Types II

Hwansoo Han
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 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 calculate 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 bounds can be symbolic names rather than constants
- Conformant array is an example of dynamic arrays
  - Shape is determined by the function parameters

```c
void square( int n, double M[n][m] ) {   // 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#)
    ```java
    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 \([c..d]\)
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

```c
char days[10] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};

...    
days[2][3] == 's'; /* in Tuesday */
```

```c
char *days[] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};

...    
days[2][3] == 's'; /* in Tuesday */
```
Address Calculations for Arrays


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 - [(L1 * S1) + (L2 * S2) + (L3 * S3)]

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

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

```
    R
   /   
  X     Y
 /     /
Z     W
```
Binary Tree Types in Reference Model

ML: \texttt{datatype btree = empty | node of char * btree * btree;}

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

Implementation in Lisp (\emph{C} – cons, \emph{A} – atom)
Pointers and Arrays in C

- Types are compatible
  
  \[
  \text{int } *a == \text{int } a[ ] \\
  \text{int } **a == \text{int } *a[ ] \\
  \]

- 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

  \[
  \text{int } **a, \text{int } *a[ ] \quad \text{// pointer to pointer to int} \\
  \text{int } *a[n] \quad \text{// n-element array of row pointers} \\
  \text{int } a[n][m] \quad \text{// 2-d array} \\
  \]
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:
    ```c
    int a[ ][ ]    // bad
    int (*a)[ ]    // bad
    ```

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

    ```c
    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

```c
new(my_ptr);
my_ptr

ptr2 := my_ptr;
my_ptr
ptr2

delete(my_ptr);
my_ptr
RIP
ptr2
```

(Potentially reused)
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

```c
new(my_ptr);
my_ptr | 135942 --> 135942

ptr2 := my_ptr;
my_ptr | 135942
ptr2  | 135942

delete(my_ptr);
my_ptr | 135942
ptr2  | 135942
```

(Potentially reused)
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

- 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