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
    - continuous 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

<table>
<thead>
<tr>
<th>subscript range</th>
<th>storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>static: compile time</td>
<td>compile time</td>
</tr>
<tr>
<td>fixed stack-dynamic: compile time</td>
<td>declaration elaboration time</td>
</tr>
<tr>
<td>stack-dynamic: runtime, but fixed for lifetime</td>
<td>heap-dynamic: runtime</td>
</tr>
<tr>
<td>Fixed stack-dynamic and Stack-dynamic</td>
<td></td>
</tr>
<tr>
<td>a'first, a'last, a'length</td>
<td></td>
</tr>
</tbody>
</table>

- 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]
  - \( n \times m = \) dimensions of a
  - \( e = \) size of one element

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

beginning of ith row
dimensional offset in ith row

\[
\begin{align*}
22 & \quad 40 \\
58 & \quad 76 \\
94 & \quad \\
\end{align*}
\]

- \( b = 22 \)
- \( n = 5 \quad m = 6 \)
- \( e = 3 \)

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

```pascal
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
ture : . . .
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.
    \[
    \text{ex := (flag => false,}\n    \quad z => 1.5)\]

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

Algol 68 Union Types

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

  - Can assign either type . . .
    
    \[
    \begin{align*}
    \text{ir1} & : 5; \\
    & \ldots \\
    \text{ir1} & : 3.4; \\
    \end{align*}
    \]

  - . . . but need conformity clause to access value
    
    \[
    \begin{align*}
    \text{real x;} \\
    \text{int count;} \\
    & \ldots \\
    \text{count} & := \text{ir1}; \quad \text{-- illegal}\n    \end{align*}
    \]

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

    esac
### Sets

- **E.g.,**
  
  ```
  type day = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
  work_days: set of day;
  class_days: set of day;
  church_days: set of day;
  late_days, early_days, long_days, really_long_days: set of day;
  ```

  ```
  late_days := work_days ∩ class_days;
  early_days := work_days ∩ church_days;
  long_days := late_days ∪ early_days;
  really_long_days := late_days ∩ early_days;
  ```

- **Main issue:** cardinality of base set.

- **For efficiency,**
  - max set size ≤ word length
  - then
    - union = or
    - intersection = and etc...

- **But larger sets should be available, for a price...**

---

### 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:**

```
p1 ───> dyn var
p2
```

**with tombstone:**

```
p1 ───> tombstone ───> dyn var
p2
```

Safe, but add space and time overhead
Preventing Dangling References (continued)

- **Locks and Keys**
  - Additional information stored with pointers and dynamic variables:
    
    \[
    \text{pointer} = (\text{key, address}) \\
    \text{dynamic variable} = (\text{lock, var})
    \]
  - A pointer can only access a dynamic variable if its key matches the lock.
  - When a dynamic variable is deallocated, its lock is changed.
  - Again, 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

Heap Management (continued)

- **method 2a: Garbage Collection with mark-and-sweep**
  - Each cell has a mark bit.
  - **Mark and Sweep:**
    - set all mark bits to "garbage"
    - for each pointer into the heap, mark all reachable cells "not garbage"
    - look at each cell in memory; if marked "garbage," reclaim.
  - **Advantages:**
    - reclaims all garbage
    - little space/no time overhead during normal execution
  - **Disadvantages:**
    - must stop execution to garbage collect
    - fragmentation
    - time ~ memory size
  - **M&S initiated when system runs out of memory**
Heap Management (continued)

- method 2b: Garbage Collection with copying
  - Start with two heaps of same size
    
    *working heap*
    
    *other heap*
  
  - Copying:
    
    allocate new cells in *working heap*
    
    when *working heap* is full,
    
    for each pointer into *working heap*, copy all reachable cells into *other heap*.
    
    *other heap* is new *working heap*, *working heap* is new *other heap*
  
  - Advantages:
    
    both advantages of mark & sweep (reclaims all garbage, little space/no time overhead during normal execution)
    
    time ~ used cells, not total memory size
    
    automatic compaction (ameliorates fragmentation)
  
  - Disadvantages
    
    stopping to copy still a problem
    
    need twice as much memory for heap => only practical with virtual memory systems

Heap Management (continued)

- This is all much easier for fixed-size allocate/deallocate than for variable-size:
  
  - Fixed size (& format) Lisp
    
    Know where pointers are within cells
    
    Fragmentation not a problem
    
    to reclaim linked list (avoid garbage)
  
  - Variable size (& format)
    
    Need header for each object in memory telling:
    
    its size
    
    where it may contain pointers to other objects
    
    Fragmentation is a problem—need compaction