CS412/413

Introduction to Compilers
Radu Rugina

Lecture 34: Exception Handling
23 Apr 04

Exceptions

- Many languages allow exceptions: alternate return paths from a function
  - null pointer, overflow, emptyStack,...
- Function either terminates normally or with an exception
  - total functions ⇒ robust software
  - no encoding error conditions in result
- Several different exception models: effect on implementation efficiency

Generating Exceptions

- Java, C++: statement throw E is statement that terminates exceptionally with exception E
- Exception propagates lexically within current function to nearest enclosing try..catch statement containing it (exception handler)
- Handlers may re-throw exceptions
- If not caught within function, propagates dynamically upward in call chain.
- Tricky to implement dynamic exceptions efficiently

Declaration of Exceptions

- Must a function declare all exceptions it can throw?
- § Implementer convenience: annoying to declare all exceptions (overflow, null pointers,…)
- § vs. Client robustness: want to know all exceptions that can be generated

- Java: must declare "non-error" exceptions
- ML: cannot declare exceptions at all (good for quick hacking, bad for reliable software)
- C++: declaration is optional (useless to user, compiler)

Naming Exceptions

- Java, C++: exceptions are objects
  - name of exception is name of object's class
  - exceptional return distinguished from normal return

```
Exception m() throws Exception {
if (c) throw new Exception();
else  return new Exception(); }
```

- ML: exceptions are special names with associated data

```
Exception OutOfRange of int * int
… raise OutOfRange(n,m)
```

- Ada: exceptions are simple tags

```
SomethingWrong : exception;
raise SomethingWrong;
```


Desired Properties

- Exceptions are for unusual situations and should not slow down common case:
  1. No performance cost when function returns normally
  2. Little cost for executing a try..catch block—when exception is not thrown.
  3. Cost of throwing and catching an exception may be somewhat more expensive than normal termination
- Not easy to find such an implementation!

Lexical Exception Throws

- Some exceptions can be turned into goto statements; can identify lexically

```
try {
    if (b) throw new Foo();
    else x = y;
} catch (Foo f) { … }

⇒      if (b) { f = new Foo(); goto l1; }
       x = y; goto l2;
       l1: { … }
       l2:
```


Dynamic Exception Throws

- Cannot always statically determine the exception handlers…
- Need to dynamically find closest enclosing try..catch that catches the particular exception being thrown
- No generally accepted technique! (see absence of discussion in Appel, Dragon Book)

Impl. 1: Extra Return Value

- Return an extra (hidden) boolean from every function indicating whether function returned normally or not

```
throw e       ⇒  return (true, e)
return e      ⇒  return (false, e)
a = f(b, c)   ⇒  (exc, t1) = f(b,c);
                 if (exc) goto handle_exc_34;
                 a = t1;
```

- No overhead for try..catch blocks
- Simple run-time mechanism: just need return (true, e), a check, and a jump to statically determined handler
- Can express as source-to-source translation
- Drawback = function call overhead: every function call requires extra parameter, extra check

Impl. 2: setjmp/longjmp

- setjmp(buf) saves all regs + stack state into a buffer, returns 0
- longjmp(buf) restores state in buf, makes setjmp "return 1"
- Implementation: CatchStack *stk;

try S catch C

```
{ CatchInfo current;
  stk->push(current);
  if (!setjmp(current->buf))
        S
  else  C;
  stk->pop();}
```

throw e

```
CatchInfo *current = top(stk);
while (!handles(current,e))
        current = stk->pop();
current->data = e;
longjmp(current->buf);
```


setjmp/longjmp Summary

- Advantages:
  - Easy to implement, portable
  - No overhead as long as try/catch, throw unused
- Disadvantages:
  - Is not thread-safe (stk must be thread-specific)
  - Setjmp/longjmp turn off inter-procedural optimizations and optimizations of heap variables
  - There is overhead executing try/catch, try/catch/finally even if no exception is thrown
  - May need to walk up through several enclosing try..catch blocks until right one is found

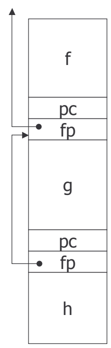
Impl. 3: PC-Based Techniques

- Idea: map PC values to exception handlers!
- Need to map PC values at throw statements and call sites
- Approach one: place markers in the code (implicit mapping)

```
call foo
.long handlerinfo
add $4, %esp    #normal post-call code
```

  - Extra info after each call about handlers
  - Throw statements are also calls (to run-time exception dispatcher routines)
  - If routine not found, walk up stack one frame at a time (fp known)
  - In each frame, check table for matching handlers (PC known because return address is pushed on stack)

Example




```
f() {
    try g()
        catch A => S1
        catch B => S2
}

g() throws A, B {
    try h()
        catch B => S3
}

h() throws A,B  {
    throw A
}
```


PC-Based Techniques, Part2

- Drawback of code markers: return from calls must skip the inserted info after the call
- Alternative approach: use explicit tables which map PC addresses to handlers
  - Either use hash tables
  - Or map ranges of PC addresses
  - To find a handler: lookup current PC for matching entry
  - Entry contains info about the kind of exception handled and the actual handler address
  - Also need to unwind the stack if no matching handlers
  - Need to set up PC map tables

PC-Based Techniques

- Advantages:
  - no cost for try/catch: tables created statically
  - no extra cost for function call
  - throw → catch is reasonably fast (one table lookup per stack frame, can be cached)
- Disadvantages:
  - can't implement as source-to-source translation
  - must restore callee-save registers during walk up stack (can use symbol table info to find them)
  - table lookup/stack unwinding more complex if using Java/C++ exception model (need dynamic type discrimination mechanism, finalization code in Java, destructors in C++)

Summary

- Several different exception implementations commonly used
- Extra return value, setjmp/longjmp impose overheads but can be implemented in C
- PC-based techniques (using static exception tables) have no overhead except on throw, but require back end compiler support