = \epsilon$ 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 \text{letcc} and \text{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)
```

Correspondences:

- Frames corresponds to functions. For example, the frame $(n, \square)$ corresponds to fn $r \mapsto n * r$.

- Pushing frames corresponds to composition: For example, the stack $(n, \square); k$ corresponds to the function $k \circ (fn r \mapsto n * r)$. 

Rolling Your Own Continuations

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

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

Notice that \( k_0 \) never changes; it is always the return continuation for `cps_mult_list`!

Continuation-Passing Style

An **expression** of type \( \tau \) is transformed into an expression of type

\[
\tau^\dagger = (\tau^\ast \to \alpha) \to \alpha. 
\]

Here \( \alpha \) is a chosen **answer** type for the ultimate value of a complete program.

A **value** of type \( \tau \) is transformed into a value of type \( \tau^\ast \):

\[
\begin{align*}
\text{int}^\ast & = \text{int} \\
\text{bool}^\ast & = \text{bool} \\
(\tau_1 \to \tau_2)^\ast & = \tau_1^\ast \to \tau_2^\ast \\
& = \tau_1^\ast \to (\tau_2^\ast \to \alpha) \to \alpha
\end{align*}
\]

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

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

\[
\text{int list} \to (\text{int} \to \alpha) \to \alpha 
\]

The second version has type

\[
\text{int list} \to (\text{int} \to \alpha) \to (\text{int} \to \alpha) \to \alpha 
\]

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

Summary

Continuations are **refied** control stacks.

- Seized by `letcc`, activated by `throw`.
- Values of type \( \text{cont}(\tau) \) are continuations accepting values of type \( \tau \).

Continuations are a powerful programming mechanism!