CS 242

Control in Sequential Languages

John Mitchell

Topics

- Structured Programming
  - Go to considered harmful
- Exceptions
  - “structured” jumps that may return a value
  - dynamic scoping of exception handler
- Continuations
  - Function representing the rest of the program
  - Generalized form of tail recursion
- Control of evaluation order (force and delay)
  - May not cover in lecture. Book section straightforward.

Fortran Control Structure

```
10 IF (X .GT. 0.000001) GO TO 20
11 X = -X
   IF (X .LT. 0.000001) GO TO 50
20 IF (X*Y .LT. 0.00001) GO TO 30
   X = X-Y-Y
30  X = X+Y
   ...
50 CONTINUE
   X = A
   Y = B-A
   GO TO 11
   ...
```

Similar structure may occur in assembly code

Historical Debate

- Dijkstra, Go To Statement Considered Harmful
  - Letter to Editor, *C ACM*, March 1968
  - Link on CS242 web site
- Knuth, Structured Prog. with go to Statements
  - You can use goto, but do so in structured way …
- Continued discussion
  - Welch, “GOTO (Considered Harmful)^n, n is Odd”
- General questions
  - Do syntactic rules force good programming style?
  - Can they help?

Advance in Computer Science

- Standard constructs that structure jumps
  - if … then … else … end
  - while … do … end
  - for … { … }
  - case …
- Modern style
  - Group code in logical blocks
  - Avoid explicit jumps except for function return
  - Cannot jump *into* middle of block or function body

Exceptions: Structured Exit

- Terminate part of computation
  - Jump out of construct
  - Pass data as part of jump
  - Return to most recent site set up to handle exception
  - Unnecessary activation records may be deallocated
    - May need to free heap space, other resources
- Two main language constructs
  - Declaration to establish exception *handler*
  - Statement or expression to *raise* or *throw* exception

Often used for unusual or exceptional condition, but not necessarily

ML Example

```
exception Determinant;  (* declare exception name *)
fun invert (M) =          (* function to invert matrix *)
        …
        if …
              then raise Determinant    (* exit if Det=0 *)
              else …
 end;
…
invert (myMatrix) handle Determinant => … ;
```

Value for expression if determinant of myMatrix is 0

C++ Example

```
Matrix invert(Matrix m) {
   if … throw Determinant;
   …
};

try { … invert(myMatrix); …
}
catch (Determinant) { …
   // recover from error
}
```

C++ vs ML Exceptions

- C++ exceptions
  - Can throw any type
  - Stroustrup: "I prefer to define types with no other purpose than exception handling. This minimizes confusion about their purpose. In particular, I never use a built-in type, such as int, as an exception." -- The C++ Programming Language, 3rd ed.
- ML exceptions
  - Exceptions are a different kind of entity than types.
  - Declare exceptions before use

Similar, but ML requires the recommended C++ style.

ML Exceptions

- Declaration
  - exception ⟨name⟩ of ⟨type⟩
    - gives name of exception and type of data passed when raised
- Raise
  - raise ⟨name⟩ ⟨parameters⟩
    - expression form to raise and exception and pass data
- Handler
  - ⟨exp1⟩ handle ⟨pattern⟩ => ⟨exp2⟩
    - evaluate first expression
    - if exception that matches pattern is raised,
      - then evaluate second expression instead

General form allows multiple patterns.

Which handler is used?

```
exception Ovflw;
fun reciprocal(x) =
    if x<min then raise Ovflw else 1/x;
(reciprocal(x) handle Ovflw=>0) / (reciprocal(y) handle Ovflw=>1);
```

- Dynamic scoping of handlers
  - First call handles exception one way
  - Second call handles exception another
  - General dynamic scoping rule
    - Jump to most recently established handler on run-time stack
- Dynamic scoping is not an accident
  - User knows how to handler error
  - Author of library function does not

Exception for Error Condition

```
- datatype 'a tree = LF of 'a | ND of ('a tree)*('a tree)
- exception No_Subtree;
- fun lsub (LF x) = raise No_Subtree
  |   lsub (ND(x,y)) = x;
> val lsub = fn : 'a tree -> 'a tree
```

- This function raises an exception when there is no reasonable value to return
- We'll look at typing later.

Exception for Efficiency

- Function to multiply values of tree leaves

```
fun prod(LF x) = x
|   prod(ND(x,y)) = prod(x) * prod(y);
```

- Optimize using exception

```
fun prod(tree) =
    let exception Zero
        fun p(LF x) = if x=0 then (raise Zero) else x
        |   p(ND(x,y)) = p(x) * p(y)
    in
        p(tree) handle Zero=>0
    end;
```

Dynamic Scope of Handler

```
exception X;
(let fun f(y) = raise X
     and g(h) = h(1) handle X => 2
in
     g(f) handle X => 4
end) handle X => 6;
```

scope

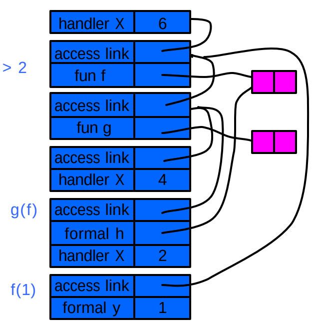
handler

Which handler is used?

Dynamic Scope of Handler

```
exception X;
(let fun f(y) = raise X
     and g(h) = h(1) handle X => 2
in
     g(f) handle X => 4
end) handle X => 6;
```

Dynamic scope:
find first X handler, going up the dynamic call chain leading to raise X.



Compare to static scope of variables

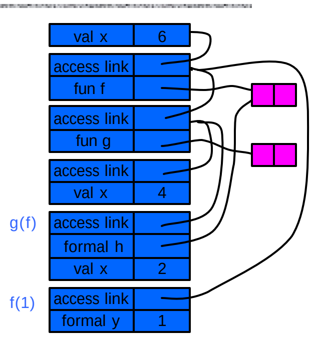
```
exception X;
(let fun f(y) = raise X
     and g(h) = h(1)
                handle X => 2
in
     g(f) handle X => 4
end) handle X => 6;
```

```
val x=6;
(let fun f(y) = x
     and g(h) = let val x=2 in
                   h(1)
 in
     let val x=4 in g(f)
end);
```

Static Scope of Declarations

```
val x=6;
(let fun f(y) = x
     and g(h) = let val x=2 in
                   h(1)
 in
     let val x=4 in g(f)
end);
```

Static scope: find first x, following access links from the reference to X.



Typing of Exceptions

- Typing of raise ⟨exn⟩
  - Recall definition of typing
    - 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
- Typing of handle ⟨exn⟩ => ⟨value⟩
  - Converts exception to normal termination
  - Need type agreement
  - Examples
    - 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

Exceptions and Resource Allocation

```
exception X;
(let
   val x = ref [1,2,3]
 in
   let
      val y = ref [4,5,6]
   in
      ... raise X
   end
end);  handle X => ...
```

- Resources may be allocated between handler and raise
- May be "garbage" after exception
- Examples
  - Memory
  - Lock on database
  - Threads
  - ...

General problem: no obvious solution

Continuations

- General technique using higher-order functions
  - Allows "jump" or "exit" by function call
- Used in compiler optimization
  - Make control flow of program explicit
- General transformation to "tail recursive form"
- Idea:
  - The continuation of an expression is "the remaining work to be done after evaluating the expression"
  - Continuation of *e* is a function applied to *e*

Example of Continuation Concept

- Expression
  - 2*x + 3*y + 1/x + 2/y
- What is continuation of 1/x?
  - Remaining computation after division

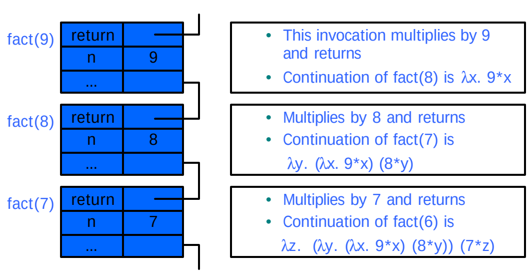
```
let val before = 2*x + 3*y
    fun continue(d) = before + d + 2/y
in
    continue (1/x)
end
```

Example: Tail Recursive Factorial

- Standard recursive function
  fact(n) = if n=0 then 1 else n*fact(n-1)
- Tail recursive
  f(n,k) = if n=0 then k else f(n-1, n*k)
  fact(n) = f(n,1)
- How could we derive this?
  - Transform to continuation-passing form
  - Optimize continuation functions to single integer

Continuation view of factorial

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



Derivation of tail recursive form

- Standard function
  fact(n) = if n=0 then 1 else n*fact(n-1)
- Continuation form
  fact(n, k) = if n=0 then k(1)
                else fact(n-1, λx.k (n*x))   (continuation)

  fact(n, λx.x) computes n!
- Example computation

  fact(3, λx.x) = fact(2, λy.((λx.x) (3*y)))
  = fact(1, λx.((λy.3*y)(2*x)))
  = λx.((λy.3*y)(2*x)) 1 = 6

Tail Recursive Form

- Optimization of continuations

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

Each continuation is effectively λx.(a*x) for some a

- Example computation

```
fact(3,1) = fact(2, 3)     was  fact(2, λy.3*y)
          = fact(1, 6)     was  fact(1, λx.6*x)
          = 6
```

Other uses for continuations

- Explicit control
  - Normal termination -- call continuation
  - Abnormal termination -- do something else
- Compilation techniques
  - Call to continuation is functional form of "go to"
  - Continuation-passing style makes control flow explicit

MacQueen: "Callcc is the closest thing to a 'come-from' statement I've ever seen."

Theme Song: Charlie on the MTA

- Let me tell you the story
  Of a man named Charlie
  On a tragic and fateful day
  He put ten cents in his pocket,
  Kissed his wife and family
  Went to ride on the MTA
- Charlie handed in his dime
  At the Kendall Square Station
  And he changed for Jamaica Plain
  When he got there the conductor told him,
  "One more nickel."
  Charlie could not get off that train.
- Chorus:
  Did he ever return,
  No he never returned
  And his fate is still unlearn'd
  He may ride forever
  'neath the streets of Boston
  He's the man who never returned.

Capturing Current Continuation

- Language feature (use open SMLofNJ; on Leland)
  - callcc : *call* a function with *current continuation*
  - Can be used to abort subcomputation and go on
- Examples
  - callcc (fn k => 1);

  > val it = 1 : int
    - Current continuation is "fn x => print x"
    - Continuation is not used in expression.
  - 1 + callcc(fn k => 5 + throw k 2);

  > val it = 3 : int
    - Current continuation is "fn x => print 1+x"
    - Subexpression throw k 2 applies continuation to 2

More with callcc

- Example

```
1 + callcc(fn k1=>   ...
        callcc(fn k2 => ...
              if ... then (throw k1 0)
                    else (throw k2 "stuck")
        ))
```

- Intuition
  - Callcc lets you mark a point in program that you can return to
  - Throw lets you jump to that point and continue from there

Example

- Pass two continuations and choose one

```
fun f(x,k1,k2) = 3 + (if x>0 then throw k1(x)
                               else throw k2(x));
fun g(y,k1) = 2 + callcc(fn k2 => f(y,k1,k2));
fun h(z) = 1 + callcc(fn k1 => g(z+1,k1));

h(1);
h(~2);
```

Answers: h(1) ⇒ 3   h(~2) ⇒ 2

Continuations in Mach OS

- OS kernel schedules multiple threads
  - Each thread may have a separate stack
  - Stack a blocked thread is stored within the kernel
- Mach "continuation" approach
  - Blocked thread represented as
    - Pointer to a continuation function, list of arguments
    - Stack is discarded when thread blocks
  - Programming implications
    - Sys call such as msg_recv can block
    - Kernel code calls msg_recv with continuation passed as arg
  - Advantage/Disadvantage
    - Saves a lot of space, need to write "continuation" functions

Continuations in compilation

- SML continuation-based compiler [Appel, Steele]
  1) Lexical analysis, parsing, type checking
  2) Translation to λ-calculus form
  3) Conversion to continuation-passing style (CPS)
  4) Optimization of CPS
  5) Closure conversion – eliminate free variables
  6) Elimination of nested scopes
  7) Register spilling – no expression with >n free vars
  8) Generation of target assembly language program
  9) Assembly to produce target-machine program

Coroutines

- Homework Problem 8

```
datatype tree = leaf of int | node of tree*tree;

datatype coA = A of (int* coB) cont   (* searchA wants int and  B-cont*)
and      coB = B of      coA  cont;   (* searchB wants an A-continuation *)

fun resumeA(x, A k) = callcc(fn k' => throw k (x, B k'));
fun resumeB(   B k) = callcc(fn k' => throw k (A k'));
exception DISAGREE; exception DONE;

fun searchA(leaf(x),(y, other: coB)) =
        if x=y then resumeB(other) else raise DISAGREE
|   searchA(node(t1,t2), other) = searchA(t2, searchA(t1, other));

fun searchB(leaf(x), other : coA) = resumeA(x,other)
|   searchB(node(t1,t2), other) = searchB(t2, searchB(t1, other));

fun startB(t: tree)  =  callcc(fn k => (searchB(t, A k); raise DONE));
fun compare(t1,t2) = searchA(t1, startB(t2));
```

Summary

- Structured Programming
  - Go to considered harmful
- Exceptions
  - "structured" jumps that may return a value
  - dynamic scoping of exception handler
- Continuations
  - Function representing the rest of the program
  - Generalized form of tail recursion
  - Used in Lisp, ML compilation