**Exceptions**

- Simple model of a function or method: takes in set of arguments, returns value
- Many languages also allow exceptions: alternate return paths from a function
  - null pointer, overflow, emptyStack,...
- Function either terminates normally or with an exception
  - total functions make robust software
  - no encoding error conditions in result
- Several different exception models: affects 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
  - If not caught within function, propagates dynamically upward in call chain.
  - Tricky to implement efficiently

**Implicit vs. explicit re-throw**

- Implicitly vs. explicitly re-thrown: does an exception automatically propagate out of a function?
- Issue: convenience vs. “no surprises”
- Java, C++, ML: yes; CLU: no (converts to special implicitly-thrown failure exception)

```
f() throws Exc = (....throw Exc,)
g() throws Exc = (   try
  f()
  )
  catch (Exc) throw Exc;
```

**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
  - CLU: must declare all but failure (but uncaught exceptions automatically converted failure)
  - 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 class name
  - exceptional return distinguished from normal return even w/ same type
  ```java
  Exception m() throws Exception {
    throw new Exception();
  }
  ```
- ML, CLU: exceptions are special names with associated data: disjoint
  ```java
  exception badness(int);
  void m() throws badness {
    throw badness(4);
  }
  ```

Desired Properties

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

Static exception throws

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

- Need to find closest enclosing try..catch dynamically that catches the particular exception being thrown
- No generally accepted technique!
  (See Appel, Muchnick, Dragon Book for absence of discussion)

Impl. #1: extra return value

- Return an extra (hidden) boolean from every function indicating whether function returned normally or not
  ```java
  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;
  ```
- Every function call requires extra parameter, extra check
- No cost for try..catch unless exception thrown. Goto labels determined statically.
- Can express as source-to-source translation

#2: setjmp/longjmp (orig. Java)

- `setjmp(buf)` saves all registers into a buffer buf (incl. sp, pc!), returns 0
- `longjmp(e)` restores all registers from buffer e; places 1 into return register.
  ```java
  throw(e) => exc = e;
  longjmp(current_catch);
  ```
  try S catch C => push_catch();
  if (setjmp(current_catch) == 0) S
  else C;
  pop_catch();
### setjmp/longjmp summary

- **Advantages:**
  - no cost as long as `try/catch, throw` unused
  - works even without declared exceptions: no static information needed
- **Disadvantages:**
  - `try/catch, try/catch/finally` are slow even if no exception is thrown
  - May need to walk up through several `longjumps` until right `try..catch` is found.
  - `current_catch` must be thread-specific

### Exceptions as continuations

- Goal of exception handling mechanism is to map an exception to its continuation
- Extra boolean: pass only one continuation, returned boolean & exception value resolved into continuation in caller’s code
- `setjmp/longjmp`: two continuations passed: normal and exceptional
  - Thread-specific global variable is optimization of extra argument; resolving of exceptional continuations done the slow way.

### #3: Tables of continuations

- Extra boolean: walk up stack frame by frame
- `setjmp/longjmp`: walk up one try/catch at a time
- Would like to be able to jump up the stack to the right place immediately
- **Problem:** need precise continuation info to do this; `try..catch` must update a **continuation table**
  
  - `CT`: (exception → (pc, fp))
  - `throw e1` /
    - `call-continuation(CT[e1])`
  - `try S catch(e1) S1...catch(e n) S n`
    - `CTsave = CT; CT = CT[e1]; S; goto L; L1: CT=CTsave; S1; goto L`
  - `L : Expensive!`

### #4: Static Exception Tables

- Invented for CLU by Bob Scheifler
- **Observation:** exceptions that are caught usually go up only one or two stack frames; more important to find right exception handler (pc) than stack frame (fp)
- **Throw code:**
  - walk up stack one frame at a time (fp known)
  - in each frame, use return address to select table
  - table maps exception to right pc
- **Table is static → no cost for try/catch!**
**Static Exception Tables**

- **Advantages:**
  - no cost for try/catch: tables created by compiler
  - no extra cost for function call
  - throw → catch is reasonably fast (one table lookup per stack frame, can be cached)

- **Disadvantages:**
  - table lookup more complex if using Java/C++ exception model (need dynamic type discrimination mechanism)
  - 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)

**Summary**

- Several different exception implementations commonly used
- Extra return value, `setjmp/longjmp` impose overheads but can be implemented in C (hence used by C++, Java)
- Static exception tables have no overhead except on throw, but require control of compiler back end.