

### 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.

1

### Exceptions

Add a primitive **failure** mechanism to  $\mathcal{L}\{\text{nat} \rightarrow\}$  (PCF):

Category	Item	Abstract	Concrete
Expr	$e$	$::= \text{fail}[\tau]$	<b>fail</b>
		$  \text{catch}(e_1; e_2)$	$\text{try } e_1 \text{ ow } e_2$

Think of **fail** as raising a fixed, anonymous exception.

2

### Static Semantics of Exceptions

No surprises here:

$$\frac{}{\Gamma \vdash \text{fail}[\tau] : \tau} \quad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{catch}(e_1; e_2) : \tau}$$

Failures have **any** type.

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

3

### Dynamic Semantics of Exceptions

Use the  $\mathcal{K}\{\text{nat} \rightarrow\}$  abstract machine since it provides access to the control stack.

An additional form of **state**:

- a **failure** state  $k \blacktriangleleft$  corresponds to passing a failure to control stack  $k$

An additional frame:

$$\frac{e_2 \text{ exp}}{\text{catch}(-; e_2) \text{ frame}}$$

4

### Dynamic Semantics of Exceptions

Evaluate "fail" clause:

$$\overline{k \triangleright \text{fail}[\tau] \mapsto k \blacktriangleleft}$$

Evaluate "catch" clause:

$$\overline{k \triangleright \text{catch}(e_1; e_2) \mapsto k; \text{catch}(-; e_2) \triangleright e_1}$$

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

$$\overline{k; \text{catch}(-; e_2) \triangleleft v \mapsto k \triangleleft v}$$

5

### Dynamic Semantics of Exceptions

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

$$\frac{(f \neq \text{catch}(-; e_2))}{k; f \blacktriangleleft k \blacktriangleleft}$$

Then invoke pending handler.

$$\overline{k; \text{catch}(-; e_2) \blacktriangleleft k \triangleright e_2}$$

6

### The Modified $\mathcal{K}\{\text{nat} \rightarrow\}$ Abstract Machine

States: as just described

- Initial:  $\varepsilon \triangleright e$
- Final:  $\varepsilon \triangleleft e$  with  $e$  val
- Final:  $\varepsilon \blacktriangleleft$

Transitions:  $\mapsto$  as given by structural semantics rules.

7

### Type Safety

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

#### Theorem 1 (Preservation)

If  $s$  ok and  $s \mapsto s'$ , then  $s'$  ok.

#### Theorem 2 (Progress)

If  $s$  ok then either  $s$  final, or there exists  $s'$  such that  $s \mapsto s'$ .

8

### 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.

9

### Value-Passing Exceptions

Category	Item	Abstract	Concrete
Expr	$e$	$::= \text{raise}[\tau](e)$	$\text{raise}[\tau](e)$
		$  \text{handle}(e_1; x.e_2)$	$\text{try } e_1 \text{ ow } x \Rightarrow e_2$

Argument  $e$  to  $\text{raise}[\tau](e)$  is evaluated to determine value passed to handler.

$\text{handle}(e_1; x.e_2)$  binds variable  $x$  in handler  $e_2$  to value passed with exception raised during execution of  $e_1$ .

10

### Dynamic Semantics of Value-Passing Exceptions

An extension to the **failure state**:

- $k \blacktriangleleft e$  where  $e$  val corresponds to passing a value along with the failure to control stack  $k$

Stack frames:

$$\frac{\text{raise}[\tau](-) \text{ frame}}{\text{handle}(-; x.e_2) \text{ frame}}$$

11

### Dynamic Semantics of Value-Passing Exceptions

Evaluate value to be passed by "raise" clause:

$$\frac{}{k \triangleright \text{raise}[\tau](e) \mapsto k; \text{raise}[\tau](-) \triangleright e}$$

Evaluate "raise" clause:

$$\frac{}{k; \text{raise}[\tau](-) \triangleleft e \mapsto k \triangleleft e}$$

$$\frac{}{k; \text{raise}[\tau](-) \blacktriangleleft e \mapsto k \blacktriangleleft e}$$

Evaluate "handle" clause:

$$\frac{}{k \triangleright \text{handle}(e_1; x.e_2) \mapsto k; \text{handle}(-; x.e_2) \triangleright e_1}$$

12

### Dynamic Semantics of Value-Passing Exceptions

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

$$\frac{}{k; \text{handle}(-; x.e_2) \triangleleft e \mapsto k \triangleleft e}$$

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

$$\frac{(f \neq \text{handle}(-; x.e_2))}{k; f \triangleleft e \mapsto k \triangleleft e}$$

Then invoke pending handler:

$$\frac{}{k; \text{handle}(-; x.e_2) \blacktriangleleft e \mapsto k \triangleright [e/x]e_2}$$

13

### Value-Passing Exceptions

Static semantics:

$$\frac{}{\Gamma \vdash \text{raise}[\tau](e) : \tau} \quad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma, x : \tau_{\text{exn}} \vdash e_2 : \tau}{\Gamma \vdash \text{handle}(e_1; x.e_2) : \tau}$$

Question: how to choose  $\tau_{\text{exn}}$ ?

14

### 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.

15

### Value-Passing Exceptions

A naïve choice:  $\tau_{\text{exn}} = \text{string}$ .

```
fun div (m, 0) = raise "Division by zero attempted."
  | div (m, n) = ... raise "Arithmetic overflow occurred." ...
```

But how can the handler distinguish exceptions?

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

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

16

### Value-Passing Exceptions

A more reasonable choice:  $\tau_{\text{exn}} = \text{exc}$ .

```
Datatype exc = Div | Overflow | Match | Bind | ...
```

Then we can easily distinguish exceptions using pattern matching:

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

17

## Value-Passing Exceptions

This is just a labelled sum type:

$$\tau_{\text{exn}} = [\text{div}; \text{unit}, \text{fnf}:\text{string}, \dots]$$

and the handler code becomes:

```
try e1 ow x =>
  case x
    div {} => ediv
  | fnf s => efnf
  | ...
```

18

## 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**

19

## 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**

20

## Informal Overview

Introduce a type `cont( $\tau$ )` of continuations.

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

Concrete syntax:  $\tau \text{ cont}$

21

## Informal Overview

Seize the **current continuation**: `letcc $[\tau]$ ( $x.e$ )`.

- Introduction form for `cont( $\tau$ )`.
- Reify the current control stack (current continuation)  $k$ .
- Bind  $x$  to  $k$ .
- Evaluate  $e$ .

Grab the current control point (continuation) for use elsewhere.  
Concrete syntax: `letcc  $x$  in  $e$`

22

## Informal Overview

Pass control to a **reified** continuation: `throw $[\tau]$ ( $e_1; e_2$ )`.

- Elimination form for `cont( $\tau$ )`.
- 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.  
Concrete syntax: `throw  $e_1$  to  $e_2$`

23

### 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!

24

### 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:int cont in
    let fun mult nil = 1
        | mult (0::_) = throw 0 to ret
        | mult (n::_) = n * mult l
    in mult l end
```

25

### 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::_) ret = n * mult l ret
  in letcc ret:int cont in (mult l) ret end
```

26

### Example: Composition

Problem: composing a continuation with a function.

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

27

### 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.

28

### Example: Composition

The continuation we want: `throw  $f(-)$  to  $k$` .

If we fill the hole with  $v'$ , then

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

29

### Example: Composition

How do we obtain that continuation?

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

We want the continuation at the argument to  $f$ .

30

### Example: Composition

Idea: seize the continuation using `letcc`:

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

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

31

### Example: Composition

Return the continuation using short-circuit return:

```
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:  $\tau' \text{ cont}$ .

32

### Continuations

Extend  $\mathcal{L}_{\{\text{nat} \rightarrow\}}$  (PCF) with continuation types and corresponding expressions:

Category	Item	Abstract	Concrete
Type	$\tau$	$::= \text{cont}(\tau)$	$\tau \text{ cont}$
Expr	$e$	$::= \text{letcc}[\tau](x.e)$   $\text{throw}[\tau](e_1; e_2)$   $\text{cont}(k)$	$\text{letcc } x \text{ in } e$ $\text{throw } e_1 \text{ to } e_2$

Note: `letcc` binds  $x$  in  $e$ .

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

33

### Static Semantics

Typing rules:

$$\frac{\Gamma, x:\text{cont}(\tau) \vdash e:\tau}{\Gamma \vdash \text{letcc}[\tau](x.e):\tau} \quad \frac{\Gamma \vdash e_1:\tau_1 \quad \Gamma \vdash e_2:\text{cont}(\tau_1)}{\Gamma \vdash \text{throw}[\tau'](e_1; e_2):\tau'}$$

$$\frac{k:\tau}{\Gamma \vdash \text{cont}(k):\text{cont}(\tau)}$$

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

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

Stack  $k$  accepting values of type  $\tau$  is a continuation value  $\text{cont}(k)$  of type  $\text{cont}(\tau)$

34

### Extended $\mathcal{K}_{\{\text{nat} \rightarrow\}}$ Abstract Machine: Stack Frames and Values

Two new **stack frames** to record pending computations:

$$\frac{e_2 \text{ exp}}{\text{throw}[\tau](-; e_2) \text{ frame}} \quad \frac{e_1 \text{ val}}{\text{throw}[\tau](e_1; -) \text{ frame}}$$

Typing rules for the new frames:

$$\frac{e_2:\text{cont}(\tau)}{\text{throw}[\tau](-; e_2):\tau \Rightarrow \tau'} \quad \frac{e_1:\tau \quad e_1 \text{ val}}{\text{throw}[\tau](e_1; -):\text{cont}(\tau) \Rightarrow \tau'}$$

Every reified control stack is a value:

$$\frac{k \text{ stack}}{\text{cont}(k) \text{ val}}$$

35

## Dynamic Semantics

Specify evaluation order:

$$\frac{}{k \triangleright \text{throw}[\tau](e_1; e_2) \mapsto k; \text{throw}[\tau](-; e_2) \triangleright e_1}$$

$$\frac{e_1 \text{ val}}{k; \text{throw}[\tau](-; e_2) \triangleleft e_1 \mapsto k; \text{throw}[\tau](e_1; -) \triangleright e_2}$$

36

### Example

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

$$\begin{aligned} k_0 \triangleright e &\mapsto^* k_0 \triangleright \text{letcc } r \text{ in } \text{throw } (F(\text{letcc } x \text{ in } \text{throw } x \text{ to } r)) \text{ to } k \\ &\mapsto^* k_0 \triangleright \text{throw } (F(\text{letcc } x \text{ in } \text{throw } x \text{ to } k_0)) \text{ to } k \\ &\mapsto^* \underbrace{k_0; \text{throw } - \text{ to } k; F(-)}_{k'} \triangleright \text{letcc } x \text{ in } \text{throw } x \text{ to } k_0 \\ &\mapsto^* k' \triangleright \text{throw } k' \text{ to } k_0 \\ &\mapsto^* k_0 \triangleleft k' \end{aligned}$$

So the continuation  $k'$  is returned to  $k_0$ , as desired. But is  $k'$  the desired continuation?

38

### Safety

Well-formed states:

$$\frac{k : \tau \quad e : \tau}{k \triangleright e \text{ ok}}$$

$$\frac{k : \tau \quad e : \tau \quad e \text{ val}}{k \triangleleft e \text{ ok}}$$

#### Theorem 3 (Preservation)

If  $s \text{ ok}$  and  $s \mapsto s'$ , then  $s' \text{ ok}$ .

40

## Dynamic Semantics

letcc duplicates control stack:

$$\frac{}{k \triangleright \text{letcc}[\tau](x.e) \mapsto k \triangleright [\text{cont}(k)/x]e}$$

throw abandons current control stack:

$$\frac{}{k; \text{throw}[\tau](v; -) \triangleleft \text{cont}(k') \mapsto k' \triangleleft v}$$

37

### Example

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

$$\begin{aligned} k_0 \triangleright \text{throw } v \text{ to } k' &\mapsto k' \triangleleft v \\ &= k_0; \text{throw } - \text{ to } k; F(-) \triangleleft v \\ &\mapsto k_0; \text{throw } - \text{ to } k \triangleleft [F, v/f, x]e \\ &\mapsto^* k_0; \text{throw } - \text{ to } k \triangleleft v' \\ &\mapsto k \triangleleft v' \end{aligned}$$

This is the desired behavior!

39

### Proof of Preservation

Suppose that  $k \triangleright \text{letcc}[\tau](x.e) \mapsto k \triangleright [\text{cont}(k)/x]e$  and that  $k \triangleright \text{letcc}[\tau](x.e) \text{ ok}$ .

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

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

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

Hence  $k \triangleright [\text{cont}(k)/x]e \text{ ok}$ .

41

### Proof of Preservation

Suppose that  $k; \text{throw}[\tau](v; -) \triangleleft \text{cont}(k') \mapsto k' \triangleleft v$  and that  $k; \text{throw}[\tau](v; -) \triangleleft \text{cont}(k')$  ok.

Then there exists  $\tau'$  such that  $\text{cont}(k') : \text{cont}(\tau')$  and  $k; \text{throw}[\tau](v; -) : \text{cont}(\tau')$ .

Hence  $v : \tau'$  and  $k' : \tau'$ .

Hence  $k' \triangleleft v$  ok.

42

### Safety

#### Lemma 4 (Canonical Forms)

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

This is easily proved by induction on typing.

#### Theorem 5 (Progress)

If  $s$  ok then either

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

Left as an exercise!

43

### Summary

Continuations are **reified** control stacks.

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

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

44