Discussion questions for *Implementing Zero-Overhead Exception Handling*

N. Ramsey and D. Nunez, for COMP 250RTS

September 27, 2017

The exception model

We'll examine a simple exception model suitable for imperative code with side effects. (Raising an exception is a side effect, so if you're imagining a pure language, you need also to imagine something like an I/O monad.) Rather than use the model sketched on pages 1–3, we use the TRY-EXCEPT form from Modula-2 and Modula-2+, as descrbed on pages 3, 4, and 5:

```
TRY
  ... the "body", which is "in the scope of"
      the handler, and is executed for side
      effect ...
EXCEPT E =>  (* E names an exception *)
  ... the "handler", executed for side effect ...
END
```

Notes:

- In the language of Drew, Gough, and Ledermann, the handler is "syntactically associated" with the body. But it's not tied to a procedure.
- If you prefer an expression-oriented language like Scheme, ML, Icon, or Haskell, imagine both the TRY block and the handler as expressions that have values.
- For a nice alternative syntax, see Nick Benton and Andrew Kennedy, Exceptional Syntax. *Journal of Functional Programming*, 11(4):395–410, July 2001.
- We can make the model more complicated by adding any or all of the following features:
  - Handler that is associated with multiple exceptions
  - Multiple handlers per TRY block
  - Handler that takes control of "all other" exceptions
  - TRY-FINALLY, containing finalization code which executes on any terminating outcome

  But the simple model is good enough to expose most of the implementation issues.

Informally, the operational semantics are as described on page 2 (paraphrased):

> When control enters the `TRY` block, the handler becomes the **current handler** and *raising* exception `E` (and possibly any other exception) causes control to transfer to that handler.
>
> The handler remains current during the execution of the body, except when execution enters another TRY block which has its own handler. This may occur when TRY blocks are nested or when the body (transitively) calls another procedure in which execution enters a TRY block.
>
> When control (normally) exits the body of a TRY block, the previously current handler becomes current once again.

We could formalize this semantics, but instead we'll talk about a direct implementation of this idea, which amounts to the same thing. If time permits.

The meaning of "zero overhead"

Drew, Gough, and Ledermann write,

> We would like to entirely eliminate any overhead involved in making particular handlers current, and then removing them.

The phrase "zero overhead" has spilled a lot of ink. Here are two interpretations:

A. Changing the current handler cannot be charged to any machine instruction. Or in other words, no instructions are emitted whose purpose is to change the current handler. (The *Performance* section on page 8 could have said, "during program executions which do not raise exceptions, [the system] does not execute a single additional instruction." It's not clear to me what they mean by "additional." Compared to what?)

B. Assuming no exception is ever raised, the dynamic cost of execution is exactly what it would be in a program without TRY-EXCEPT blocks.

Pages 9 and 10 make it clear that Drew, Gough, and Ledermann intend interpretation A, not interpretation B. Many others have criticized the phrase "zero overhead" on the grounds that interpretation B cannot actually be achieved.

Questions for class

Tracking the current handler

These questions are about zero-overhead exception handling:

(1) During execution, what part or parts of the machine state represents the current handler?

(2) What machine instruction or instructions change the relevant parts of the machine state?

(3) Argue that maintaining the current handler costs nothing. (Hint: "free" means "already paid for.")

Constant-time exceptions

Suppose we want to be able to raise an exception in constant time, and in order to achieve such a blessed outcome, we are willing to pay a small run-time overhead to enter or exit the scope of an exception handler. Assume that an exception is raised by calling a `Raise` procedure that is implemented in the run-time system.

(4) Translate `TRY` S_1 `EXCEPT E =>` S_2 `END` to low-level IR

(5) Sketch an implementation of `Raise`.

(6) What about callee-saves registers?

Cost of `Raise` (zero-overhead style)

The section "Unwinding the stack," starting on page 4, says a little bit about the cost of raising an exception in the zero-overhead world:

- The stack unwinder restores the state of every activation.
- Raising an exception may unwind the stack, which may require restoration of callee-saves registers.

(7) In the zero-overhead model, what is the worst-case bound on the number of callee-saves registers restored? Can the bound be improved?

Extra questions, for which there is no time

Cooperation between compiler and runtime

In the zero-overhead world, assume that all exceptions are allocated at link time.

(8) Building on your work from last class (stack walking), and also on the pseudocode from page 7, explain how `Raise` is implemented. Use abstractions and functions from last class as needed. Identify any additional abstractions you want from this paper.

(9) How are the run-time abstractions in the previous question supported by the compiler? Or to put it another way, what must the compiler and the runtime agree on?

In the paper, some of the compiler support is emitted as an instruction sequence called the "dummy epilogue" (top of page 6).

(10) Is there any capability *not* supported in the dummy epilogue, which therefore requires a table?

(11) On page 3, the last paragraph of the short section on CLU says that the implementation started off with compiler cleverness (vectored return) but ended up using a table. In this paper, which design do you prefer, a dummy epilogue or "encoding state-restoration actions in a compact table form"?

(12) In what other situations do you imagine that a system designer has the opportunity to shift the implementation of functionality between the compiler and the run-time system?

Bonus questions to take home

Applicability

(13) Could this implementation be used in a higher-order, functional language like ML or Haskell?

The performance penalty

On page 9, the Discussion section presents an example in which local variables `p` and `l` may be flushed to the stack, even if no exception is raised. The reason given is that

> The presence of exception handling introduces control flow (even interprocedural control flow) which is not represented in the control-flow graph.

This seems odd.

(14) Why isn't the control flow represented in the control-flow graph, and what would you recommend to do about it?

(15) Do constant-time exceptions present the same issue? If so, what would you recommend to do about it?

Compilation

For students with compiler experience:

(16) Consult the data directives of your favorite assembly language, and figure out how a compiler builds the relevant tables and what bread crumbs it drops into the instruction stream. You might like to revisit the chatter about the gardens point IR at the bottom of page 5, and also the discussion of "Creating the address map" on pages 7 and 8.

Introspecting the call stack

On page 6, in the section "Encoding the state-restoration actions," the authors note that if no handler is found, the run-time system must produce an authentic core dump showing the state of the stack *at the point where the unhandled exception was raised*. For that reason, the stack is unwound using dummy registers.

In a language like Java, a programmer can ask for the procedure stack leading up to the exception.

(17) What do you think the language model should look like?

(18) Can it be implemented using the techniques described by Drew, Gough, an Ledermann? If so, how?

Noted without comment

On page 5, "Most of our back ends do not waste a register as a frame pointer, even on the Intel architecture."