Chapter 8 :: Subroutines and Control Abstraction

Programming Language Pragmatics

Michael L. Scott

Administrative Notes

• Final Test
  – Thursday, August 3 2006 at 11:30am
  – No lecture before or after the mid-term test
  – You are responsible for material presented in the lectures not necessarily covered in the textbook
  – Similar format (time) to the mid-term test

• Final Test Review Tomorrow
  – We will go over the solutions of test 1
  – I will let you know what sections you are responsible for

Administrative Notes

• Assignment Two
  – Due date: Friday, August 4 2006 at 1:00pm
  – Submit your assignment in the drop-box located at the Computer Science and Engineering undergraduate office
  – Late assignments are subject to a penalty of 10% each day
  – I may choose to mark only a subset of the assigned questions
  – You must "show your work" where appropriate to obtain full marks

Administrative Details

• Test 1
  – Will be returned to you tomorrow (Wednesday)

• Assignment Three
  – Will be placed online one Wednesday

Review

• What is type compatibility ?
• What is type equivalence ?
• What are the two major approaches for type checking ?
• What is coercion ?
• Name two languages where coercion is highly used
• How are records arranged in memory ?
• Describe the issues with records/memory allocation

Review

• What is an array and what is the relationship between arrays/pointers in C ?
• What is an array slice ?
• Does C support array slices ?
• How can the elements of the array be accessed (two methods)
• What are the two layout strategies for array
• What is a String ?
• What are pointers ?
• What is garbage collection ?
Review Of Stack Layout

- Allocation strategies
  - Static
    - Code
    - Globals
    - Own variables
    - Explicit constants (including strings, sets, other aggregates)
  - Small scalars may be stored in the instructions themselves

- Allocation strategies (2)
  - Stack
    - parameters
    - local variables
    - temporaries
    - bookkeeping information
  - Heap
    - dynamic allocation

- Contents of a stack frame
  - bookkeeping
    - return PC (dynamic link)
    - saved registers
    - line number
    - saved display entries
    - static link
    - arguments and returns
    - local variables
    - temporaries

Calling Sequences

- Maintenance of stack is responsibility of calling sequence and subroutine prolog and epilog – discussed in Chapter 3
  - space is saved by putting as much in the prolog and epilog as possible
  - time may be saved by putting stuff in the caller instead, where more information may be known
    - e.g., there may be fewer registers IN USE at the point of call than are used SOMEWHERE in the callee

- Common strategy is to divide registers into caller-saves and callee-saves sets
  - caller uses the "callee-saves" registers first
  - "caller-saves" registers if necessary
- Local variables and arguments are assigned fixed OFFSETS from the stack pointer or frame pointer at compile time
  - some storage layouts use a separate arguments pointer
  - the VAX architecture encouraged this
Calling Sequences

Caller

- saves into the temporaries and locals area any caller-saves registers whose values will be needed after the call
- puts up to 4 small arguments into registers $4-$7 (a0-a3)
- it depends on the types of the parameters and the order in which they appear in the argument list
- puts the rest of the arguments into the arg build area at the top of the stack frame
- does jal, which puts return address into register ra and branches
  - note that jal, like all branches, has a delay slot

In prolog, Callee

- subtracts framesize from sp
- saves callee-saves registers used anywhere inside callee
- copies sp to fp

In epilog, Callee

- puts return value into registers (mem if large)
- copies fp into sp (see below for rationale)
- restores saved registers using sp as base
- adds to sp to deallocate frame
- does jra

This is a normal state of affairs; optimizing compilers keep things in registers whenever possible, flushing to memory only when they run out of registers, or when code may attempt to access the data through a pointer or from an inner scope

Many parts of the calling sequence, prologue, and/or epilogue can be omitted in common cases

- particularly LEAF routines (those that don't call other routines)
  - leaving things out saves time
  - simple leaf routines don't use the stack - don't even use memory – and are exceptionally fast
Parameter Passing

- Parameter passing mechanisms have three basic implementations
  - value
  - value/result (copying)
  - reference (aliasing)
- Many languages (e.g., Pascal) provide value and reference directly

- Many languages (e.g., Pascal) provide value and reference directly

C/C++: functions
- parameters passed by value (C)
- parameters passed by reference can be simulated with pointers (C)
  ```
  void proc(int * x, int y) {*x = *x+y }
  ...
  proc(&a,b);
  ```
- or directly passed by reference (C++)
  ```
  void proc(int & x, int y) {x = x + y }
  proc(a,b);
  ```

Ada goes for semantics: who can do what
- In: callee reads only
- Out: callee writes and can then read (formal not initialized); actual modified
- In out: callee reads and writes; actual modified
- Ada in/out is always implemented as
  - value/result for scalars, and either
  - value/result or reference for structured objects

In a language with a reference model of variables (Lisp, Clu), pass by reference (sharing) is the obvious approach
- It's also the only option in Fortran
  - If you pass a constant, the compiler creates a temporary location to hold it
  - If you modify the temporary, who cares?
- Call-by name is an old Algol technique
  - Think of it as call by textual substitution (procedure with all name parameters works like macro) - what you pass are hidden procedures called THUNKS

Generic Subroutines and Modules

- Generic modules or classes are particularly valuable for creating containers: data abstractions that hold a collection of objects
- Generic subroutines (methods) are needed in generic modules (classes), and may also be useful in their own right
Exception Handling

- What is an exception?
  - a hardware-detected run-time error or unusual condition detected by software
- Examples
  - arithmetic overflow
  - end-of-file on input
  - wrong type for input data
  - user-defined conditions, not necessarily errors

Exception Handling

- What is an exception handler?
  - code executed when exception occurs
  - may need a different handler for each type of exception
- Why design in exception handling facilities?
  - allow user to explicitly handle errors in a uniform manner
  - allow user to handle errors without having to check these conditions
  - explicitly in the program everywhere they might occur

Coroutines

- Coroutines are execution contexts that exist concurrently, but that execute one at a time, and that transfer control to each other explicitly, by name
- Coroutines can be used to implement
  - iterators (Section 6.5.3)
  - threads (to be discussed in Chapter 12)
- Because they are concurrent (i.e., simultaneously started but not completed), coroutines cannot share a single stack

Coroutines

Figure 6.5: A coroutine stack. Each branch to the side represents the creation of a coroutine (A, B, C, and D). The static nesting of blocks is shown at right. Static links are shown with arrows. Dynamic links are indicated simply by vertical arrangement; each routine has called the one above it.