Exceptions

To make things simple we’ll first consider a simple failure mechanism.

- Like exceptions, but no associated values.
- Separates control aspects from data aspects.

Then we’ll consider value-carrying exceptions.

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expr</td>
<td>e :=</td>
<td>fail[r]</td>
<td>fail</td>
</tr>
<tr>
<td></td>
<td></td>
<td>catch(e₁,e₂)</td>
<td>try e₁ ov e₂</td>
</tr>
</tbody>
</table>

Think of fail as raising a fixed, anonymous exception.

Static Semantics of Exceptions

No surprises here:

\[
\Gamma \vdash \text{fail}[\tau] : \tau \\
\Gamma \vdash \text{catch}(e₁,e₂) : \tau
\]

Failures have any type.

Normal and failure return for handler must have the same type.

Dynamic Semantics of Exceptions

Use the \( K[\text{sat} \rightarrow] \) abstract machine since it provides access to the control stack.

An additional form of state:

- a failure state \( k \) corresponds to passing a failure to control stack \( k \).

An additional frame:

\[
\begin{align*}
& e₂ \text{ exp} \\
\Rightarrow & \text{catch}(\neg, e₂) \text{ frame}
\end{align*}
\]
Dynamic Semantics of Exceptions

VF "catch" clause fails: unwind stack to nearest enclosing handler.

Then invoke pending handler.

The Modified K{\text{naf}}\rightarrow Abstract Machine

States as just described:

- \text{\textit{initial}}: \varepsilon \mapsto e
- \text{\textit{final}}: \varepsilon \mapsto e
- Transition: \mapsto \rightarrow as given by structural semantics rules.

Type Safety

With suitable extensions to definitions of stack typing for the K{\text{naf}} abstract machine, type safety proved as previously, but with different meaning since final state can now represent an uncaught exception.

Theorem 1 (Preservation)

If \( s \text{\textit{ok}} \) and \( s \mapsto \rightarrow s' \), then \( s' \text{\textit{ok}} \).

Theorem 2 (Progress)

If \( s \text{\textit{ok}} \) then either \( s \text{\textit{final}} \), or there exists \( s' \) such that \( s \mapsto \rightarrow s' \).

Value-Passing Exceptions

It's important to be able to distinguish different sorts of failures:

- Arithmetic overflow
- Match and bind failures
- User-defined failures

Division by zero, arithmetic overflow.

Solution: pass values along with exceptions.

Arguments to \text{\textit{raise}}(f) are evaluated to determine value passed to handler:

\text{\textit{handle}}(h x . e) binds variable \( x \) in handler \( h \) to value passed with exception raised during execution of \( e \).

An extension to the failure state:

- \( k \mapsto e \) where \( e \) corresponds to passing a value along with the failure to control stack.

The Modified K{\text{naf}}\rightarrow Abstract Machine

Stacks as just described:

- Initial e
- Final e \mapsto e
- Transition: \mapsto \rightarrow as given by structural semantics rules.
Dynamic Semantics of Value-Passing Exceptions

Evaluate value to be passed by "raise" clause:

\[ k \cdot \text{raise}[\tau](v) \rightarrow k, \text{raise}[\tau](\cdot) \cdot e \]

Evaluate "raise" clause:

\[ k, \text{raise}[\tau](\cdot) \cdot e \rightarrow k, \cdot e \]

Evaluate "handle" clause:

\[ k, \text{handle}(\cdot, x, e) \rightarrow k, \text{handle}(\cdot, x, e) \cdot e \]

If it achieves a value, return it and drop handler:

\[ k, \text{handle}(\cdot, x, e) \rightarrow k, \cdot e \]

If "handle" clause fails, unwind stack to nearest enclosing handler:

\[ \{ f \text{ handle}(\cdot, x, e) \} \rightarrow k, \cdot e \]

Then invoke pending handler:

\[ k, \text{handle}(\cdot, x, e) \rightarrow k, \cdot e \]

Value-Passing Exceptions

Static semantics:

\[ \frac{\tau \cdot e \cdot \text{exn}}{} \]

\[ \frac{\tau \cdot e_1 \cdot \tau \cdot e_2 \cdot \text{exn}}{} \]

\[ \frac{\tau \cdot e \cdot \text{exn}}{} \]

\[ \frac{\tau \cdot e_1 \cdot \tau \cdot e_2 \cdot \text{exn}}{} \]

Question: how to choose \( \tau \text{exn} \)?

Value-Passing Exceptions

A naive choice: \( \tau \text{exn} = \text{string} \).

\[
\begin{align*}
\text{fun} \quad \text{div} \ (m, 0) &= \text{raise} \ "\text{Division by zero attempted.}" \\
| \quad \text{div} \ (m, n) &= \ldots \text{raise} \ "\text{Arithmetic overflow occurred.}" \\
\end{align*}
\]

But how can the handler distinguish exceptions?

\begin{itemize}
  \item Must parse the string.
  \item Must rely on formatting conventions.
\end{itemize}

Unworkable in practice! (Similar problems for "error numbers").

Value-Passing Exceptions

A more reasonable choice: \( \tau \text{exn} = \text{exc} \).

\[
\begin{align*}
\text{Datatype} \ \text{exc} &= \text{Div} \mid \text{Overflow} \mid \text{Match} \mid \text{Bind} \mid \ldots
\end{align*}
\]

Then we can easily distinguish exceptions using pattern matching:

\[
\begin{align*}
\text{fun} \quad \text{div} \ (m, 0) &= \text{raise} \ \text{Div} \\
| \quad \text{div} \ (m, n) &= \ldots \text{raise} \ \text{Overflow} \\
\text{fun} \quad \text{hdlr} \ \text{Div} &= \ldots \\
| \quad \text{hdlr} \ \text{Overflow} &= \ldots
\end{align*}
\]
Value-Passing Exceptions

This is just a labelled sum type:

\[ \tau_{\text{exn}} = [\text{div}; \text{unit}; \text{fnf}; \text{string}; ...] \]

and the handler code becomes:

\[
\begin{align*}
\text{try} & \; \epsilon_1 \; \text{ow} \; x \Rightarrow \\
& \quad \text{case} \; x \\
& \qquad \text{div} \; () \Rightarrow \epsilon_{\text{div}} \\
& \qquad | \; \text{fnf} \; s \Rightarrow \epsilon_{\text{fnf}} \\
& \qquad | \; \ldots
\end{align*}
\]

First-Class Continuations

The semantics for exceptions (and co-routines) can be expressed using reified control stacks.

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.

Such a reified control stack is a first-class continuation.

Informal Overview

Seize the current continuation: \( \text{letcc} \! [r] \! (x. \epsilon) \).

- Introduction form for \( \text{cost}(r) \).
- Reify the current control stack (current continuation) \( k \).
- Bind \( x \) to \( k \).
- Evaluate \( \epsilon \).

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

Concrete syntax: \( \text{letcc} \! \! x \! \! \epsilon \! \! \text{in} \! \! c \)

Informal Overview

Pass control to a reified continuation: \( \text{throw} \! [r] \! (\epsilon_1 ; \epsilon_2) \).

- Elimination form for \( \text{cost}(r) \).
- Evaluate \( \epsilon_1 \) to a value \( \epsilon_1' \).
- Evaluate \( \epsilon_2 \) to a continuation (stack) \( k \).
- Pass \( \epsilon_1' \) to \( k \).

“Jump” to a given continuation, passing a value.

Concrete syntax: \( \text{throw} \! \! \epsilon_1 \! \! \text{to} \! \! \epsilon_2 \)

Value-Passing Exceptions

Requires that we fix in advance the set of exceptions.

- Non-modular. Makes writing libraries difficult.
- Non-extensible. No user-defined exceptions.

Better: a dynamically extensible sum type.

- Will treat separately later as: dynamic classification and dynamic classes.
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

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:int cont in mult l ret end
```

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

Problem: composing a continuation with a function.

- Given: a function \( f \) of type \( r' \rightarrow r \) and a continuation \( k \) of type \( r' \rightarrow \text{cont} \).
- Return: a continuation \( k' \) of type \( r' \rightarrow \text{cont} \) that, when thrown a value \( v' \) of type \( r' \), throws \( f(v') \) to \( k \).
Example: Composition

How do we obtain that continuation?

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

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

Example: Composition

```haskell
fun compose (f, k) =
  ... throw (f (letcc r:τ' cont in ...)) to k ...
```

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

Example: Composition

Return the continuation using short-circuit return:

```haskell
fun compose (f:τ'→τ, k:τ cont) =
  letcc ret; in
    throw (f (letcc r:τ' cont in throw r to ret)) to k
```

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

Answer: \( τ'\) cont cont

Example: Composition

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

```haskell
fun compose (f, k) =
  ... throw (f (letcc r:τ' cont in ...)) to k ...
```

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

Control stacks are values, but not available as expressions to the programmer.

Extended \( \mathcal{K}(\text{nat} \to \tau) \) Abstract Machine:

Stack Frames and Values

Two new stack frames to record pending computations:

- \( e_2 \) exp
- \( e_1 \) val

Typing rules for the new frames:

- \( \text{throw}[r; e_2 : τ] \) frame
- \( \text{throw}[r; e_1 : \cdot] \) frame

Every reified control stack is a value:

- \( k \) stack
  - \( \text{cont}(k) \) val

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>( τ )</td>
<td>( \text{cont}(r) )</td>
<td>( r ) cont</td>
</tr>
<tr>
<td>Expr</td>
<td>( e )</td>
<td>( \text{letcc}[r; x.e] )</td>
<td>( \text{letcc} ) in ( e )</td>
</tr>
</tbody>
</table>
**Dynamic Semantics**

Specify evaluation order:

\[ k \triangleright \text{throw}(v) \overset{c_1, c_2}{\rightarrow} k \triangleright \text{throw}(v) \triangleright c_1 \triangleright c_2 \]

\( e_1 \text{ val} \)

\( k \triangleright \text{throw}(v) \overset{c_1, \ldots, c_2}{\rightarrow} k \triangleright \text{throw}(v) \triangleright c_1 \triangleright \ldots \triangleright c_2 \)

---

**Example**

Let’s trace the execution of \( e = \text{compose}(F, k) \), where \( F : \tau' \rightarrow \tau \) and \( k : \text{cont}(\tau) \).

\[ k_0 \triangleright e \overset{*}{\rightarrow} k_0 \triangleright \text{letcc}\[ \tau身上 \text{throw}(F(\text{letcc } x \text{ in throw } \tau x \text{ to } r)) \text{ to } k \]

\[ k_0 \triangleright \text{throw}(F(\text{letcc } x \text{ in throw } \tau x \text{ to } k_0)) \text{ to } k \]

\[ k_0 \triangleright \text{throw} - \text{to } k; F(\ldots) \triangleright \text{letcc } x \text{ in throw } \tau x \text{ to } k_0 \]

\[ k' \triangleright \text{throw} k' \text{ to } k_0 \]

\[ k_0 \triangleright k' \]

So the continuation \( k' \) is returned to \( k_0 \), as desired. But is \( k' \) the desired continuation?

---

**Safety**

Well-formed states:

\[ k : \tau \quad e : \tau \]

\[ k : \tau \quad e : \tau \quad e \text{ val} \]

\[ k : \tau \quad e : \text{ok} \quad e \text{ val} \]

\[ k : \tau \quad e : \text{ok} \]

Theorem 3 (Preservation)

If \( s \text{ ok} \) and \( s \rightarrow s' \), then \( s' \text{ ok} \).

---

**Dynamic Semantics**

letcc duplicates control stack:

\[ k \triangleright \text{letcc}\[ \tau身上 \text{throw}(v) \overset{c}{\rightarrow} k \triangleright [\text{cont}(k)/v]c \]

throw abandons current control stack:

\[ k \triangleright \text{throw}(v) \overset{\tau \text{ cont}(k)}{\rightarrow} k' \]

---

**Example**

Let \( F = \text{fun}[\tau']\{x.e\} \).

\[ k_0 \triangleright \text{throw} v \overset{\tau'}{\rightarrow} k' \]

\[ k_0 ; \text{throw} - \text{to } k ; F(\ldots) \overset{\tau'}{\rightarrow} k_0 ; \text{throw} - \text{to } k ; \{F,v/f,x\} e \]

\[ k_0 ; \text{throw} - \text{to } k ; v' \overset{\tau'}{\rightarrow} k ; v' \]

This is the desired behavior!

---

**Proof of Preservation**

Suppose that \( k \triangleright \text{letcc}\[ \tau身上 (x.e) \overset{r}{\rightarrow} k \triangleright [\text{cont}(k)/x]e \]

and that \( k \triangleright \text{letcc}\[ \tau身上 (x.e) \overset{\text{ok}}{\rightarrow} \).

Then there exists \( r \) such that \( k : \tau \) and \( \text{letcc}\[ \tau身上 (x.e) : \tau \).

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

Hence \( [\text{cont}(k)/x]e : \tau \).

Hence \( k \triangleright [\text{cont}(k)/x]e : \text{ok} \).
Proof of Preservation

Suppose that $k;\text{throw}\{e;:-\} < \text{cont}(k') \mapsto k' \triangleright v$ and that $k;\text{throw}\{e;:-\} < \text{cont}(k') \triangleright ok$.

Then there exists $r'$ such that $\text{cont}(k') : \text{cont}(r')$ and $k;\text{throw}\{e;:-\} : \text{cont}(r')$.

Hence $v : r'$ and $k' : r'$.

Hence $k' \triangleright ok$.  

Safety

Lemma 4 (Canonical Forms)

If $v : \text{cont}(r)$ and $v \triangleright \text{val}$, then $v = \text{cont}(k)$ for some control stack $k$ such that $k : r$.

This is easily proved by induction on typing.

Theorem 5 (Progress)

If $s \triangleright \text{ok}$ then either

1. $s \text{ final}$, or
2. there exists $s'$ such that $s \mapsto s'$.

Left as an exercise!

Summary

Continuations are reified control stacks.

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

Continuations are a powerful programming mechanism!

- Can be used to implement co-routines – see Harper
- Can be used to implement exceptions – left as an exercise