Compiler Construction

Lecture 13: exceptions

Jeremy Yallop

jeremy.yallop@cl.cam.ac.uk

Lent 2026

Exceptions

Exception-handling constructs

Raising exceptions

```
raise e
```

Evaluate `e` to value `v`

then raise `v` as an *exceptional value* which can only be handled.

Handling exceptions

try e_1 with x $\rightarrow$ e_2

If e_1 evaluates to value `v`
then `v` is the result of the entire expression.

Otherwise, an exceptional value `w` is raised
in the evaluation of e_1, and `w` is *handled*:

i.e. e_2 is evaluated with `w` bound to `x`
and becomes the result of the entire expression.

Exceptions in OCaml

Exception types

```
(* extensible type *)
type exn = ...

(* add constructor *)
type exn +=
   E of string
```

Raising

```
raise (E "...")
```

Catching

```
try e with
| E₁ x → e₁
| E₂ x → e₂
...
| Eₙ x → eₙ
```

desugar

```
try e with
| v → (match v with
       | E₁ x → e₁
       | E₂ x → e₂
       ...
       | Eₙ x → eₙ
       | _ → raise v)
```

Implementing exceptions

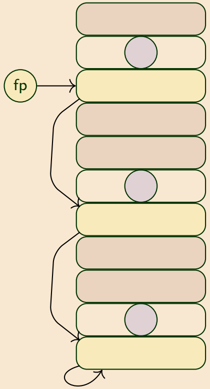
Stack searching

`raise` **transfers control to the most-recent handler**.

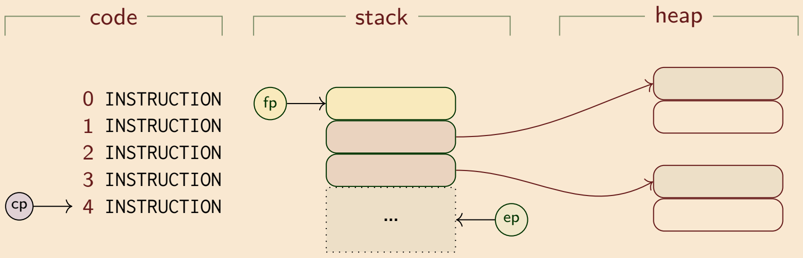
Handlers behave like a stack:
  entering `try` pushes to the stack
  exiting `try` pops from the stack

We *could* use the `fp` chain to search for the handler

Instead: **remember the position of the handler**



New VM state: exception pointer



cp code pointer (to next instruction)

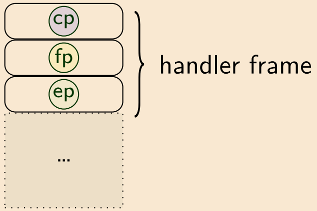
fp frame pointer (to current activation frame)

ep exception pointer (to current handler frame)

not shown: stack pointer, heap limit

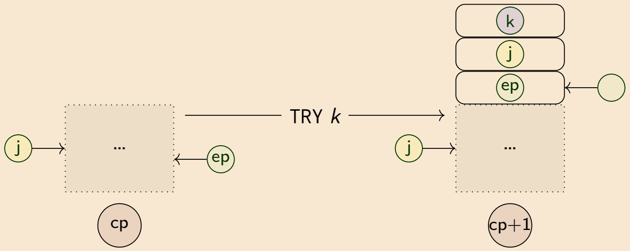
Handler frames

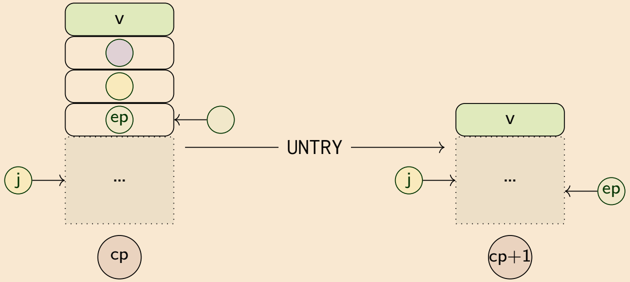
The exception pointer points to a **handler frame** with the information `raise` needs:



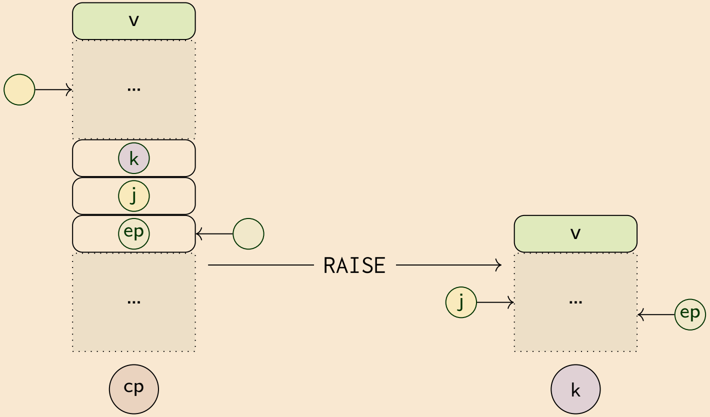
- cp code address for `raise` to jump to
- fp saved frame pointer for `raise` to restore
- ep saved exception pointer for `raise` to restore

New instructions: TRY/UNTRY

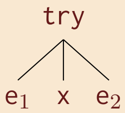




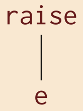
New instructions: RAISE



Compilation scheme



```
     TRY k
     <code for e₁>
     UNTRY
     GOTO m
k:   <code for (λx.e₂)>
     APPLY
m:
```



```
     <code for e>
     RAISE
```

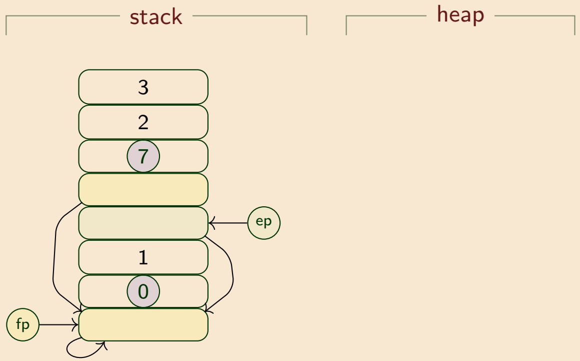
Tracing execution: try

code

```
 0 PUSH STACK_INT 1
 1 TRY L0 = 7
 2 PUSH STACK_INT 2
 3 PUSH STACK_INT 3
 4 OPER ADD
 5 UNTRY
 6 GOTO L1 = 10
 7 LABEL L0
 8 MK_CLOSURE(L2 = 13, 0)
 9 APPLY
10 LABEL L1
11 OPER ADD
12 HALT
13 LABEL L2
14 LOOKUP STACK_LOCATION -2
15 PUSH STACK_INT 10
16 OPER ADD
17 RETURN
```

cp → 4

```
1 + try 2 + 3
    with e → e + 10
```

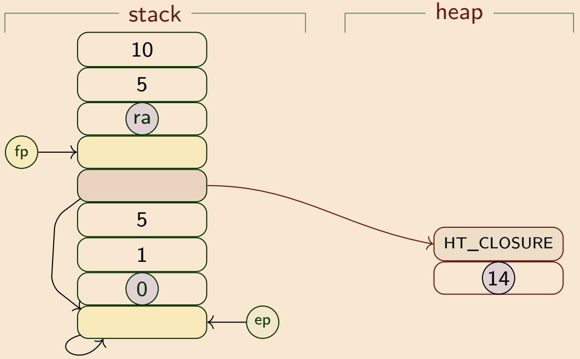


Tracing execution: try + raise

code

```
   0 PUSH STACK_INT 1
   1 TRY L0 = 8
   2 PUSH STACK_INT 2
   3 PUSH STACK_INT 5
   4 RAISE
   5 OPER ADD
   6 UNTRY
   7 GOTO L1 = 11
   8 LABEL L0
   9 MK_CLOSURE(L2 = 14, 0)
  10 APPLY
  11 LABEL L1
  12 OPER ADD
  13 HALT
  14 LABEL L2
  15 LOOKUP STACK_LOCATION -2
  16 PUSH STACK_INT 10
cp → 17 OPER ADD
  18 RETURN
```

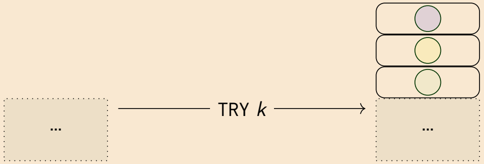
```
1 + try 2 + raise 5
    with e → e + 10
```



Exception pragmatics

Exceptions and tail calls

Since handler frames use stack space, a **call inside a handler is not a tail call**



For example, the following function is **not tail recursive**

```
let rec all_except f = function
  | [] → true
  | x :: xs → try f x && all_except f xs with Not_found → false
```

but can be made tail-recursive by moving the recursive call outside the handler:

```
let rec all_except2 f = function
  | [] → true
  | x :: xs → (try f x with Not_found → false) && all_except2 f xs
```

Exceptions and destructors

In C++, raising an exception deallocates stack-allocated objects.

Deallocation executes the code of each object's *destructor*

```
struct C { ~C() { cout << "Goodbye\n"; } /* destructor */ };

void g() { throw runtime_error("No resources\n"); }

void f() { C c; g(); }

int main() {
  try { f(); }
  catch (const runtime_error& e) { cout << e.what(); }
}
```

For example, the example above prints:

```
Goodbye
No resources
```

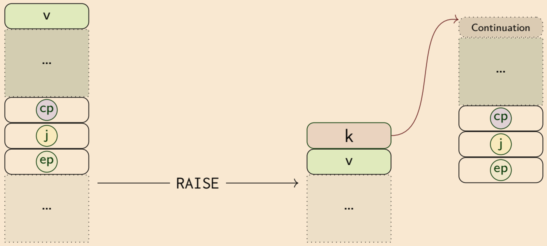
Jumping directly to the handler is not valid: throw must **unwind the stack**

Resumable exceptions

Idea: don't discard the stack on `raise`

Option 1: handle the exception before discarding the stack
(the program chooses: discard the stack / continue)

Option 2: make the stack available to the program
(the program chooses: discard stack / continue / save stack & restore later)



Next time: optimisation