Data Types

- Two components:
  - set of objects in the type
  - applicable operations

- May be determined:
  - statically (at compile time)
  - dynamically (at run time)

- A language’s data types may be:
  - built-in
  - programmer-defined

- Languages may be:
  - strongly typed
  - weakly typed (untyped?)

Primitive Types

- Scalars:
  - arithmetic types
  - logical types
  - boolean types (T,F)
  - strings (may be a structure in some cases)
  - characters

- Structures:
  - arrays
  - lists
  - records
  - associative arrays (see PERL)

- Program elements:
  - program units
Strings

- Array of char or primitive string type

- Length
  - fixed (static)
  - limited dynamic (dynamic up to the point of first allocation, static thereafter)
  - dynamic

- Operations
  - substring reference
  - (con)catenation
  - relational operations
  - pattern matching

Implementing Strings

- Static
  - Need length descriptor only at compile-time

- Limited dynamic
  - Need max length and current length descriptors at runtime

- Dynamic
  - Need current length descriptor at runtime
  - Dynamic storage allocation:
    - linked list
    - contiguous memory
Ordinal Types

- Each element can be associated with an integer
  - character
  - boolean
  - user-defined
    - enumeration
      - Can a literal appear in more than one type? If so, how to distinguish?
        - alphabet = [a..z]
        - vowels = [a,e,i,o,u]
    - subrange
      - How to typecheck?
        - i: integer;
        - j: 1..10;
        - ...j := i; -- prohibit, or check dynamically?

Arrays (finite mappings)

- homogeneous
- index computed dynamically
- binding of
  - subscript range
  - storage
    - static: compile time compile time
    - semistatic: compile time decl. elaboration time
    - semidyn: runtime, but fixed for lifetime
    - dynamic: runtime runtime

- Semidynamic & dynamic: a'first, a'last, a'length
- Type compatibility of arrays?
Implementing Arrays

- Stored contiguously
- Access ith element of a, where
  → b = address of a[1]
  → e = size of one element

\[
\text{addr}(a[i]) = b + e \times (i - 1) = b + ei - e = b - e + ei
\]

may know compute statically dynamically

Multidimensional Arrays

- Storage layout
  → row major
  → column major
- Row-major access a[i,j], where
  → b = address of a[1,1]
  → n x m = dimensions of a
  → e = size of one element

\[
\text{addr}(a[i,j]) = b + ((i - 1) \times m + (j - 1)) \times e
\]

\[
\begin{array}{ccccccc}
22 & 40 & 76 & 22 & 58 & 94 & \\
40 & 58 & 84 & 40 & 76 & 102 & \\
76 & 102 & 128 & 76 & 112 & 138 & \\
22 & 58 & 94 & 22 & 58 & 94 & \\
40 & 76 & 112 & 40 & 76 & 112 & \\
76 & 112 & 148 & 76 & 112 & 148 & \\
94 & 130 & 166 & 94 & 130 & 166 & \\
\end{array}
\]

\[
b = 22 \\
n = 5 \\
m = 6 \\
e = 3
\]

\[
a[4,3] = 22 + (3 \times 6 + 2) \times 3 = 82
\]
Records (Cartesian product)

- heterogeneous

- selector determined statically (why?)

- implementation
  - fields stored contiguously
  - offset of each field known statically
  - no runtime info necessary

Union Types

- may store different types during execution

- discriminated union ==> tag stores type of current value

- e.g., Pascal variant record

```pseudocode
type rec =
  record
    case flag : bool of
      true : (x : integer;
            y : char);
      false : (z : real)
    end
  end

var ex : rec;
ex.flag := true;
ex.x := 5
```
Type-checking Issues with Union Types

- System must check value of flag before each variable access
  
  ```
  ex.flag := true;
  ex.x := 10;
  :
  :
  print(ex.z);  -- error
  ```

- Still not good enough!
  
  ```
  ex.flag := true;
  ex.x := 5;
  ex.y := "a";
  ex.flag := false;
  print (ex.z);  -- this should be an error, but how to check
  ```

Pascal Free Union

- Declaration
  
  ```
  type rec = record
  case bool of
    true : . . .
    false : . . .
  end
  ```

- No storage for tag, so union is inherently unsafe.

- So Pascal's union type is insecure in at least two ways.
Ada Union Types

- Similar to Pascal, except
  - no free union
  - when tag is changed, all fields must be set too.
    
    \[
    \text{ex} := \text{(flag) \Rightarrow false,} \\
    \text{z \Rightarrow 1.5)}
    \]

- So Ada union types are safe.
  - Ada systems \textit{required} to check the tag of all references to variants

Algol 68 Union Types

- Declaration

  \[
  \text{union (int, real) ir1, ir2}
  \]

  - \textbf{Can assign} either type . . .
    
    \[
    \text{ir1 := 5;} \\
    ... \\
    \text{ir1 := 3.4;}
    \]

  - . . . but need conformity clause to \textbf{access} value

  \[
  \text{real x; int count;} \\
  ... \\
  \text{count := ir1; -- illegal}
  \]

  \[
  \text{case ir1 in} \\
  \text{(int i) : count := i; Type-checked statically,} \\
  \text{(real r) : x := r; chosen dynamically}
  \]

  esac
Pointers

- Should be able to point to only one type of object

- Dereferencing
  - explicit
  - implicit

- Used for
  - dynamic vars only
  - any kind of variable

Dangling References

- Pointer to variable that has been deallocated.

  Pascal:
  
  ```pascal
  var p,q : ^cell;
  begin
    new(p);
    q := p;
    dispose(p);
  end;
  ```

  C:
  
  ```c
  int *p;
  int fun1();
  {
    int x;
    p = &x;
    ... main ();
    fun1 ();
  }
  ```

  -- q is a dangling ref.
  -- *p is a dangling ref.
Preventing Dangling References

- Tombstones
  - Pointers can't point directly to a dynamic variable
    extra level of indirection called a tombstone.

  without tombstone:
  
  with tombstone:

  Safe, but add space and time overhead

---

Garbage ("dangling objects")

- An object is garbage if it is stored in an inaccessible memory address.

  - Pascal:
    ```pascal
    var p, q : ^cell;
    begin
      new(p);
      new(q);
      ...  -- assuming no dispose or reassign
    p := q;
    ```
    
    - original p^'s storage is now garbage
    - Wasteful, but not dangerous.
Heap Management

- **Allocation**
  - Maintain a free list of available memory cells

- **Deallocation (Reclamation)**

- **method 1: Reference Counting**
  - Each cell has a tag with # of pointers to that cell.
  - When reference count = 0 => deallocate cell.
  - **Advantage:**
    - cost is distributed over time
  - **Disadvantages:**
    - space/time overhead in maintaining reference counts
    - won't collect circular structures