Chapter 7:: Data Types (cont.)

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

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

- Drop Deadline
  - Computer Science Department drop-deadline for the course is July 31 2006
  - If you wish to drop the course before this deadline will have to petition to "drop late" as far as the registrars office is concerned, the drop deadline has passed (they treated the course as D2 - see registrars office)
    - Of course, the petition will have the support of the department and of myself as well and will be approved under these circumstances

Review

- What is a data type ?
- What is the purpose of a data types e.g., what are they good for ?
- What is strong typing ?
- What is static typing ?
- What is orthogonality ?
- What is a type system ?
- What is type compatibility ?
- What is type equivalence ?

Review

- What is structural equivalence ?
- What is name equivalence ?
- Which of the two above equivalences is more “fashionable” currently ?
- What are the two main variants of name equivalence ?

Type Checking

- Two major approaches: structural equivalence and name equivalence
  - Name equivalence is based on declarations
  - Structural equivalence is based on some notion of meaning behind those declarations
  - Name equivalence is more fashionable these days

Type Checking

- There are at least two common variants on name equivalence
  - The differences between all these approaches boils down to where you draw the line between important and unimportant differences between type descriptions
  - In all three schemes described in the book, we begin by putting every type description in a standard form that takes care of "obviously unimportant" distinctions like those above

Type Checking

- Structural equivalence depends on simple comparison of type descriptions substitute out all names
  - expand all the way to built-in types
- Original types are equivalent if the expanded type descriptions are the same

Type Checking

- Coercion
  - When an expression of one type is used in a context where a different type is expected, one normally gets a type error
  - But what about

```
var a : integer; b, c : real;
     ...
c := a + b;
```


Type Checking

- Coercion
  - Many languages allow things like this, and COERCE an expression to be of the proper type
  - Coercion can be based just on types of operands, or can take into account expected type from surrounding context as well
  - Fortran has lots of coercion, all based on operand type

Type Checking

- C has lots of coercion, too, but with simpler rules:
  - all `floats` in expressions become `doubles`
  - short `int` and `char` become `int` in expressions
  - if necessary, precision is removed when assigning into LHS

Type Checking

- In effect, coercion rules are a relaxation of type checking
  - Recent thought is that this is probably a bad idea
  - Languages such as Modula-2 and Ada do not permit coercions
  - C++, however, goes hog-wild with them
  - They're one of the hardest parts of the language to understand

Type Checking

- Make sure you understand the difference between
  - type conversions (explicit)
  - type coercions (implicit)
  - sometimes the word 'cast' is used for conversions (C is guilty here)

Records (Structures) and Variants (Unions)

- Records
  - usually laid out contiguously
  - possible holes for alignment reasons
  - smart compilers may re-arrange fields to minimize holes (C compilers promise not to)
  - implementation problems are caused by records containing dynamic arrays
    - we won't be going into that in any detail

Records (Structures) and Variants (Unions)

- Unions (variant records)
  - overlay space
  - cause problems for type checking
- Lack of tag means you don't know what is there
- Ability to change tag and then access fields hardly better
  - can make fields "uninitialized" when tag is changed (requires extensive run-time support)
  - can require assignment of entire variant, as in Ada

Records (Structures) and Variants (Unions)

- Memory layout and its impact (structures)

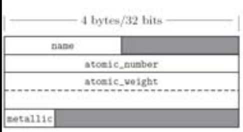

Figure 7.1: **Likely layout in memory for objects of type `element` on a 32-bit machine.** Alignment restrictions lead to the shaded "holes."

Records (Structures) and Variants (Unions)

- Memory layout and its impact (structures)

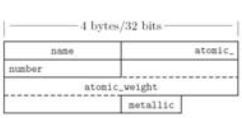

Figure 7.2: **Likely memory layout for packed `element` records.** The `atomic_number` and `atomic_weight` fields are nonaligned, and can only be read or written (on most machines) via multi-instruction sequences.

Records (Structures) and Variants (Unions)

- Memory layout and its impact (structures)

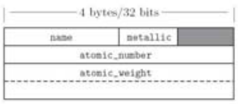

Figure 7.3: **Rearranging record fields to minimize holes.** By sorting fields according to the size of their alignment constraint, a compiler can minimize the space devoted to holes, while keeping the fields aligned.

Records (Structures) and Variants (Unions)

- Memory layout and its impact (unions)

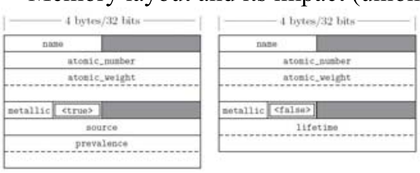

Figure 7.4: **Likely memory layouts for `element` variants.** The value of the `naturally_occurring` field (shown here with a double border) determines which of the interpretations of the remaining space is valid. Type `string_ptr` is assumed to be represented by a (four-byte) pointer to dynamically allocated storage.

Records (Structures) and Variants (Unions)

- Memory layout and its impact (unions)

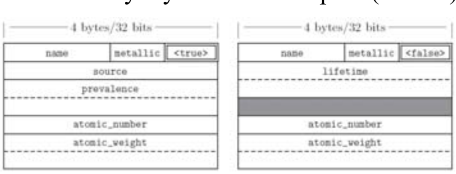

Figure 7.5: **Likely memory layout for a variant record in Modula-2.** Here the variant portion of the record is not required to lie at the end. Every field has a fixed offset from the beginning of the record, with internal holes as necessary following small-size variants.

Arrays

- Arrays are the most common and important composite data types
- Unlike records, which group related fields of disparate types, arrays are usually homogeneous
- Semantically, they can be thought of as a mapping from an *index type* to a *component* or *element type*
- A *slice* or *section* is a rectangular portion of an array (See figure 7.6)

Arrays

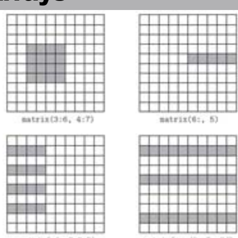

Figure 7.6: **Array slices (sections) in Fortran 90.** Much like the values in the header of an enumeration-controlled loop (Section 6.5.1), $a : b : c$ in a subscript indicates positions $a, a + c, a + 2c, \ldots$ through b. If a or b is omitted, the corresponding bound of the array is assumed. If c is omitted, 1 is assumed. It is even possible to use negative values of c in order to select positions in reverse order. The slashes in the second subscript of the lower right example delimit an explicit list of positions.

Arrays

- Dimensions, Bounds, and Allocation
  - ***global lifetime, static shape*** — If the shape of an array is known at compile time, and if the array can exist throughout the execution of the program, then the compiler can allocate space for the array in static global memory
  - ***local lifetime, static shape*** — If the shape of the array is known at compile time, but the array should not exist throughout the execution of the program, then space can be allocated in the subroutine's stack frame at run time.
  - ***local lifetime, shape bound at elaboration time***

Arrays

```
procedure foo (size : integer) is
M : array (1..size, 1..size) of real;
...
begin
...
end foo;
```

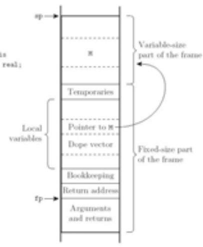

Figure 7.7: **Allocation in Ada of local arrays whose shape is bound at elaboration time.** Here M is a square two-dimensional array whose width is determined by a parameter passed to `foo` at run time. The compiler arranges for a pointer to M to reside at a static offset from the frame pointer. M cannot be placed among the other local variables because it would prevent those higher in the frame from having static offsets. Additional variable-size arrays are easily accommodated. The purpose of the *dope vector* field is explained in Section 7.4.3.

Arrays

- Contiguous elements (see Figure 7.9)
  - column major - only in Fortran
  - row major
    - used by everybody else
    - makes array [a..b, c..d] the same as array [a..b] of array [c..d]

Arrays

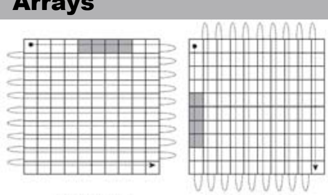

Figure 7.9: **Row- and column-major memory layout for two-dimensional arrays.** In row-major order, the elements of a row are contiguous in memory; in column-major order, the elements of a column are contiguous. The second cache line of each array is shaded, on the assumption that each element is an eight-byte floating-point number, that cache lines are 32 bytes long (a common size), and that the array begins at a cache line boundary. If the array is indexed from A[0,0] to A[9,0], then in the row-major case elements A[0,4] through A[0,7] share a cache line; in the column-major case elements A[4,0] through A[7,0] share a cache line.

Arrays

- Two layout strategies for arrays (Figure 7.10):
  - Contiguous elements
  - Row pointers
- Row pointers
  - an option in C
  - allows rows to be put anywhere - nice for big arrays on machines with segmentation problems
  - avoids multiplication
  - nice for matrices whose rows are of different lengths
    - e.g. an array of strings
  - requires extra space for the pointers

Arrays

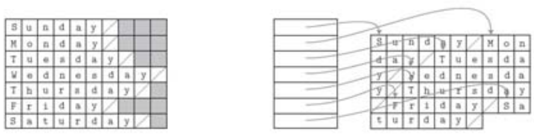

Figure 7.10: **Contiguous array allocation v. row pointers in C.** The declaration on the left is a true two-dimensional array. The slashed boxes are NUL bytes; the shaded areas are holes. The declaration on the right is an array of pointers to arrays of characters. In both cases, we have omitted bounds in the declaration that can be deduced from the size of the initializer (aggregate). Both data structures permit individual characters to be accessed using double subscripts, but the memory layout (and corresponding address arithmetic) is quite different.

Arrays

- **Example**: Suppose

```
A : array [L1..U1] of array [L2..U2]
  of array [L3..U3] of elem;
  D1 = U1-L1+1
  D2 = U2-L2+1
  D3 = U3-L3+1
```

Let

```
  S3 = size of elem
  S2 = D3 * S3
  S1 = D2 * S2
```

Arrays

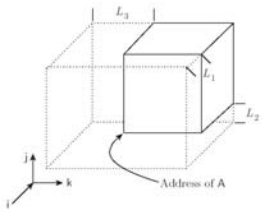

Figure 7.11: **Virtual location of an array with nonzero lower bounds.** By computing the constant portions of an array index at compile time, we effectively index into an array whose starting address is offset in memory, but whose lower bounds are all zero.

Arrays

- **Example** (continued)

  We could compute all that at run time, but we can make do with fewer subtractions:

```
== (i * S1) + (j * S2) + (k * S3)
     + address of A
  - [(L1 * S1) + (L2 * S2) + (L3 * S3)]
```

  The stuff in square brackets is compile-time constant that depends only on the type of A

Strings

- Strings are really just arrays of characters
- They are often special-cased, to give them flexibility (like polymorphism or dynamic sizing) that is not available for arrays in general
  - It's easier to provide these things for strings than for arrays in general because strings are one-dimensional and (more important) non-circular

Sets

- We learned about a lot of possible implementations
  - Bitsets are what usually get built into programming languages
  - Things like intersection, union, membership, etc. can be implemented efficiently with bitwise logical instructions
  - Some languages place limits on the sizes of sets to make it easier for the implementor
    - There is really no excuse for this

Pointers And Recursive Types

- Pointers serve two purposes:
  - efficient (and sometimes intuitive) access to elaborated objects (as in C)
  - dynamic creation of linked data structures, in conjunction with a heap storage manager
- Several languages (e.g. Pascal) restrict pointers to accessing things in the heap
- Pointers are used with a value model of variables
  - They aren't needed with a reference model

Pointers And Recursive Types

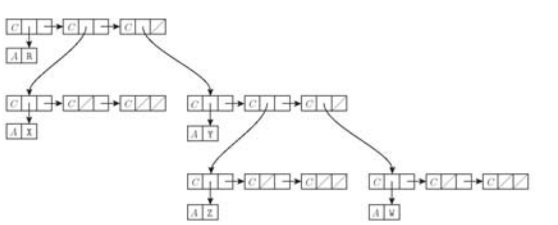

Figure 7.13: **Implementation of a tree in Lisp.** A diagonal slash through a box indicates a `nil` pointer. The *C* and *A* tags serve to distinguish the two kinds of memory blocks: cons cells and blocks containing atoms.

Pointers And Recursive Types

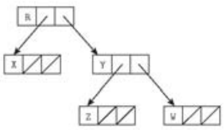

Figure 7.14: **Typical implementation of a tree in a language with explicit pointers.** As in Figure 7.13, a diagonal slash through a box indicates a `nil` pointer.

Pointers And Recursive Types

- C pointers and arrays

```
int *a == int a[]
int **a == int *a[]
```

- BUT equivalences don't always hold
  - Specifically, a declaration allocates an array if it specifies a size for the first dimension
  - otherwise it allocates a pointer

```
int **a, int *a[] pointer to pointer to int
int *a[n], n-element array of row pointers
int a[n][m], 2-d array
```


Pointers And Recursive Types

- Compiler has to be able to tell the size of the things to which you point
  - So the following aren't valid:

```
int a[][]      bad
int (*a)[]     bad
```

  - C declaration rule: read right as far as you can (subject to parentheses), then left, then out a level and repeat

```
int *a[n], n-element array of pointers to
  integer
int (*a)[n], pointer to n-element array
  of integers
```

Pointers And Recursive Types

- Problems with dangling pointers are due to
  - explicit deallocation of heap objects
    - only in languages that *have* explicit deallocation
  - implicit deallocation of elaborated objects
- Two implementation mechanisms to catch dangling pointers
  - Tombstones
  - Locks and Keys

Pointers And Recursive Types

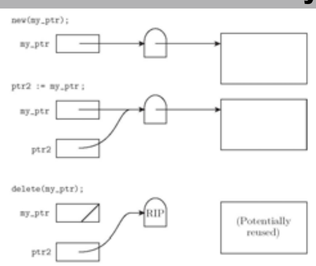

Figure 7.15: **Tombstones.** A valid pointer refers to a tombstone that in turn refers to an object. A dangling reference refers to an "expired" tombstone.

Pointers And Recursive Types

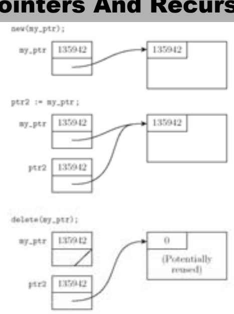

Figure 7.16: **Locks and Keys.** A valid pointer contains a key that matches the lock on an object in the heap. A dangling reference is unlikely to match.

Pointers And Recursive Types

- Problems with garbage collection
  - many languages leave it up to the programmer to design without garbage creation - this is VERY hard
  - others arrange for automatic garbage collection
  - reference counting
    - does not work for circular structures
    - works great for strings
    - should also work to collect unneeded tombstones

Pointers And Recursive Types

- Garbage collection with reference counts

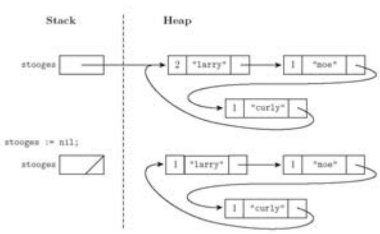

Figure 7.17: **Reference counts and circular lists.** The list shown here cannot be found via any program variable, but because it is circular, every cell contains a nonzero count.

Pointers And Recursive Types

- Mark-and-sweep
  - commonplace in Lisp dialects
  - complicated in languages with rich type structure, but possible if language is strongly typed
  - achieved successfully in Cedar, Ada, Java, Modula-3, ML
  - complete solution impossible in languages that are not strongly typed
  - conservative approximation possible in almost any language (Xerox Portable Common Runtime approach)

Pointers And Recursive Types

- Garbage collection with pointer reversal

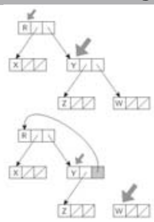

Figure 7.18: **Heap exploration via pointer reversal.** The block currently under examination is indicated by the large grey arrow. The previous block is indicated by the small grey arrow. As the garbage collector moves from one block to the next, it changes the pointer it follows to refer back to the previous block. When it returns to a block it restores the pointer. Each reversed pointer must be marked, to distinguish it from other, forward pointers in the same block. We assume in this figure that the root node R is outside the heap, so none of its pointers are reversed.

Lists

- A list is defined recursively as either the empty list or a pair consisting of an object (which may be either a list or an atom) and another (shorter) list
  - Lists are ideally suited to programming in functional and logic languages
    - In Lisp, in fact, a program *is* a list, and can extend itself at run time by constructing a list and executing it
  - Lists can also be used in imperative programs

Files and Input/Output

- Input/output (I/O) facilities allow a program to communicate with the outside world
  - *interactive* I/O and I/O with files
- Interactive I/O generally implies communication with human users or physical devices
- Files generally refer to off-line storage implemented by the operating system
- Files may be further categorized into
  - *temporary*
  - *persistent*