

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?)

Data Types as a Structure

- **The set of objects in the type**

- by enumeration
- by description

- **The set of applicable operations**

- **a standard set including equality, inequality, assignment**
- **specific operations**
 - need to provide operands types, result type
- **variable or constant?**
 - constant has no assignment operator

DECLARATIONS

- **Explicit association between identifier, types, and representations (values)**

Ada example:

```
-- scalar type declarations
type COLOUR is (red, orange, yellow, green, blue, indigo, violet) ;
type SMALL_INT is range 0..255 ;
-- array declaration
type INT_ARRAY is array (1..10) of INTEGER ;
-- array declaration without fixed bounds
type TEXT is array (NATURAL range <>) of CHARACTER ;
-- record declaration with pointer component
--- use a forward declaration of the pointer reference type to allow it to
-- be used in the pointer declaration
type ELEM ;
type LISTP is access ELEM ;
type ELEM is record
    Val: INTEGER ;
    Next: LISTP ;
end record ;
-- Some variable declarations
Rainbow: COLOUR ;
Char_value: SMALL_INT ;
First_10: INT_ARRAY ;
```

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

DON'T FORGET

- **EVERY IDENTIFIER HAS AN ASSOCIATED TYPE**
 - **procedure**
 - **block**
 - **label**
 - **function**
 - **program**
 - **task**
 - **operator**

Numeric Types

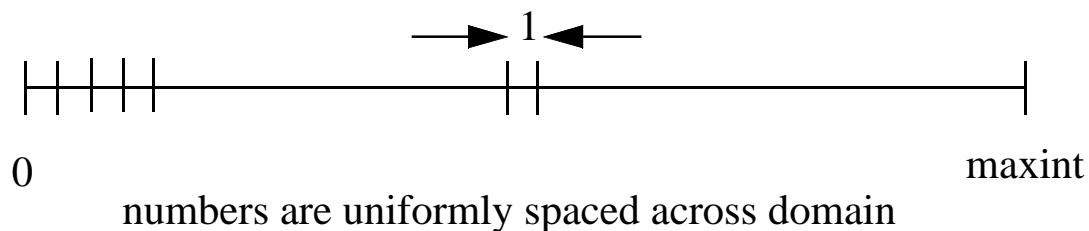
- Integer, Real, Double, Complex, ...
- Type must include range (maxint, ...)
- Platform will define precision (some languages override internal representation)
- Representationability depends on **MODEL NUMBERS** (i.e. the set of representable numbers)

PRECISION is a measure of a representation
ACCURACY is a measure of an operation

MODEL NUMBERS

DEFINITION: In any particular numeric domain, the set of exactly representable numbers

Integer Domain:

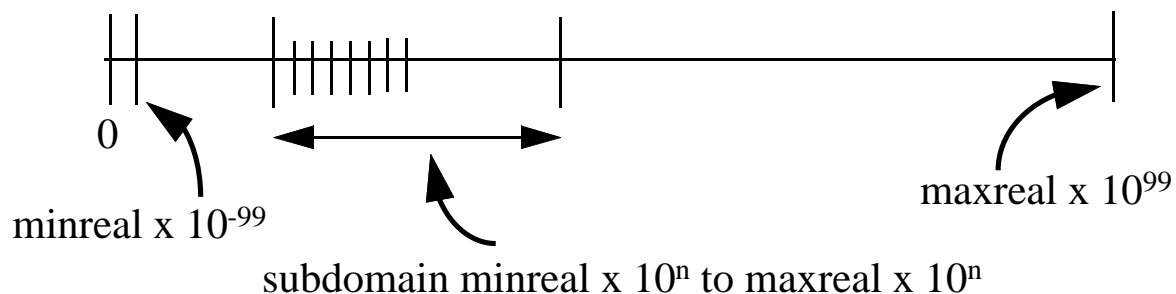


Floating Point Domain:

Assume representation is decimal with two-digit exponent (exrad) and five-digit normalized fractional mantissa (significand)

maxreal (significand) = .99999 minreal (significand) = .10000

Complete domain is sequence of subdomains each from minreal to maxreal of the mantissa (significand) of the representation spaced at $.00001 \times YY$ where YY is value of exrad



Type Compatibility

- **name**

- **efficient but restrictive**

- **structural**

- **flexible, harder to implement**

- **Issues:**

- hard to make types incompatible

- what is considered the same structure?

- array [0..10] of int

- array [1..11] of int

- what about two records?

- a: integer,
b: real;

- and

- x: integer,
y: real;

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

	<u>subscript range</u>	<u>storage</u>
<i>static:</i>	compile time	compile time
<i>semistatic:</i>	compile time	decl. elaboration time
<i>semidyn:</i>	runtime, but fixed for lifetime	
<i>dynamic:</i>	runtime	runtime

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

Implementing Arrays

- **Stored contiguously**
- **Access i th element of a , where**
 - **b = address of $a[1]$**
 - **e = size of one element**

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

$$= b + ei - e$$

$$= b - e + ei$$

may know
statically

compute
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

$$\text{addr}(a[i,j]) = b + \underbrace{((i - 1) * m)}_{\text{beginning of } i\text{th row}} + \underbrace{(j - 1)}_{\text{offset in } i\text{th row}} * e$$

22					
40					
58					
76		*			
94					

$b = 22$

$n = 5 \quad m = 6$

$e = 3$

$$a[4, 3] = 22 + (3 * 6 + 2) * 3 = 82$$

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

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

- ex := (flag => false,

- 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**

```
union (int, real) ir1, ir2
```

→ **Can assign either type . . .**

```
ir1 := 5;
```

```
...
```

```
ir1 := 3.4;
```

→ **. . . but need conformity clause to access value**

```
real x;
```

```
int count;
```

```
...
```

```
count := ir1;    -- illegal
```

```
case ir1 in
```

```
  (int i) : count := i;
```

```
  (real r) : x := r;
```

```
esac
```

Type-checked statically,
chosen dynamically

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  $\cap$  class_days;  
early_days := work_days  $\cap$  church_days;  
long_days := late_days  $\cup$  early_days;  
really_long_days := late_days  $\cap$  early_days;
```

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

- **For efficiency,**

- **max set size \leq 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:

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

-- q is a dangling ref.

C:

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

-- *p is a dangling ref.

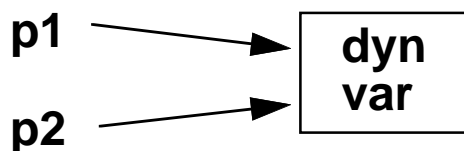
Preventing Dangling References

- **Tombstones**

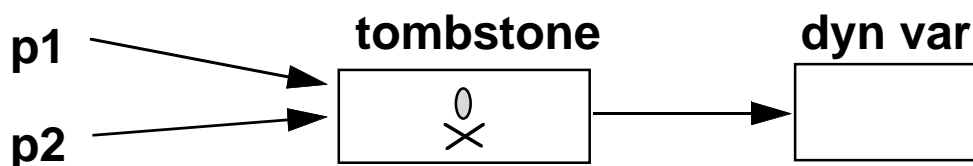
→ **Pointers can't point directly to a dynamic variable**

extra level of indirection called a *tombstone*.

without tombstone:



with tombstone:



Safe, but add space and time overhead

Preventing Dangling References (continued)

- **Locks and Keys**

- **Additional information stored with pointers and dynamic variables:**

- pointer \equiv (key, address)

- dynamic variable \equiv (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:**

```
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

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)**

Know where pointers are within cells

Fragmentation not a problem

Lisp

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