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
12 IF (X LT. 0.00001) GO TO 30
20 IF (X*Y LT. 0.00001) GO TO 30
21 X = X*Y
30 X = X-Y
31 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, CACM, 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 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

```ml
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 => 0;
```

Value for expression if determinant of myMatrix is 0

### C++ Example

```cpp
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?

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

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

### 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

- Dynamic scoping is not an accident
  - User knows how to handler error
  - Author of library function does not
## Exception for Efficiency

- Function to multiply values of tree leaves
  \[ \text{fun } \text{prod}(\text{LF } x) = x \]
  \[ \text{prod}(\text{ND}(x, y)) = \text{prod}(x) \times \text{prod}(y); \]
- Optimize using exception
  \[ \text{fun } \text{prod}(\text{tree}) = \]
  \[
  \begin{align*}
  &\text{let exception Zero} \\
  &\text{fun } p(\text{LF } x) = \text{if } x=0 \text{ then } (\text{raise Zero}) \text{ else } x \\
  &\text{in} \\
  &p(\text{tree}) \text{ handle Zero=>$0$} \\
  &\text{end;}
  \end{align*}
\]

## Dynamic Scope of Handler

- **Dynamic Scope of Handler**
  - exception \( X; \)
    - (let fun \( f(y) = \text{raise } X \) and \( g(h) = h(1) \) handle \( X => 2 \))
      - \( \text{in} \)
    - \( g(f) \text{ handle } X => 4; \)
      - \( \text{end} \text{ handle } X => 6; \)
    - Which handler is used?

- **Compare to static scope of variables**
  - exception \( X; \)
    - (let fun \( f(y) = \text{raise } X \) and \( g(h) = h(1) \) handle \( X => 2 \))
      - \( \text{in} \)
    - \( g(f) \text{ handle } X => 4; \)
      - \( \text{end} \text{ handle } X => 6; \)
    - 

## Static Scope of Declarations

- **Static Scope of Declarations**
  - \( \text{val } x=6; \)
    - (let fun \( f(y) = x \) and \( g(h) = \text{let } \text{val } x=2 \text{ in } h(1) \) handle \( X => 2 \))
      - \( \text{in} \)
    - \( \text{let } \text{val } x=4 \text{ in } g(f) \text{ handle } X => 6; \)
      - \( \text{end}; \)
    - 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: \( 1 + \text{raise } X \)
- **Typing of handle (exn) => (value)**
  - Converts exception to normal termination
  - Need type agreement
    - Examples
      - \( 1 + (\text{raise } X) \text{ handle } X => e_j \)
      - \( 1 + (e_i \text{ handle } X => e_j) \text{ Type of } e_i, e_j \text{ must be int} \)
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);
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

Dynamic Scope of Handler

- Idea:
  - The continuation of an expression is "the remaining work to be done after evaluating the expression"
- Continuation of e is a function normally applied to e

- General programming technique
  - Capture the continuation at some point in a program
  - Use it later: "jump" or "exit" by function call
- Useful in
  - Compiler optimization: make control flow explicit
  - Operating system scheduling, multiprogramming
  - Web site design

Example of Continuation Concept

- Expression
  - $2x + 3y + 1/x + 2/y$
- 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

\[
\text{fact}(n) = \begin{cases} 
    n & \text{if } n=0 \\
    \text{if } n \neq 0 \text{ then } n \cdot \text{fact}(n-1) & \text{else}
\end{cases}
\]

- Continuation of **fact(6)** is 6 \* fact(5) = 6 \* 120 = 720
- Continuation of **fact(7)** is 7 \* fact(6) = 7 \* 720 = 5040
- Continuation of **fact(8)** is 8 \* fact(7) = 8 \* 5040 = 40320
- Continuation of **fact(9)** is 9 \* fact(8) = 9 \* 40320 = 362880

Derivation of tail recursive form

- Standard function
  \[
  \text{fact}(n) = \begin{cases} 
    n & \text{if } n=0 \\
    \text{if } n \neq 0 \text{ then } n \cdot \text{fact}(n-1) & \text{else}
\end{cases}
\]
- Continuation form
  \[
  \text{fact}(n, \text{k}) = \begin{cases} 
    \text{k} & \text{if } n=0 \\
    \text{else } \text{fact}(n-1, \text{k} \cdot \text{n}) & \text{else}
\end{cases}
\]
- Example computation
  \[
  \text{fact}(3, 1) = \text{fact}(2, 3) = 3 \cdot \text{fact}(1, 1) = 3 \cdot 1 = 3
\]
- Example computation
  \[
  \text{fact}(3, \lambda x. x) = \text{fact}(2, \lambda y. 3 \cdot y) = 3 \cdot \text{fact}(1, \lambda x. 2 \cdot x) = 6
\]

Tail Recursive Form

- Optimization of continuations
  \[
  \text{fact}(n, a) = \begin{cases} 
    a & \text{if } n=0 \\
    \text{if } n \neq 0 \text{ then } a \cdot \text{fact}(n-1, a) & \text{else}
\end{cases}
\]

- Each continuation is effectively \(\lambda x. (a \cdot x)\) for some \(a\)

- Example computation
  \[
  \text{fact}(3, 1) = \text{fact}(2, 3) \quad \text{was} \quad \text{fact}(2, \lambda y. 3 \cdot y) = \text{fact}(1, 6) \quad \text{was} \quad \text{fact}(1, \lambda x. 6 \cdot x) = 6
\]

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 hanged in his time
  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 (see fun and Also on Leland)
  - callcc: 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 applied 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 of 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 Web programming

• Asynchronous XHR also similar to continuations:

  ```
  function callWithContinuation(url, k) {
    var xhr = new XMLHttpRequest();
    xhr.open('GET', url, true);
    xhr.onreadystatechange = function() {
      if (xhr.readyState == 4)
        k(xhr.responseText);
    }
    xhr.send(null);
  }
  ```

  Usage: callWithContinuation('http://a.com/describe?id=10', alert);

  • Client continues while server runs
  • Basis of AJAX Web programming paradigm

“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**

(this is complicated...)

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

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

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

except DISAGREE; except 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 and ML compilation, some OS projects, web application development, ...