Programming Languages

Control Abstraction

Benjamin J. Keller

Department of Computer Science, Virginia Tech

Overview

- Runtime Stack Review
- Calling sequence
- Parameter passing
- Generic subroutines
- Exceptions
- Coroutines

Subroutines

- Control abstraction — defined to perform some operation
- Program calls subroutime, which performs operation and returns to caller
- Distinguish between *functions* with return value, and *procedures* without return value
- Represented during execution by unit instance, composed of code segment and activation record
- Activation Record Structure:
  - Return address
  - Access info on parameters
  - Space for local variables
- Units often need access to non-local variables.

Stack-based Languages

- Examples: ALGOL 60/Pascal
- Conflict during procedure activation between static (scope) and dynamic (execution) environments
- Stack reflects dynamic environment: activation record pushed at procedure call, and popped after return
- Activation record structure:
  1. Return address
  2. Dynamic link
  3. Static link
  4. Parameters and local variables

Stack-Based Languages (Example)

```
Program main;
  type array_type = array [1..10] of real;
  var a : integer;
      b : array_type;
  Procedure x (var c : integer; d : array_type);
    var e : array_type;
    procedure y (f : array_type);
      var g : integer;
        begin ...  z(a+c); ...  end; {y}
      begin {x}
        ... := b[6]...
        y(e);
      end; {x}
  Procedure z (h : integer);
    var a : array_type;
      begin ... x (h,a); ... end;
 begin {main}
   ... x (a,b); ...
 end. {main}
```

Stack-Based Languages (Example)

- Static scope: `x`, `z` in `main`, `y` in `x`
- What is dynamic call sequence?
  Main ⇒ _ ⇒ _ ⇒ _ ⇒ _ ⇒ _ ⇒ ...
- What does run-time stack look like after `x` calls `y` 2nd time?
- How do we get reference to
  - `b` in `x`?
  - `a` in `y`?
- Where are these variables stored?

Stack-Based Languages (cont)

- *Dynamic link* — pointer to caller's activation record
- *Static link* — pointer to beginning of activation record of statically containing program unit

Locating Variables in Stack-Based Languages

- Must keep track of the static nesting level of each variable and procedure
- When access variable or procedure, subtract static nesting level of definition from static nesting level of the access
- Tells how far down static chain to environment of definition
- Example:

Name	Level	Name	Level	Name	Level	Name	Level
main	0	x	1	y	2	z	1
a	1	c	2	f	3	h	2
b	1	d	2	g	3	a	2
		e	2				

Notes on Locating Variables

1. Length of static chain from any fixed procedure to main program is always same length (independent of activation)
2. Any non-local variable will be found after some fixed number of static links (independent of activation)
3. The number of links is a constant that can be computed at compile time as the difference between nesting level of call and callee
4. Thus represent identifier references in program as pair: $\langle \text{chain position}, \text{offset} \rangle$
5. Example: from within `y` represent `d` as $\langle 1, nx + 2 \rangle$ where nx is size of activation record of `x` before parameters. Similarly, `a` is represented as $\langle 2, nmain + 1 \rangle$

Display

- If static chain requires k dereferences, then an object will require $O(k)$ memory accesses.
- Display allows constant time access
- Array where jth element is reference to most recently active subroutine at lexical nesting level j
- If current routine at level i, then access entry $j = i - k$
- If display is in memory, only two memory accesses to reach object
- Display not commonly used

Calling Sequences

- Prologue (calling procedure):
  1. Make parameters available to callee
  2. Save state of caller (register, program counter)
  3. Make sure callee knows how to find where to return to
  4. Enter callee at 1st instruction
- Epilogue (returning from procedure):
  1. Get return address and transfer execution to that point
  2. Caller restores state
  3. If function, make sure result value left in accessible location (register, on top of stack, etc.)

Saving Registers

- Much of work can be done either by caller or callee
- Having callee do most of work can be more efficient
- With registers generally only want to save those used by both caller and callee, but too hard to know which those are
- Simple strategy: caller saves registers using, callee saves registers will use
- MIPS strategy:
  - Registers are *caller-saves* or *callee-saves*
  - Use callee-saves for local variables
  - Use caller-saves for transient values

Managing Static Chain

- Caller usually does work
- Cases
  - Callee is nested inside the caller — link is to caller
  - Callee is $k \geq 0$ scope levels out — dereference links
- Static links passed as implicit parameter
- Displays can be updated like static links
- Also, callee can save old entry at level j on stack, and put its own frame pointer into display

Inline Expansion

- Alternative to stack-based calling convention
- Call is replaced by code of subroutine
- Avoids storage allocations and allows optimizations not possible without inlining
- Programmer makes suggestion, and compiler decides
- Compiler dependent criteria for whether expansion is done
- C++ `inline` keyword
- Ada uses pragma — comment with message to compiler

Procedure Parameters

- Use of parameters supports abstraction — Creates more flexible program phrases.
- Mechanisms for accessing non-local information:
  - Common block, Global variables
  - Parameters — data, subprograms, types

Kinds of Parameters

- Call by Reference (FORTRAN, Pascal):
  - Pass address of actual parameter
  - Access via indirection
  - What if parameter is expression or constant? CHGTO4(2).
- Call by Copy (Algol 60, Pascal, C, etc.):
  - Actual parameter copies value to formal parameter (and/or vice-versa)
  - value (in), result (out), value-result (in-out)
  - result and value-result parameters must be variables, value can be any storable value
  - Can be expensive for large parameters.

Kinds of Parameters

- Call by Name (Algol-60)
  - Actual parameter provides expression to formal parameter — re-evaluated whenever accessed

```
Procedure  swap(a, b : integer);
    var temp : integer;
  begin
      temp := a;
      a := b;
      b := temp
  end;
```

  - Won't always work

    `swap(i, a[i]) with i = 1, a[1] = 3, a[3] = 17.`
  - No way to define a correct swap in Algol-60!

Call-by-Name (cont)

- Expressive power — Jensen's device:
  To compute: $x = \sum_{i=1}^{n} V i$

```
real procedure SUM (k, lower, upper, ak);
    value lower, upper;
    integer k, lower, upper;
    real ak;
    begin
        real s;
        s := 0;
        for k := lower step 1 until upper do
            s := s + ak;
        sum := s
    end;
```

- What is result of sum(i, 1, m, A[i])?
- What about sum(i, 1, m, sum(j, 1, n, B[i,j]))?

Call-by-Name (cont)

- If evaluating parameters has side-effects (e.g., read), then must know how many times parameter is evaluated to predict what will happen.
- Therefore try to avoid call-by-name with expressions with side-effects.
- Lazy evaluation is efficient implementation of call-by-name where only evaluate parameter once. Requires that there be no side-effects, since otherwise get different results.
- Implement call-by-name using thunks — procedures which evaluate expressions — difficult and slow. Must pass around code for evaluating expression (including environment).
- Note different from call-by-text (which would allow capture of free variables).

Parameter Passing

- Can classify parameter passing as *copying* (value, result, or value-result) or *definitional*
- Definitional parameters are constant, variable, procedural, or functional
- Constant parameters are treated as values, not variables — different from call-by-value. Default for Ada in parameters.
- Can think of call-by-name as definitional with expression parameter.
- Note that difference in parameter passing depends on what is bound (value or address) and when it is bound.
- Already seen how to pass functional (and procedural) parameters in our interpreter using closures.

Exceptions

- Need mechanism to handle exceptional conditions
- Example: Trying to pop element off of an empty stack
- Clearly corresponds to mistake of some sort, but stack module doesn't know how to respond
- Without exception handling:
  - print error message and halt
  - function/procedure returns boolean success flag — programmer has to check
  - Add procedure parameter which handles exceptions

Exceptions

- Exception mechanism in programming languages allows raising an exception which is sent back to caller for handling
- A *robust* program is able to recover from exceptional conditions, rather than just halting (or crashing).
- Typical exceptions:
  - Arithmetic or I/O faults (e.g., divide by 0, read int and get char, array or subrange bounds, etc.)
  - failure of precondition,
  - unpredictable conditions (read past end of file, end of printer page, etc.),
  - tracing program flow during debugging.
- Raised exception must be handled or program will fail

Ada Exception Handling

- Raise exception with `raise exception_name`
- Attach exception handlers to subprogram body, package body, or block

```
begin
        C
exception
        when excp_name1 => C'
        when excp_name2 => C''
        when others => C'
end
```

Locating Exception Handler

- When an exception is raised, must be handled or caught
- Typical approach to locating handler
  - Look for handler in current block (or subprogram)
  - If not there, force return from unit and raise same exception to routine which called current one
  - Continue up the dynamic links until find handler or get to outer level and fail.
- Semantics of raising and handling exceptions is dynamic rather than static
- Handler can attempt to handle exception, but give up and raise another exception

Resuming After Exceptions

- Once exception is handled what happens next?
- Ada: return from the procedure (or unit) containing the handler — called *termination* model.
- PL/I: re-execute statement where failure occurred (makes sense for read errors, for example) unless handler forces otherwise (with goto) — called *resumption* model
- Eiffel (an OOL): uses variant of resumption model.
- ML: exceptions can pass parameter to exception handlers (like values in datatype). Otherwise very similar to Ada.

ML Exceptions

- Example program to check for balanced parenthesis in a string

```
datatype 'a stack = EmptyStack | Push of 'a * ('a stack);
exception empty;
fun pop EmptyStack = raise empty
  | pop(Push(n,rest)) = rest;
fun top EmptyStack = raise empty
  | top (Push(n,rest)) = n;
fun IsEmpty EmptyStack = true
  | IsEmpty (Push(n,rest)) = false;

exception nomatch;

fun buildstack nil initstack = initstack
  | buildstack ("("::rest) initstack = buildstack rest (Push("(",initstack))
  | buildstack (")"::rest) (Push("(",bottom)) = bottom
  | buildstack (")"::rest) initstack = raise nomatch
  | buildstack (fst::rest) initstack = buildstack rest initstack;

fun balanced string = (buildstack (explode string) = EmptyStack)
                        handle nomatch => false;
```

ML Exceptions (cont)

- Notice that need to put parentheses around the expression to which the handler is associated – awkward
- Might argue that this is not unexpected situation. Just a way fancy way of introducing goto's.

Implementing Exceptions

- Goal is to have no overhead during normal execution
- Solution: maintain table of protected block and handlers
  - Entries: start address of block, address of handler
  - Table is sorted by address, and use binary search when exception occurs
  - Include pointer to table in stack frame for separately compiled code
- Solution: C `setjmp` and `longjmp` — save and restore program state, must mark variables as `volatile` so that variables in registers are saved to memory
- Solution: continuation passing — closure passed to continuation passing mechanism allows execution to proceed in any environment

Coroutines

- An execution context that exists concurrently
- Coroutine is an abstraction that employs a closure (code address, and referencing environment)
- `transfer` jumps into coroutine at current location
- Subsequent transfers jump to last location
- Complicates scoping
- Used to implement iterators and in discrete event simulation
- Analogous to threads

Coroutine Example

```
us, cfs : coroutine

coroutine update_screen -- initialize
   detach               -- initiate concurrent execution
   loop
      ...
      transfer(cfs)
      ...

coroutine check_file_system
   detach
   for all files
      ...
      transfer(us)
      ...

begin                   -- main
   us := new update_screen
   cfs := new check_file_system
   resume(us)
```