Programming Language & Compiler

Types

Hwansoo Han

Data Types

- Intuitive notion of what types are:
  - Denotational point of view
    - A set of values from a "domain"
  - Constructive point of view
    - Either a small collection of built-in types (integer, character, boolean, real, etc.) or
    - A composite type created by constructor (record, array, set, etc.)
  - Abstraction-based point of view
    - Collection of well-defined operations that can be applied to objects of that type

What Are Types Good For?

- Provide implicit context
  - Make sure that certain meaningless operations do not occur
    - a + b : use integer addition, if types of both are integer

- Limit the set of operations
  - Prevent programmers from using semantically invalid operations (e.g., add a *character* to a *record*)

  - Type checking cannot prevent all meaningless operations
    - It catches enough of them to be useful

Type System

- Type system consists of
    1. A mechanism to define types and their language constructs
    2. A set of rules for type equivalence, compatibility, inference

- Notions in type system
    - Type equivalence
        - When are the types of two values <u>the same</u>?
    - Type compatibility
        - When <u>can</u> a value of type A <u>be used</u> in a context that expects type B?
    - Type inference
        - What is the <u>type of an expression</u>, given the types of the operands?

Type Checking – Strong vs. Weak

- Strong typing
  - A popular buzz-word like *structured programming*
  - Informally, prevents you from applying an inappropriate operation to data
  - <u>Strongly typed languages</u>
    - Ada, Java
- Weak typing
  - <u>Weak typed languages</u>
    - C, C++
    - C89 is more strongly typed than its predecessor dialects, but less strongly typed than Pascal

Type Checking – Static vs. Dynamic

- Static typing
  - Means that all the type checking can be done at compile time
  - Statically typed languages
    - e.g., Ada, Pascal
    - In practice, most type-checking can be done at compile time,
      - But not 100% can be done at compile time – needs dynamic checking (e.g., array index range check)

- Dynamic typing
  - Dynamic type checking
    - Lisp, Smalltalk, and most scripting languages (e.g., Python and Ruby) perform type checking at run time (but strongly typed)
    - Languages with dynamic scoping are generally dynamically typed

Classification of Types

- Numeric types
  - integer, real
    - multiple precision (bit length): short/int/long, single/double-precision
    - signed, unsigned, decimal (base-10), fixed-point (real number with two integers)
- Enumeration types
  - type weekday = {sun, mon, tue, wed, thu, fri, sat};
- Subrange types
  - type score = 0..100; workday = mon..fri;
- Composite types (constructed types)
  - Records (structures), variant records (unions), arrays, strings, sets, pointers (often implement recursive data types), lists, files

Orthogonality in Type System Design

- A useful goal in the design of languages (and types)
  - A collection of features is orthogonal if there are no restrictions on the ways in which the features can be combined

- Example: vector type
  - C and Pascal are more orthogonal than Fortran in arrays
    - C and Pascal allows any type as an element
    - Fortran77 allows only scalar type as an element

Type Checking - Compatibility

- Types of an object in a context are constrained
  - An object can be used in a context, if its type is type-equivalent to the type that is expected in that context
- Type compatibility, a looser relation than equivalence
  - Type of an object and type of a context are compatible, even when their types are not equivalent, but allowed to be used
- Type checking is to check the type compatibility

Structural vs. Name Equivalence

- Two major approaches in equivalence
  - Structural type system – structural equivalence
  - Nominative type system – name equivalence

- Structural equivalence
  - Based on implementation-oriented view (Algol-68, Modula-3, C, ML)
    - Cannot distinguish two types that have the same structure by coincidence
- Name equivalence
  - Based on type names in declarations (Java, C#, Pascal, Ada)

struct XY
{ int x;
  int y;
}
	= ?	struct coordinate
		{ int x;
		  int y;
		}
			= ?	struct reverseXY
				{ int y;
				  int x;
				}

Type Conversion

- Static typing expects a specific type for many contexts
  - *a* = *expr* --- *expr* should be the same type of *a*
  - *a* + *b* --- + requires both *a* and *b* are integers or reals
  - *foo*(*a*, *b*) --- *a* and *b* should be the same types of *foo*'s formal parameters
- Cases in type conversion
  - Structurally equivalent
    - No conversion code is needed
  - Two types have common values (subrange, signed/unsigned)
    - Dynamic check is needed to avoid semantic errors
  - Structurally different
    - Convert may result in loss of precision, overflow
      - Conversions among int, unsigned, float, double
    - Many processors provide conversion instructions

Type 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

```
int a;   float b, c;
         ...
c = a + b;
```

- If languages allow different types than the expected one, implicit type conversion (called *type coercion* in this case) should take place

Type Coercion (cont'd)

- Many languages coerce an expression to be of the proper type
  - Coercion can be based just on types of operands, or expected type from surrounding context as well
- Fortran
  - All based on operand type
- C
  - all `float`s in expressions become `double`s
  - short `int` and `char` become `int` in expressions
  - if necessary, precision is removed when assigning to l-value

Type Coercion (cont'd)

- Coercion rules are a relaxation of type checking
- Modern languages advocate static typing, away from type coercion
  - Modula-2 and Ada do not permit coercions
  - C++, however, provides extremely rich set of rules for type coercions
    - `const_cast, static_cast, reinterpret_cast`

Type Conversion – Misc.

- Understand the differences
  - Type conversion -- general term for all
  - Type coercion -- implicit type conversion required by languages
  - Type cast -- used for explicit type conversion by programmers

- Non-converting type cast
  - No conversion code is used at all
  - The bits of underlying implementation of a type is interpreted as another type

Records and Variants

- Records (= structures)
  - Usually laid out contiguously
  - Possible holes for alignment reasons

  - Smart compilers may rearrange fields to minimize holes
    - But C compilers promise not to rearrange
  - Kernel programs sometimes assume a particular layout
    - E.g., to handle memory-mapped control registers for a device

Records and Variants

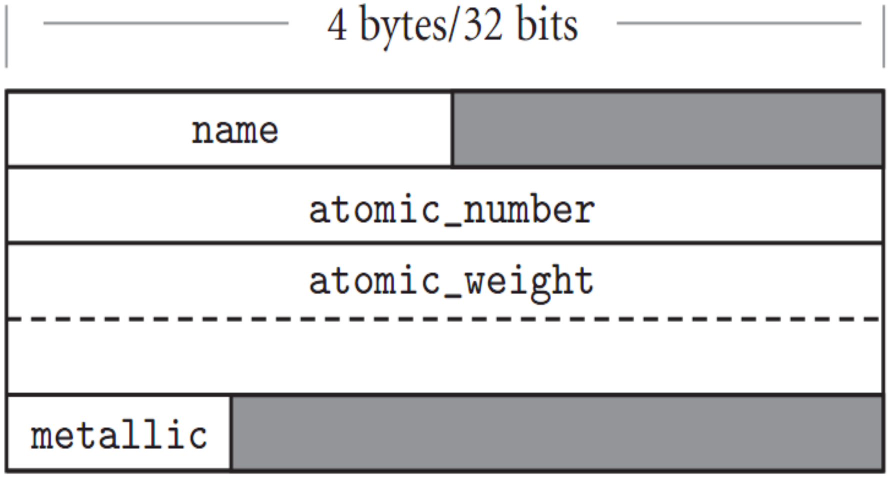
- Variant records (= unions)
  - Overlay space
  - Cause problems for type checking

- Main usage patterns
  - Same bytes interpreted in different ways at different times
  - Alternative sets of fields within a record
    - Common fields + various other fields

Structure Memory Layout

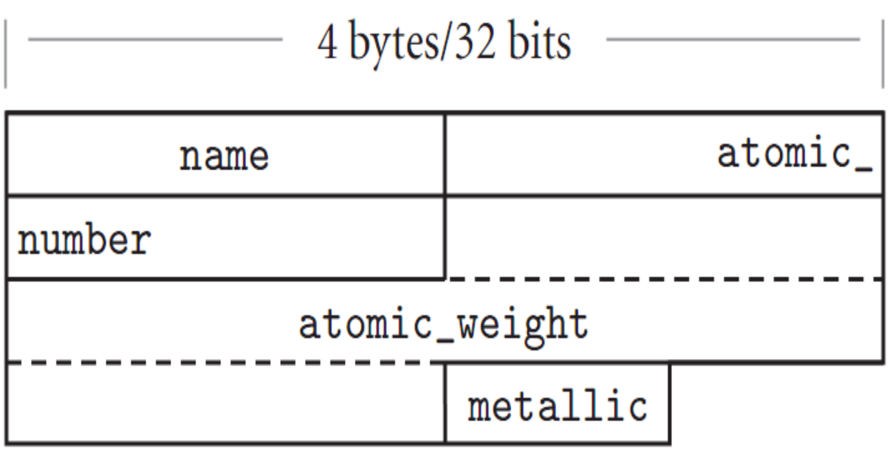
- Holes due to alignment
  - If not aligned, multiple instructions needed to read a field
  - For speed, aligning fields is needed

```
struct Atom {
    short   name;
    int     atomic_number;
    double  atomic_weight;
    char    metalic;
}
```

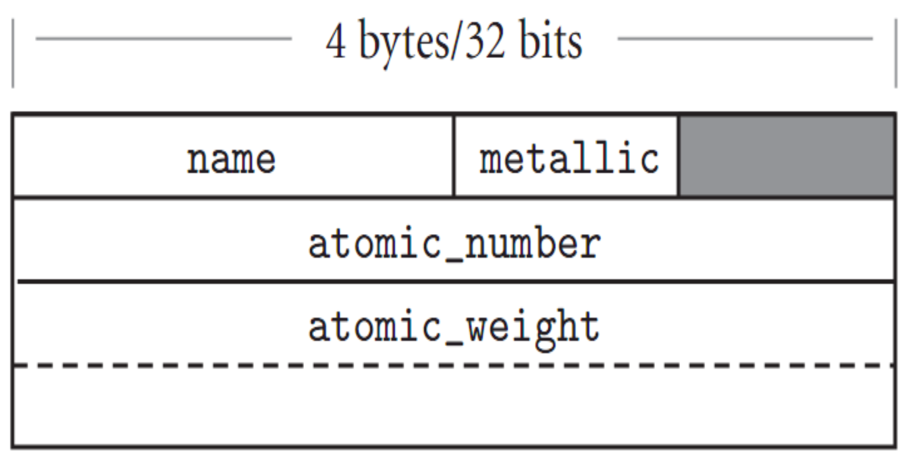


Structure Memory Layout (cont'd)

- Packed record
  - Allows the compiler to optimize for the space
    - May require multiple instructions to read non-aligned field
- Reordered (sorted) fields
  - Can minimize the space due to holes



Packed record



Reordered record

Arrays

- Most common and important composite data types

- Homogeneous elements, unlike records
  - Fortran77 requires element type be scalar
  - Elements can be any type (Fortran90, etc.)

- A mapping from an *index type* to a *component* or *element type*
  - Fortran requires index type to be *integer*
  - Many languages allow index to be any *discrete type* (integers, Booleans, characters -- countable)

Dimension, Bounds, and Allocations

- Static shape
  - The shape of an array is known at compile time
    - Shape = dimensions, bounds
  - Allocation
    - Global life time – allocate in global memory at compile time
    - Local life time – allocate in stack frame at run time
- Dynamic shape
  - Shape is known at elaboration time (module entry time)
    - Allocate in the stack frame
  - Shape changes during the execution
    - Allocate in the heap

Dope Vectors

- A dope vector contains
  - Lower bound and size of each dimension
  - Upper bound (redundant) but useful to avoid computation in dynamic bound check

- If dimensions and their sizes of an array are static
  - The compiler can lookup symbol table and generate code to calculated the addresses (no need of dope vectors)

- If dimensions and their sizes are not statically-known
  - These are *dynamic shape arrays*
  - The compiler generates address calculation code to include the dope vector lookup

Dynamic Shape Arrays

- Shape of an array is determined at run time
  - Shape = number of dimensions and their bounds

- Conformant arrays
  - Arrays are used as parameters and
  - Their <u>bounds</u> can be <u>symbolic names</u> rather than constants
  - Conformant array is an example of dynamic arrays
    - Shape is determined by the function parameters

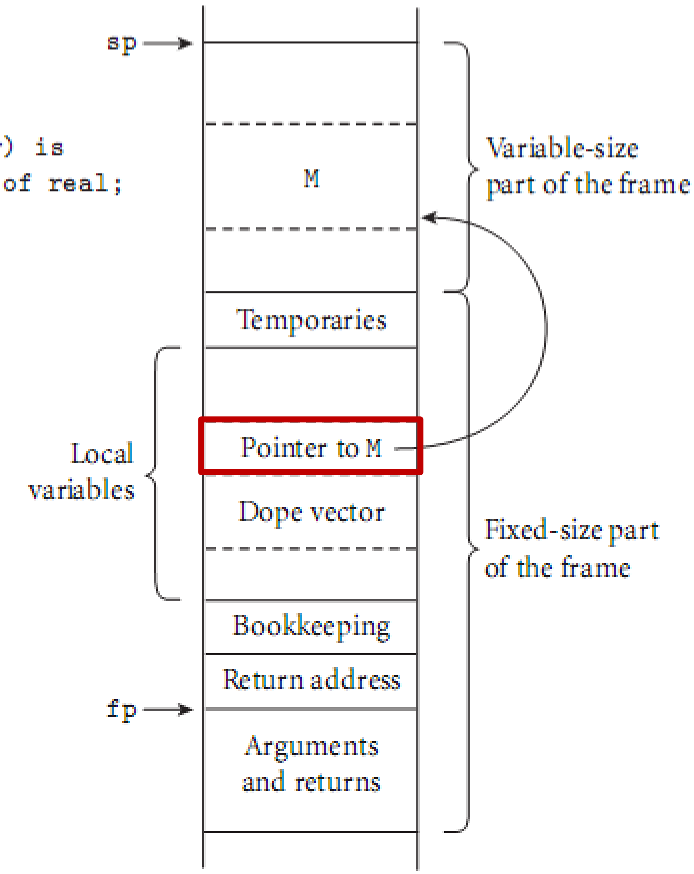
```
void square( int n, double M[n][n] ) {   // n is determined at runtime
    double T[n][n];
    ...
}
```

Dynamic Shape Arrays in Stack Frame

- Additional indirection is used

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

```
// C99:
void foo(int size) {
    double M[size][size];
    ...
}
```



Dynamic Shape Arrays in Heap

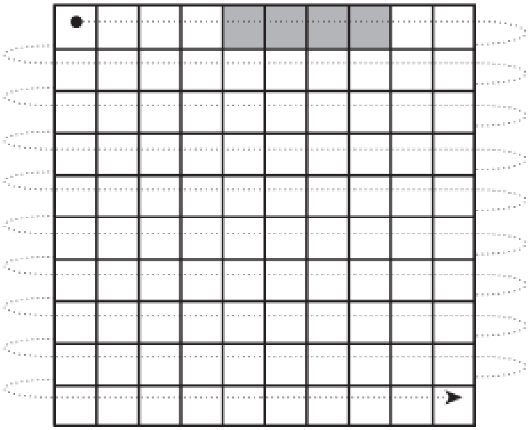
- Fully dynamic shape arrays
  - Can change their shapes arbitrary points of a program
  - Need to accommodate these arrays in the heap
  - Examples
    - Variable length strings (Java, C#)

      String s = "short";   ...   s = s + " but sweet";
    - Dynamically resizable arrays
      - Vector class, ArrayList class in C++, Java, C# libraries

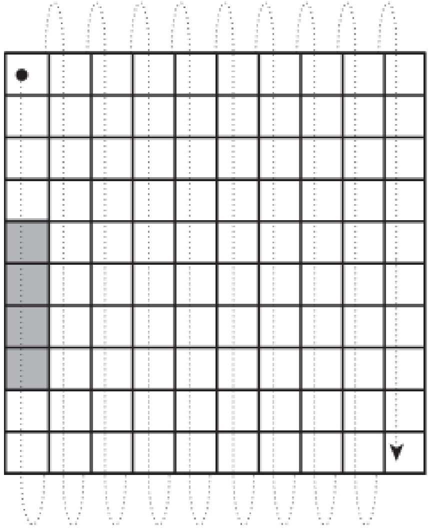
- Fully dynamic shape arrays with local lifetime
  - Space reclamation code is needed

Memory Layouts of Arrays

- Contiguous elements
- Multidimensional arrays
  1. Column major -- only in Fortran
  2. Row major -- used by everybody else
     - array [a..b, c..d] is the same as array [a..b] of array [c..d]



Row-major order



Column-major order

Memory Layouts of Arrays (cont'd)

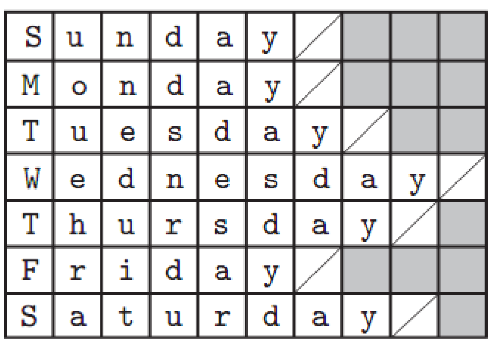
3. Row pointers
   - An option in C (with pointers), but all arrays in Java
   - Allows rows to be put anywhere
     - Nice for big arrays on machines with external segmentations
     - Can use existing, but scattered rows of contents
   - Avoids multiplication, but perform memory load
   - Nice for *ragged arrays* whose rows are of different lengths
     - e.g., an array of strings
   - Requires extra space for the pointers

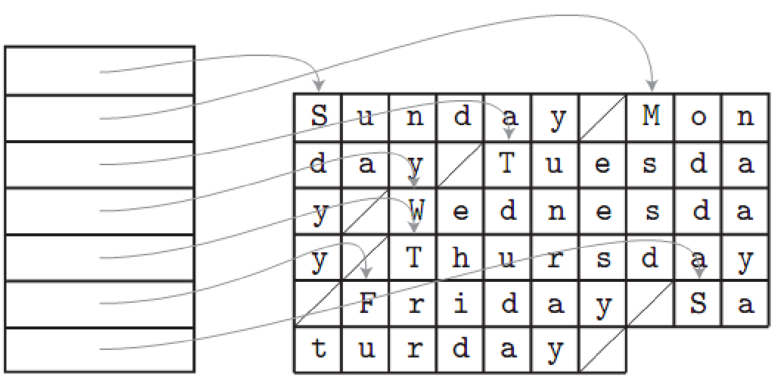
Row-Pointer Layout for Arrays

- Can save space for ragged arrays
- But need an extra pointer for a row

```
char days[][10] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...
days[2][3] == 's';  /* in Tuesday */
```

```
char *days[] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...
days[2][3] == 's';  /* in Tuesday */
```





Address Calculations for Arrays

A : array [L1..U1] of array [L2..U2] of array [L3..U3] of elem_type;

D1 = U1-L1+1

D2 = U2-L2+1

D3 = U3-L3+1

Number of composing elements in each dimension

S3 = size of elem_type

S2 = D3 * S3

S1 = D2 * S2

Size of each dimension in bytes

Address of A[i][j][k] = address_of_A
+ (i – L1) * S1 + (j-L2) * S2 + (k-L3) * S3

= (i * S1) + (j * S2) + (k * S3)
+ address_of_A - $\underbrace{\text{[(L1 * S1) + (L2 * S2) + (L3 * S3)]}}_{\text{Compile-time constant}}$

for static shape arrays

Strings

- Strings are really just arrays of characters

- Dynamic sizing is often allowed by language designers
  - Variable-length strings are fundamental to many applications
    - C++, Java, C#: string is a built-in class
    - ML, Lisp: string is a chain of blocks (linked list of chars)
  - Specially allowed in languages with no dynamic array
    - String operations (assign, concatenate, ...) implicitly create new objects and change reference to them
    - Unused space for unreachable string objects should be reclaimed automatically

Sets

- We learned about a lot of possible implementations
  - Arrays
  - Hash tables
  - Trees
  - Bit vector (characteristic array)

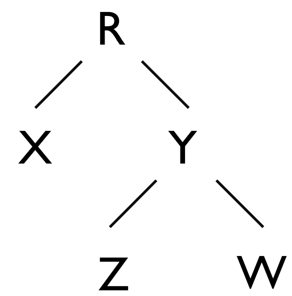
- A bit vector is fast for modest number of elements
  - Intersection, union, membership, etc. can be implemented efficiently with bitwise logical instructions
  - Some languages place limits on the sizes of sets to make it easier for the implementer

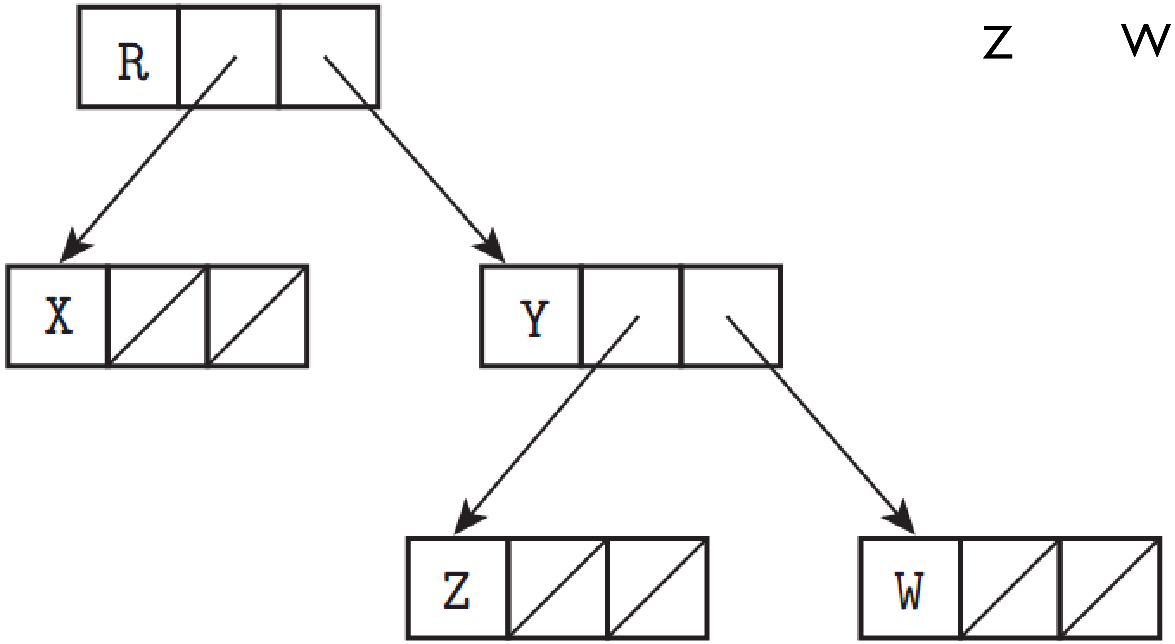
Pointers And Recursive Types

- Pointers serve two purposes:
  - Efficient access to elaborated objects (as in C)
  - Dynamic creation of linked data structures – *recursive type* (in conjunction with a heap storage manager)
- Several languages (e.g., Pascal) restrict pointers only to access objects in the heap
- Pointers are used with a value model of variables
  - They aren't needed with a reference model (they are already pointers for all variables)

Binary Tree Types in Value Model

```
struct btree {
    char   c;
    struct btree *left;
    struct btree *right;
}
```

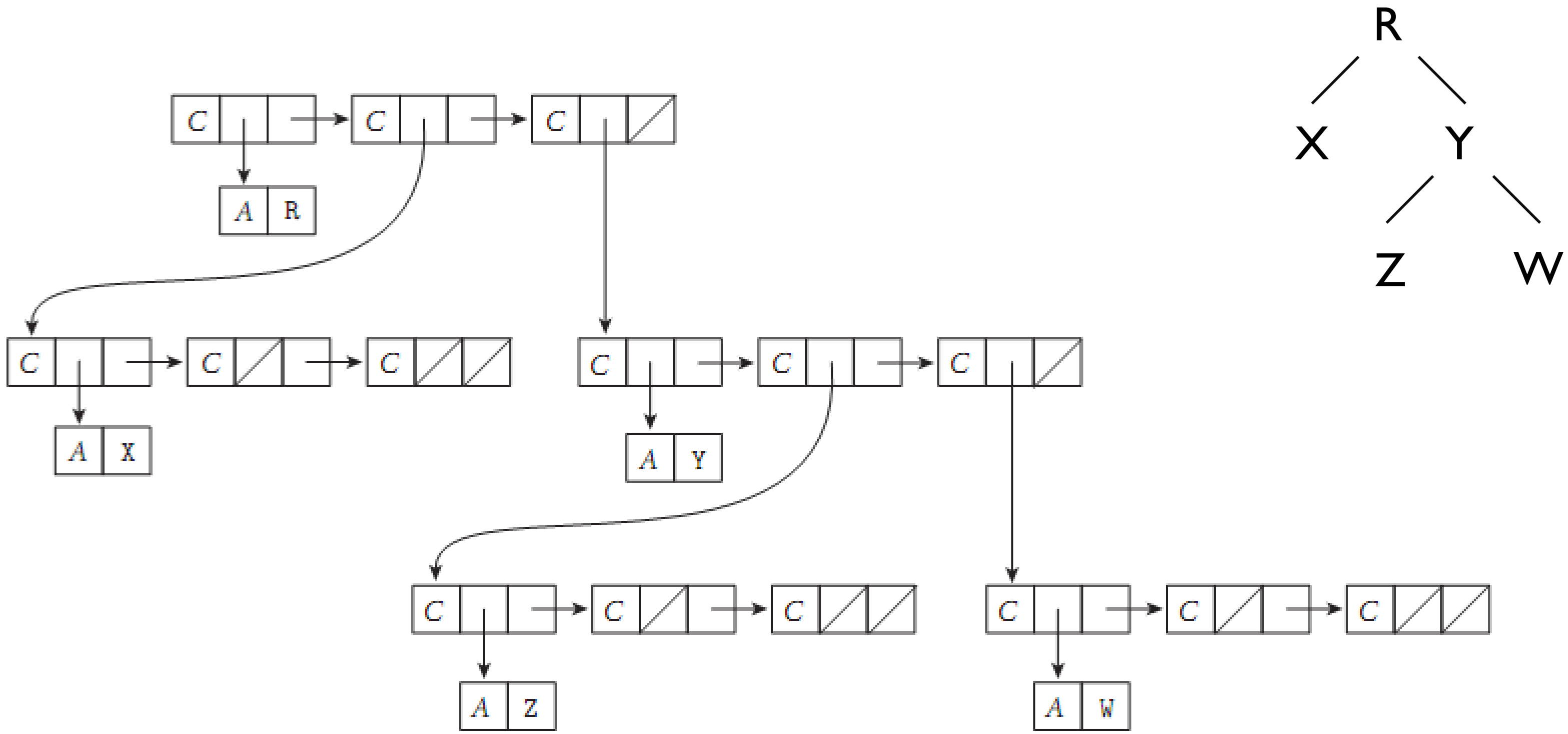




Binary Tree Types in Reference Model

ML: datatype btree = empty | node of char * btree * btree;

Lisp : '(#\R (#\X () ()) (#\Y (#\Z () ()) (#\W () ())))



Implementation in Lisp (*C* – cons, *A* – atom)

Pointers and Arrays in C

- Types are compatible

```
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 Arrays in C

- 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:
    - ( ), [ ] – highest precedence, left-to-right associativity
    - * – lower precedence, right-to-left associativity

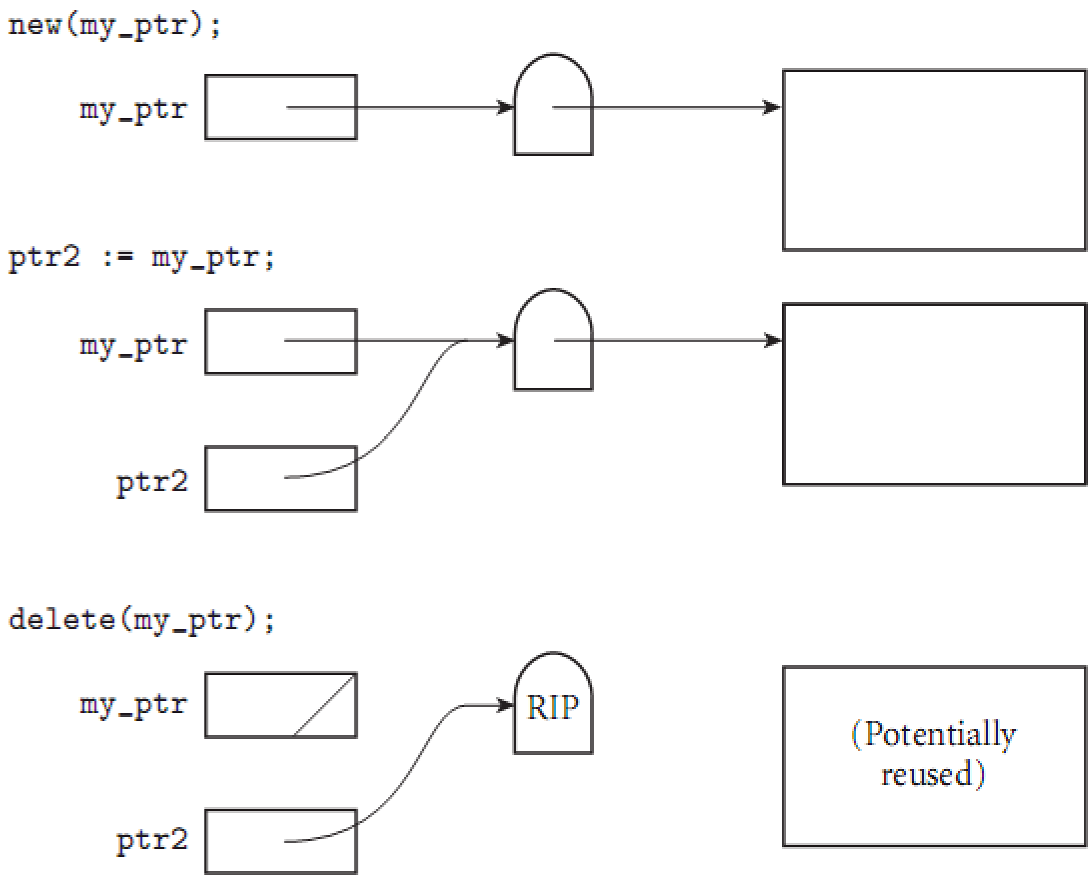
```
int *a[n]      // n-element array of pointers to integer
int (*a)[n]    // a pointer to n-element array of integers
```

Dangling References

- Dangling reference is
  - A live pointer that no longer points to a valid object
- Dangling pointer problems are often due to
  - Explicit deallocation of heap objects
    - Only in languages that *have* explicit deallocation
  - Implicit deallocation of elaborated objects

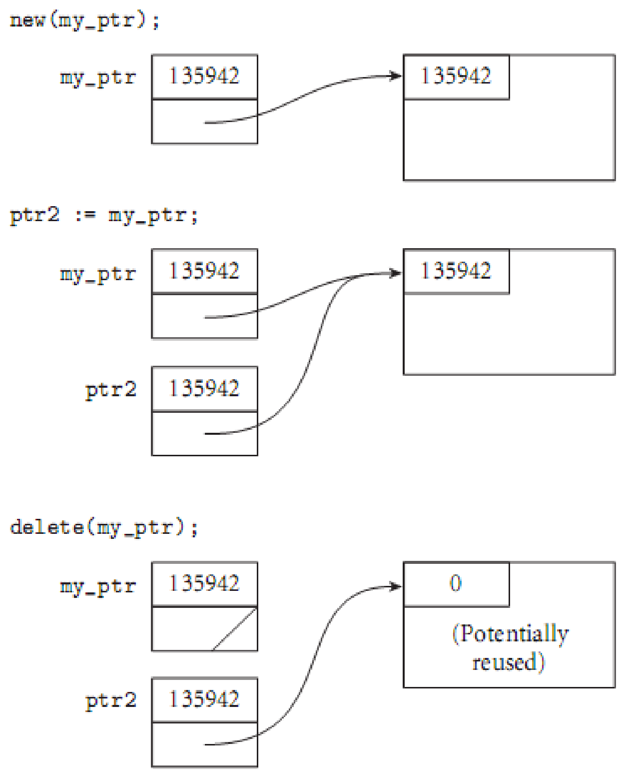
Tombstones – dangling pointer

- Extra indirection data to mark the validity of objects



Locks and Keys - dangling pointer

- Key – a word added to every pointer
- Lock – a word added to every heap object
- To be a valid pointer, its lock and key should match



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 provided in imperative programs
  - Built-in types (Clu)
  - Class libraries (C++, Java, etc.)

Files

- Input/output (I/O) facilities allow a program to communicate with the outside world
  - Interactive I/O
    - Communicates with human users or physical devices
  - File I/O
    - Communicates with files - off-line storage provided by OS
    - *Temporary* files vs. *persistent* files

- Files in languages
  - Some languages provide built-in *file* data types
  - Other languages relegate I/O entirely to library packages

Summary (1)

- Type checking
  - Strong vs. weak
  - Static vs. dynamic
- Type
  - Equivalence
  - Compatibility
  - Conversion
  - Coercion
- Structures and variants
  - Layout, alignment and holes

Summary (2)

- Structure & variant
- Array
  - Static vs. dynamic shape
  - Dope vector
  - Memory layout (row-major, column-major, row-pointers)
- String
  - Implementation to support dynamic length
- Set
  - Array, hash, tree, bit-vector implementations
- Pointers & recursive types
  - Dangling pointers
- Misc. – List, File