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.

Static Semantics of Exceptions

No surprises here:

\[ \Gamma \vdash \text{fail} : \tau \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \]

Failures have any type.

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

Dynamic Semantics of Exceptions

Use the C Machine since it provides access to the control stack.

- Could use the M Machine (exercise).
- Another solution below relies on explicit control stack.

Additional frame:

\[ e_2 \text{ expr} \quad \text{try} \quad \text{frame} \]
Dynamic Semantics of Exceptions

Evaluate "try" clause:

\[ k \triangleright \text{try} e_1 \text{ow} e_2 \text{ow} e_3, k \triangleright e_1 \]

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

\[ \text{try} \text{ow} e_2, k \triangleright \text{v} \quad \text{C} \quad \text{try} \text{ow} e_2, k \triangleright \text{v} \]

Dynamic Semantics of Exceptions

If "try" clause fails, unwind stack to nearest enclosing handler.

\[ (f \neq \text{try} \text{ow} e_2) \]

\[ \text{f} : k \triangleright \text{fail} \quad \text{C} \quad k \triangleright \text{fail} \]

Then invoke pending handler.

\[ \text{try} \text{ow} e_2, k \triangleright \text{fail} \quad \text{C} \quad k \triangleright e_2 \]

Safety for Exceptions

Well-formed machine states:

\[ \vdash k : \tau \text{stack} \quad \vdash e : \tau \]

\[ \text{(k,e) ok} \]

That is, it must be type correct to pass (the value of) e to k.

Frame Typing

\[ \vdash e_2 : \text{int} \]

\[ \vdash \text{apply} (\text{v_2}, \text{e_2}) : (\text{int},\text{int}) \text{ frame} \]

\[ \vdash \text{v_1 value} \quad \vdash \text{v_1 : int} \]

\[ \vdash \text{apply} (\text{v_1}, \text{v_2}) : (\text{int},\text{int}) \text{ frame} \]

Stack Typing

A stack is a composition of frames:

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

\[ \vdash f : (\tau,\tau') \text{ frame} \quad \vdash k : \tau' \text{ stack} \]

\[ \vdash f : k : \tau \text{ stack} \]

Frame Typing

\[ \vdash e_2 : \text{int} \]

\[ \vdash \text{apply} (\text{v_2}, \text{e_2}) : (\text{int},\text{int}) \text{ frame} \]

\[ \vdash \text{v_1 value} \quad \vdash \text{v_1 : int} \]

\[ \vdash \text{apply} (\text{v_1}, \text{v_2}) : (\text{int},\text{int}) \text{ frame} \]
Frame Typing

\[ \vdash e_1 : \tau \quad \vdash e_2 : \tau \quad \vdash \text{if} \, \varnothing \, \text{then} \, e_1 \, \text{else} \, e_2 \, \text{fi} \, \langle \text{bool}, \tau \rangle \text{ frame} \]

\[ \vdash e_2 : \tau \quad \vdash \text{try} \, \varnothing \, \text{ow} \, e_2 \, \langle \tau, \tau \rangle \text{ frame} \]

A More Realistic Semantics

Raising an exception walks the stack to find nearest enclosing handler.

- **Slow**: linear in depth of stack, not depth of handler nesting.
- **Penalizes** code that doesn’t raise exceptions by popping unused exception handlers.
- **Requires** run-time information to detect handler.

We expect a failure to jump directly to the handler and non-failing code to execute normally.

Type Safety

Theorem 1 (Preservation)

If \((k, e)\) ok and \((k, e) \rightarrow_{\top} (k', e')\), then \((k', e')\) ok.

Theorem 2 (Progress)

If \((k, e)\) ok then either

1. \(k = \varepsilon\) and \(e\) value, or
2. \(k = \varepsilon\) and \(e = \text{fail}\), or
3. there exists \(k'\) and \(e'\) such that \((k, e) \rightarrow_{\top} (k', e')\).

A More Realistic Semantics

Informal idea:

- Maintain a register containing **current handler**.
- On failure, jump directly to contents of this register.
- Restore **previous** handler before jump and before normal return.
- Restore control stack before invoking handler.

This semantics requires:

- **A handler stack** to enclose pending handlers. Current handler is top of stack. Stack is modified during execution.
- **A means of freeze-drying** and thawing stacks to save stack appropriate to handler.
The state of the H Machine takes one of two forms:

- an evaluation state $h | k > e$ corresponds to evaluating closed expression $e$ relative to control stack $k$
- a return state $h | k < v$ corresponds to evaluating stack $k$ relative to closed value $v$

The separator points to the focal entry of the state.

Handlers preserve state on handler stack:

$$h | k > try e_1 ow e_2 \rightarrow_H (try □ ow e_2; k); h | try □ ow e_2; k > e_1$$

Notice that the same frame is pushed onto $k$ to form the new top of the handler stack and onto the control stack $k$.

On failure, pop the handler stack:

$$\frac{(try □ ow e_2; k); h | k > fail \rightarrow_H h | k > e_2}{r | k > fail \rightarrow_H r | r > fail}$$

If no handler, stop execution immediately:

$$r | k > fail \rightarrow_H r | r > fail$$

If guarded expression evaluates normally, we must remember to pop the handler stack:

$$\frac{(try □ ow e_2; k); h | try □ ow e_2; k < e_1 \rightarrow_H h | k < e_1}{(try □ ow e_2; k); h | try □ ow e_2; k < e_1 \rightarrow_H h | k < e_1}$$

Is freeze-drying a stack realistic?

- Sounds like a heavyweight operation.
- Can we "hot swap" stacks on a running program?
A More Realistic Semantics

Lemma 3
Let \((h, k, e)\) be a state of the abstract machine. If \(h = (\text{try} \circ \text{ov} \circ k); k', \) then \(k'\) is an initial segment of \(k\). Moreover, the same property holds of \((k', k, e)\).

Thus stored stacks may be represented as “fingers” pointing into the control stack.

- Restore by multi-pop to a given point earlier in the stack.
- cf. setjmp/longjmp in C.

Value-Passing Exceptions

A schematic formulation:

\[
\Gamma \vdash e : \tau_{\text{exc}} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau_{\text{exc}} \rightarrow \tau \\
\Gamma \vdash \text{try} e_1 \circ e_2 : \tau
\]

Question: how to choose \(\tau_{\text{exc}}\)?

Value-Passing Exceptions

A naïve choice: \(\tau_{\text{exc}} = \text{string}\).

\begin{verbatim}
fun div (n, 0) = raise "Division by zero attempted."
| div (n, m) = ... raise "Arithmetic overflow occurred." ...
\end{verbatim}

But how can the handler distinguish exceptions?

- Must parse the string.
- Must rely on formatting conventions.

Unworkable in practice!

Value-Passing Exceptions

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

- Division-by-zero, arithmetic overflow.
- Match and bind failures.
- User-defined failures.

Solution: pass values along with exceptions.

Value-Passing Exceptions

Observation: there must be one choice governing all exceptions.

- Handler cannot tell which exception will be raised.
- Handler usually analyzes value associated with the exception.

Value-Passing Exceptions

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

Datatype \(\text{exc} = \text{Div} \mid \text{Overflow} \mid \text{Match} \mid \text{Bind} \mid \ldots\)

Then we can easily distinguish exceptions using pattern matching:

\begin{verbatim}
fun div (n, 0) = raise Div
| div (n, m) = ... raise Overflow ...
fun hdlr Div = ...
| hdlr Overflow = ...
\end{verbatim}
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.

Exn’s Are Not Exceptions!

The exn type has nothing to do with the exception mechanism!

• Useful on its own as an extensible type.
• So-called exceptions are really value constructs introduced on the fly.
• Exception mechanism passes values of type exn.

Exn’s Are Not Exceptions!

What’s important about exn:

• Dynamically extensible.
• Supports tagging and dispatch.

The Java type Object is very similar to ML’s exn.

Better: a dynamically extensible datatype.

• Called exn in ML.
• Add a new constructor using exception declaration.
• Pattern match to dispatch on exception.

exception Div
exception Overflow
fun div (m, 0) = raise Div
  | div (m, n) = ... raise Overflow ...

Exn’s Are Not Exceptions!