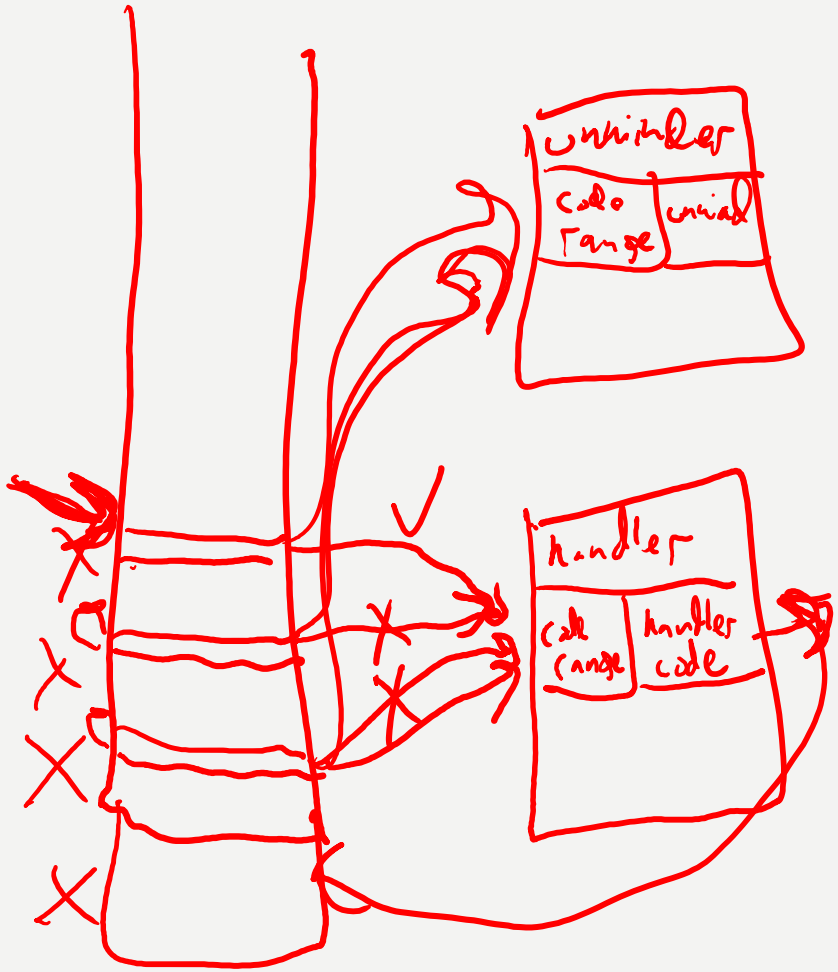
STACK AND CONTROL PRIMITIVES

ROSS TATE

KEY PEOPLE

- Arjun Guha
- Andrew Myers
- Heejin Ahn
- Thomas Lively
- Matthew Flatt

EXCEPTION HANDLING (CONCEPTUALLY)



1. Stack Walk
2. Stack Tag
3. Invoke Tag
4. Redirect Control
5. Stack Unwind

STACK MARKING (FOR GC WITHOUT REFS)

- declare mark gcrooter : [] -> []
- void foo() {

```
    i32 newref1 = my_gc_alloc(12);
    i32 newref2 = my_gc_alloc(24);
    stack.mark gcrooter() {
        my_add_reachable(newref1);
        my_add_reachable(newref2);
    } within {
        … // do stuff with this mark on stack
    }
}
```

- little to no run-time overhead
  - using code-address ranges
- mark is executable
  - in general, has params and results
  - not a first-class reference/value
  - no heap allocation or mem. management

STACK WALKING (FOR GC WITHOUT REFS)

```
void add_roots_on_stack() {
    stack.walk {
        while (true) [] {
            stack.next-mark gcrooter {
                stack.exec-mark();
            } none {
                break;
            } []
        }
    }
}
```

- stack.walk starts a stack walk
  - tracks a pointer *into* stack
  - initialized to bottom of stack
- stack.next-mark walks up stack
  - must be within stack.walk
  - looks for a matching mark, e.g. gcrooter
  - uses "none" upon reaching top of stack
- stack.exec-mark executes current mark
  - must be within stack.next-mark
  - param and result types from declaration

STACK TRACING

- declare mark coderef : [] -> [string, i32]
  // file name, line #
- mark coderef() {
  result("Foo.lang", 55); // not an instruct.
  } within {
  … // code that happens on that line
  }
- trace get_stack_trace() {
  trace t = empty_trace;
  stack.walk {
  while (true) [] {
  stack.next-mark coderef {
  t = append(t, exec-mark()…);
  } none { return t; } []
  } } }

- coderef mark produces results
- here coderef is used to get stack trace
  - traces refer to source code
  - rather than wasm code
- could also be used by a debugger
- no new stack primitives for traces

APPLICATIONS (WITHOUT REDIRECTING CONTROL)

Garbage collection

Stack tracing

Dynamic scoping

Debuggers

Stack-allocated closures

- Generators
- C# out/ref parameters
- Scala's lazy parameters
- Optimized higher-order programming

REDIRECTING CONTROL (ESCAPE HATCH - INTRO)

- escape $hatch {
  … // run the body
  escape-to(val*) $hatch; // redirect
  … // more body
  } hatch(t_h*) {
  … // execute with val if escaped-to
  } [t_o*] // output type of both body & hatch

- hatch is only entered if escaped to
  - val* must have type t_h*
- similar to block, loop, and if
  - encodable using block (for now)
  - but no need for input types for main body
- but what about stack unwinding?

STACK UNWINDING (ESCAPE HATCH – FULL)

- mark unwinder : [] -> [];
- escape $hatch {
  … // run the body
  escape-to(val*) $hatch; // redirect
  … // more body
  } unwind unwinder { // [t_h*] -> [t_h*]
  stack.exec-mark();
  } hatch(t_h*) {
  … // execute after unwinding
  } [t_o*] // output type of both body & hatch

- add unwinder marks for unwinding code
  - e.g. destructors, finally, .NET's abort
  - just a convention
- unwind clause(s)
  - specifies a mark
  - ran on matching marks as stack unwinds
  - can update the t_h* values for the hatch
  - can have multiple clauses for diff marks
- hatch executed after unwinding

EXCEPTION HANDLING

1. Stack Walk
2. Stack Tag
3. Invoke Tag
4. Redirect Control
5. Stack Unwind

- Erlang
- C++
- C#
- Python

ERLANG

- try Body of _ -> IfNoThrow
  catch throw:Thrown -> Handler
- throw(thrown)
  - stack.walk {
    while (true) {
    stack.next_mark throw_catcher {
    stack.exec-mark(e);
    } // no none! trap/debug at top
    }
    }

- mark throw_catcher : [eref] -> []
- escape $hatch {
  mark throw_catcher(eref Thrown) {
  escape-to(Thrown) $hatch;
  } within {
  Body
  }
  IfNoThrow // executed only if no throw
  } hatch(eref Thrown) { // no unwind!
  Handler
  }

C++

- try {
  … // body
  } catch (MyException err) {
  … // handler
  }
- throw e;
  - function cpp_throw(i32 e) {
    stack.walk {
    while (true) {
    stack.next_mark cpp_catcher {
    stack.exec-mark(e);
    } } } }
- throw;
  - stack.walk {
    stack.next_mark cpp_handler {
    cpp_throw(stack.exec-mark());
    } }

- mark cpp_destructor : [] -> [];
  mark cpp_catcher : [i32] -> [];
  mark cpp_handler : [] -> [i32];
- escape $hatch {
  stack.mark cpp_catcher(i32 e) {
  if (rtti_of(i32) == MyException)
  escape-to(e) $hatch;
  } within {
  … // body
  }
  } unwind cpp_destructor {
  stack.exec-mark();
  } hatch(i32 e) {
  mark cpp_handler() {
  respond(e);
  } within {
  … // handler
  }
  }

PYTHON

- try:
  … // body
  except:
  … // handler
  finally:
  … // unwinder
- mark py_finally : [] -> [];
  mark py_except : [pyexc pyref pytrace] -> [];
  mark py_handle : [] -> [pyexc pyref pytrace];
  mark py_code_ref : [] -> [pycoderef];
- sys.exc_info(); // get caught exception
  - function sys_exc_info() {
    stack.walk {
    stack.next_mark py_handle {
    return stack.exec-mark();
    } } }
- raise;
  - py_throw_with_trace(sys_exc_info())

- escape $hatch {
  mark py_finally() {
  … // unwinder
  } within {
  mark py_except(pyexc e, pyref v, pytrace t) {
  escape-to(e, v, t) $hatch;
  } within {
  … // body
  }
  … // unwinder
  }
  } unwind(pyexc e, pyref v, pytrace t) py_finally {
  stack.exec-mark();
  } unwind(pyexc e, pyref v, pytrace t) py_code_ref {
  t := add_to_trace(t, stack.exec-mark());
  } hatch(pyexc e, pyref v, pytrace t) {
  mark py_handle() {
  respond(e, v, t);
  } within {
  … // handler
  }
  }