Continuation and Exceptions

Control Flow In Sequential Languages

Imperative Programming
Control Flow of Programs

- Structured control flow
  - Sequence of statements
    - { a:= b; b := c; }
  - Conditional
    - if (a < b) then c else d;
    - switch(a){…}
  - Loops
    - for (…) {…};
    - while (…) {…};
  - Jumping out of a block
    - break, continue, return,…
- Non-structured control flow
  - Goto, conditional jump
  - Used to implement structured control flow in assembly

Controlling Jumps

- Structured jumps

  if … then … else … end

  while … do … end

  for … { … }

  case …

  - Group code in logical blocks
  - Avoid uncontrolled jumps, e.g., *into* the middle of a block
- Focus of this chapter: quickly jumping into and out of a program in an organized fashion
  - Jumping right into the mid of a block ---- continuation passing
    - The scenario: my task was interrupted, now I want to resume from where I stopped
  - Jumping out from the mid of a block? ---- exception handling
    - The scenario: something unexpected happened; need to jump out until some caller knows what to do with the errors.

Continuations

- Capture the continuation at some point to be used later
  - **A function (closure) that takes a single parameter, the result of the past evaluation, and returns the result of the entire program.**
  - Save the entire runtime environment as a closure
    - Code pointer: where to start evaluating the instructions
    - Environment pointer: the entire relevant memory stores
  - To jump into the mid of a program, make a function call to the continuation
- Useful in
  - Implementing functional programming languages
  - Operating system scheduling, Web site design
    - The scenario: my task was interrupted, now I want to resume from where I stopped

Continuation of Expressions

- Continuation: impose sequential ordering in sub-expressions
  - The continuation of an expression is "the remaining work to be done after evaluating the expression"
  - Continuation of *e* is a function applied to the result of *e*
- Enforce evaluation order in functional languages
  - Evaluate current expression
  - Save the result into a variable
  - Evaluate the rest of the computation

2*x + 3*y + 1/x + 2/y

```
let  val r2x = 2 * x in
  let  val r3y = 3 * y in
     let val sum1=r2x + r3y in
       let val r1x = 1 / x in
         let val sum2 = sum1 + r1x in
            let val r2y = 2 / y in
               sum2 + r2y
            end
…….end
```

let r2x = 2*x in … end
is equivalent to
(fn r2x=> …) (2 * x)

Continuation of 2*x

Continuation and Tail Calls

- *A function call from g to f is a tail call*
  - *if g returns the result of calling f with no further computation*
  - Example (red: tail call; blue: non-tail call)

    fun f(x) = if x > 0 then x else f(x+1)*2

    fun f(x,y) = if x>y then x else f(2*x,y);
- Tail calls do not need to return to caller
  - Can we convert all functions to tail recursion?
    - If a program needs to be re-enterable, function calls shouldn't return to caller
  - Solution: continuation passing
    - Pass continuation as parameter to callee
    - Callee does not need to return to caller

Continuation Passing

- Standard function

```
fun fact(n) = if n=0 then 1 else
    n*fact(n-1)
```

- Continuation form

```
fun fact(n, K) =
   if n=0 then K(1)
    else fact(n-1,fn x=>K(n*x));
```

fact(n, fn x=>x) computes n!

- Example computation

```
fact(3,fn x=>x)=
fact(2,fn y=>((fn x=>x)(3*y))) =
fact(1, fn x=>((fn y=>3*y)(2*x)))
    = fn x=>((fn y=>3*y)(2*x)) 1
    = 6
```

- For each function definition F
  - Extend the definition with a continuation parameter K
  - At each function call inside F
    - Convert the rest of computation into a new continuation function
    - Convert f into a tail call, which takes the new continuation function as an extra argument.
  - At each normal return
    - Return the result of invoking continuation K with the original returned value

General uses of continuations

- Explicit control
  - Normal termination -- call continuation
  - Abnormal termination -- do something else
- Compilation techniques
  - Call to continuation is functional form of "go to"
    - Jump to the middle of a block by saving the environment in the function closure and restore the environment before jump
- Web applications, Web Services, MOM and SOA services
  - Handle long running workflows
    - Workflow may take 1 year to complete
  - Progress of subtasks is asynchronous
    - Sequential programming is simpler than asynchronous
- Continuations provide
  - An easy way to suspend workflow execution at a wait state
  - Thread of control can be resumed when the next message/event occurs, maybe some long time ahead

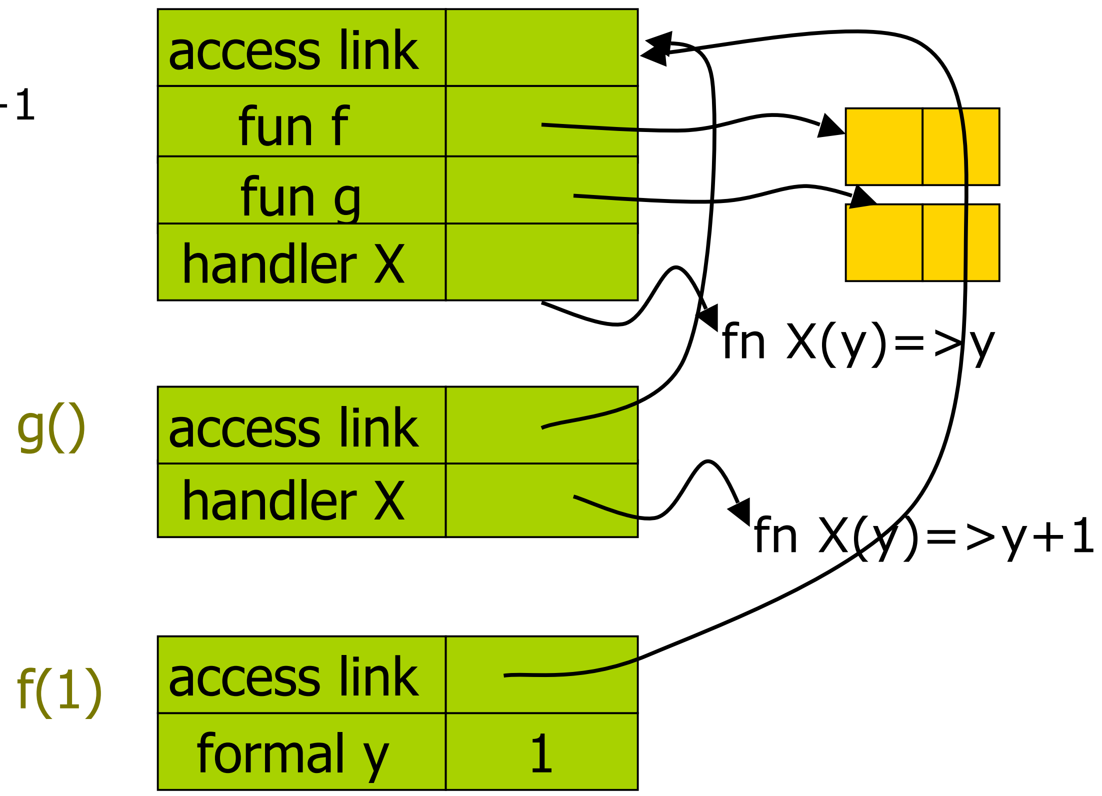
Exception: Structured Exit

- When something unusual happens, we want a program to
  - Jump out of one or many levels of nested blocks
  - Until reaching some program point to continue
  - Pass information to the continuation point
  - May need to free heap space, other resources
- An exception is a dynamic jump
  - Don’t know where to resume execution until runtime
    - Jump out of current block
    - Look for a matching exception handler in most recently entered blocks
  - General dynamic scoping rule
    - Multiple functions could handle the same exception
    - Jump to most recently established handler on run-time stack
    - Callers know how to handler error, defining block doesn’t

Dynamic Scoping of Handler

```
exception X of int;
let fun f(y) = (raise X(y); 1);
    fun g() = f(1)
              handle X(y) => y+1
in  g() handle X(y) => y
end;
```

Dynamic scoping: find first X handler by going up the dynamic call chain



When Should We Use Exceptions?

- Separation of concern: handle unusual situations
  - Examples: division by zero, null pointers, unexpected inputs
  - When exceptions are handled, error recovery
  - Otherwise, evaluation aborts on error conditions
- Flexible control flow
  - Return immediately to where the error can be handled
  - Jump out multiple blocks at a time
- What languages have exception support?
  - C++, Java, ML, Ada, …

Defining Exceptions

- Exception declaration
  - Type of data that can be passed in exception
    - ML: exception <name> of <type>
    - C++/Java: any data type
- Raising an exception
  - Abort the rest of current block and jump out
    - ML: raise <name> <arguments>;
    - C++: throw <value>;
- Handling an exception
  - Continue normal execution after exception
    - ML: <exp1> handle <pattern>=><exp2>; ...
    - C++: try { …} catch (<type> var) {…} …

Exceptions vs. Type System

- Are exceptions part of the type system?

- Raising expressions: not part of the type system
  - Expression e has type t if normal termination of e produces value of type t
  - Raising exception is not normal termination
    - Example: 1 + raise X is not valid
- Handling exceptions (=> ⟨value⟩)
  - Converts exception to normal termination
  - Need type agreement

    1 + ((raise X) handle X => e) Type of e must be int

    1 + (e_1 handle X => e_2) Type of e_1, e_2 must be int

How Are Exceptions Handled?

- ML

```
exception X of int;
let fun f(y) = (raise X(y); 1); fun g() = f(1) handle X(y) => y+1
in  g() handle X(y) => y
end;
```

- What are the events that have occurred?
  - Enter the let expression
  - Make function call g()
  - Make function call f(1)
  - Function call f(1) raises exception X(1)
  - Exception X(1) is handled in function call g()
  - Function call g() returns with value 2
  - The let expression exits

Exception vs. Continuation

- Continuation
  - Explicitly represent the rest of computation
  - Do not need to return to the caller
    - Can use exception to avoid returning to the caller
- Raising exception
  - Jumping out of multiple blocks at a time
  - Different continuation for normal and exceptional situations
    - Continuation of exception
      - rest of computation after exception is handled
- Raising exceptions may have complications
  - Resource management: opened files, garbage collection
  - Use continuation passing to implement exception
    - Pass multiple continuations: one to handle normal condition, the others to handle exceptions