Exception Handling

Exception Handling Considerations

- What constitutes an exception?
  - Domain, Range errors
  - Cristian:

    Omission: output not there

    Timing: Too early / too late

    Response: wrong output

    Crash: no response
- When an exception is raised, how far can it propagate?
- What chain does an exception follow?
  - static
  - dynamic

Exception Handling Considerations

- What to do with exception raiser
  - Resume for certain NOTIFY
  - Resume at handler's option SIGNAL
  - Terminate ESCAPE
- Should exceptions have parameters?
- Should passed over routines be allowed to clean-up?
- How should exceptions be handled in parallel environment?

CLU & Exception Handling

- What is an exception? User defined / failure
- Propagation? To caller only (except failure)
- Path searched for handler? Dynamic chain
- What to do with raiser? Terminate
- Parameters? Yes
- Declare exceptions thrown? Yes
- Clean-up? N/A (no by default)
- How handled in parallel? N/A

Ada & Exception Handling

• What is an exception?	User defined / 4.5 system defined
• Propagation?	To caller
• Path searched for handler?	Dynamic chain
• What to do with raiser?	Terminate
• Parameters?	No
• Declare exceptions thrown?	No
• Clean-up?	N/A (no by default)
• How handled in parallel environment?	Not propagated

Yemini & Berry: Exception Handling

• What is an exception?	User defined
• Propagation?	To caller
• Path searched for handler?	Dynamic chain
• What to do with raiser?	User choice (replacement model)
• Parameters?	Yes
• Declare exceptions thrown?	Yes
• Clean-up?	Yes
• How handled in parallel environment?	N/A

C++ & Exception Handling

• What is an exception?	User defined
• Propagation?	Thrown to catcher
• Path searched for handler?	Dynamic chain
• What to do with raiser?	Terminate
• Parameters?	Yes
• Declare exceptions thrown?	No
• Clean-up?	Yes
• How handled in parallel environment?	N/A

Java & Exception Handling

• What is an exception?	User defined
• Propagation?	Thrown to catcher
• Path searched for handler?	Dynamic chain
• What to do with raiser?	Terminate
• Parameters?	Yes
• Declare exceptions thrown?	Yes
• Clean-up?	Yes
• How handled in parallel environment?	N/A

Raising Exceptions: CLU and Ada

Explicit Raising:

- CLU: signal
  IF x < 0 THEN SIGNAL neg(X) ← Termination in both cases
- Ada: raise
  IF x < 0 THEN RAISE neg ← Termination in both cases

Implicit Raising:

- CLU: systems failures, failure to catch
- Ada: four (Ada95; five in Ada83) predefined failures
  - tasking, program, storage, constraint, numeric (X'd in Ada95)
  - Ada supports exception raising during elaboration

Raising Exceptions: C++ and Java

Explicit Raising:

- Throw / catch

Implicit Raising:

- C++: runtime_error, range_error, overflow_error, underflow_error, bad_alloc, bad_cast…
  - All built off base class exception
- Java: built-in hierarchy with base throwable:
  - extensive set of exceptions: AbstractMethodError, InternalError, UnknownError, InterruptedException, EmptyStackException, IOException…
    - floats don't throw exceptions!

Sample CLU Function

```
sign = proc(x:int) returns (int) signals (zero, neg (int) )
        if x < 0 then signal neg(x)
           elseif x = 0 then signal zero
           else return (x)
        end
end sign
```

Note Java-like requirement to name exceptions that can be thrown. Of course, CLU had the requirement first.

Sample Ada Function

```
Package STACK is
   ERROR: exception;
   procedure push(x: integer);
   function pop return integer;
end STACK;
package body STACK is
   ….
   function POP return integer is
      begin
        if top = 0 then  raise ERROR
        end if;
        top := top - 1;
        return s(top + 1);
   end POP
end STACK;
```

Handling Exceptions: CLU

- CLU: statement level

```
a:= sign(x)
        EXCEPT WHEN neg(i: int):  // handle
a:= sign(x) + sign(y)
       EXCEPT WHEN neg(i: int):   // Who raised?
BEGIN
s1;  EXCEPT WHEN ...  EXIT done (...)
s2;
 ...
END
 EXCEPT WHEN excp1(...)  H1;
                done (...)  H2;
```

* Can raise another exception in same procedure using Exit *

Handling Exceptions: Ada

- Ada: (frame level)

```
BEGIN
        s1;
        s2;
 EXCEPTION
        WHEN EXCP1 => H1 {raise}
        WHEN EXCP2 => H2
END
```

- ...elaboration level:

```
DECLARE
        <elaborated stuff>  // exceptions handled by invoker
BEGIN
        s1; ...      // exceptions handled in frame
EXCEPTION
        WHEN EXCP1 => H1
END
```

Propagating Exceptions: CLU and Ada

CLU:

- to *invoker* through *explicit* means
- Except for failure exception, propagation is explicit *only*.
- Procedure specifications include ID's of signalled exceptions
  - coupling between called procedures and all potential callers?

Ada:

- to end of frame (statements)
- to invoker (elaborations)
- Exceptions propagate up dynamic call chain (by default) until caught
  - interesting interaction with static scoping

    name passed out of scope and back in again

CLU Failure Initiation

```
nonzero = proc(x: int) returns(int)
   return (sign(x))
      except when  neg(y:int): return(y)
      end
end nonzero
```

"zero" exception goes uncaught

Failure("unhandled exception: zero") gets propagated

Ada Propagation

```
package D is
   procedure A;
   procedure B;
end;
procedure OUTSIDE is
begin
   D.A;
end OUTSIDE;
package body D is
   ERROR: exception;
   procedure A is
      begin
      ... raise ERROR ...
   end A;
   procedure B is
      begin
      OUTSIDE;
   exception
      when ERROR => . . .
   end B;
end D;
```

Call to D.B can create interesting situation

CLU's Failure Exceptions

- Only automatically propagated exception is failure
  - raised if no handler for raised, named exception
  - can be explicitly raised
  - occurs if unanticipated failure occurs

... thoughts about overloading of "failure"?

... thoughts about propagating failure rather than name (Ada)

Parameters

CLU: Yes

– See Yemini and Berry for argument in favor

Ada: No (not even Ada95)

– result is Ada can require access to non-locals to straighten things out
– could lead to erroneous programming (determining parameter passing mechanism)

- C++: Yes
- Java: Yes

Problem with No Parameters

```
PROC P (inout param1, param2, ...);
BEGIN
. . .
  EXCEPTION
  WHEN excp1 => ...
END
```

Possible messing with non-locals to determine if acceptable values have been set

Level of Application

CLU, Java:

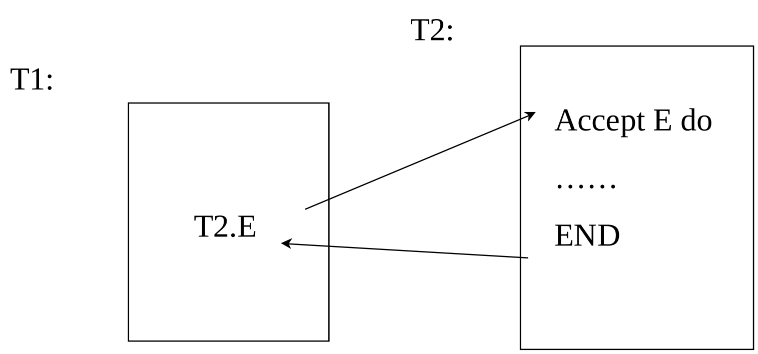
– statement level

Ada, C++:

– x:= (a + b) * c

Because of operator overloading, functions that redefine operations can do their own repairs, having an in-expression effect .

Ada and Tasking Errors



Ada and Tasking Errors

- Exception raised inside accept, no handler
  - raised after accept in T2 (Ada.95 allows handler inside Accept)
  - raised after entry call in T1
- T2 aborted -> tasking error raised in T1
- T2 non-existent (completed) -> tasking error raised in T1
- T1 aborted -> T2 completes normally
- T1, T2 doesn't handle exception -> not propagated!

Exceptions and Compiler Optimization

- Normally, programmer-defined order of events must be followed unless program's effects unchanged
  - e.g. of OK switch:

    A:= B + C;

    D:= E / F;

Exceptions and Compiler Optimization

- When considering exceptions, new problems arise, e.g. code movement:

```
term:= 0;
FOR j IN 1..10 LOOP
   term:= term + j ** a(k);
   x:= x + 1;
```

Code movement changes program meaning if exception raised inside loop

Yemini and Berry

- Five handler responses:
  - *Resume the signaller:* Do something, then resume the operation where it left off
  - *Terminate the signaller:* Do something, then return a substitute result of the required type for the signalling operation; if operation is not value returning, just proceed after operation invocation.
  - *Retry the signaller:* Do something and then invoke the signaller again.

Yemini and Berry

- Five handler responses (continued):
  - *Propagate the exception:* Do something, then allow the invoker of the invoker of the signalling operation to respond to the detection of the exception.
  - *Transfer control:* Do something, then transfer control to another location in the program. This includes doing something and then terminating a closed construct containing the invocation.

Stroustrup on Resumption vs Termination

- Resumption advantages:
  - More general (powerful, includes termination)
  - Unifies similar concepts/implementations
  - Essential for very complex, very dynamic systems (e.g. OS/2)
  - Not significantly more complex/expensive to implement
    - (Contradicts himself on this one: "…resumption requires the key mechanisms for continuations and nested functions without providing the benefits…")
    - If you don't have it you must fake it
  - Provides simple solutions for resource exhaustion problems

--from "Design and Evolution of C++", p.391

Stroustrup on Resumption vs Termination

- Termination advantages:
  - Simpler, cleaner, cheaper
  - Leads to more manageable systems
  - Powerful enough for everything
  - Avoids horrendous coding tricks
  - Significant negative experience with resumption

--from “Design and Evolution of C++”, p.391

Exception Hierarchies

```
class Matherr { };
class Overflow: public Matherr { };
class Underflow: public Matherr { };
class Zerodivide: public Matherr { };
//…
void g { }
{
    try {
          f();
        }
    catch (Overflow) { } // handle overflow, derived exceptions
    catch (Matherr)  { } // handle any Matherr that’s not overflow
}
```