Data types

Contents

Primitive data types

Points

- Numeric types
- Booleans
- Characters

A data type is not just a set of objects. We must consider **all** operations on these objects. A complete definition of a type must include a list of operations, and their definitions.

Primitive data objects are close to hardware, and are represented directly (or almost directly) at the machine level—usually word, byte, bit.

Integer types

An integer type is a finite approximation of the infinite set of integer numbers {0, 1, -1, 2, -2, ...}.

Various kinds of integers:
signed—unsigned, long—short—small.

Hardware implementations of integers:
one's complement, two's complement, ...

Floating-point types

Finite approximations of the non-denumerable set of real numbers.

Precision and range of values are defined by the language or by the programmer.

Hardware implementations (used by floating-point processors): exponent and mantissa.

Boolean type

This is not supported by all languages (e.g. it's not available in PL/I, in C).
Values: true, false. Operations as in classical two-valued propositional logic.

Hardware implementation: a single bit or a byte (this allows more efficient operations.

Character types

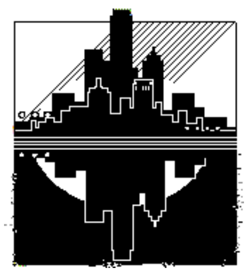
This is usually ASCII, but extended character sets are often used.

Accented characters éàü etc. should fit within ASCII, though one standard does not exist.
Chinese or Japanese character sets are examples of what's necessary but has much more that 256 elements.

Hardware implementation: a byte or several bytes.

Other primitive types

Word (e.g. in Modula-2), byte, bit (e.g. in PL/I), pointer (e.g. in C).



Structured data types

Points
- Strings
- Enumerated types
- Arrays
- Records
- Sets
- Pointers

Strings

A string is a sequence of characters. It may be:

- a special data type (its objects can be decomposed into characters)—Fortran, Basic;
- an array of characters—Pascal, Ada;
- a list of characters—Prolog;
- consecutively stored characters—C.

Syntactic form: characters in quotes. Pascal has one kind of quotes, Ada has two:

`'A'` is a character, `"A"` is a string.

Operations on strings as a separate data type:

STRING × STRING	→	STRING	concatenation
STRING × INTEGER × INTEGER	→	STRING	substring
STRING	→	CHARACTERS	decompose into an array or list
CHARACTERS	→	STRING	convert an array or list into a string
STRING	→	INTEGER	length
STRING	→	BOOLEAN	is empty?
STRING × STRING	→	BOOLEAN	equality, ordering

Specialized string manipulation languages (Snobol, Icon) include built-in pattern matching, sometimes very complicated. Examples: match the first occurrence of a small string in a large string, match a string with balanced parentheses, match any sequence of words.

The allowed length of strings is a design issue:

fixed-length strings—Pascal, Ada, Fortran;

variable-length strings—C, Java.

A character may be treated as a string of length 1, or as a separate data structure. In fact, many languages (Pascal, Ada, C, Prolog) treat strings as derivative—forms of arrays or lists. Such strings are implemented "in software". Operations on strings are the same as operations on arrays or lists of other types. For example, every character in a character array is available at once, whereas a list must be searched linearly.

A problem with variable-length strings: allocation of a maximum possible area is wasteful, allocation for single characters may be too frequent. An efficient strategy is to allocate blocks of (for example) 128 characters. All strings of 1-128 characters occupy one block, strings of 129-256 characters occupy two blocks etc.

Enumerated types (user-defined ordinal types)

[read Section 5.4]

We can declare a list of symbolic constants that are to be treated literally (they do not stand for other values as `const pi = 3.14;` does).

We also specify the implicit ordering of those newly introduced symbolic constants.

```
type day = (mo,tu,we,th,fr,sa,su);
```

Here, we have `mo<tu<we<th<fr<sa<su`.

Pascal supplies the programmer with three operations for every new enumerated type `T`:

`succ`: successor, e.g. `succ(tu)=we`

`pred`: predecessor, e.g. `pred(su)=sa`

(each is undefined at one end)

`ord`: position in the type, assuming that the first constant has position `0`.

For characters, Pascal also has `chr`, producing the character at a given position.

Ada makes this more complete:

`succ`: successor,

`pred`: predecessor,

`pos`: position,

`val`: constant at position.

Ada also supplies attributes, among them `FIRST` and `LAST`:

`day'FIRST=mo`, `day'LAST=su`.

A design issue: is the symbolic constant allowed in more than one type? In Pascal, no. In Ada, yes.

```
type stoplight is
     (red, orange, green);
type rainbow is
     (violet, indigo, blue, green,
      yellow, orange, red);
```

Qualified descriptions remove confusion: `stoplight'(red)` or `rainbow'(red)`.

Implementation of enumerated types: map the constants c_1, ..., c_k into small integers `0`, ..., `k-1`.

The role of enumerated types: increasing clarity and readability by separating concepts from their numeric codes.

Arrays

An array represents a mapping:

index_type → component_type

The *index type* must be a discrete type (integer, character, enumeration etc). In some languages this type is specified indirectly: `A(N)` in C means `0..N`, while in Fortran it's `1..N`.

There are normally few restrictions on the *component type* (we can even have arrays of procedures or files).

The form of references to array elements:

`A[I]` in Algol, Pascal, C; `A(I)` in Fortran, Ada.

Multidimensional arrays can be defined in two ways (we'll have only 2 index types, for simplicity):

$\boxed{\text{index\_type}_1 \times \text{index\_type}_2 \rightarrow \text{component\_type}}$

This corresponds to references such as `A[I,J]`.

$\boxed{\text{index\_type}_1 \rightarrow (\text{index\_type}_2 \rightarrow \text{component\_type})}$

This corresponds to references such as `A[I][J]`.

Operations on arrays

• select an element (get its value): `A[I]`

• select a slice of an array:
  (read the textbook, Section 5.5.7)

• assign a complete array to a complete array:

```
A := B;
```

There is an implicit loop here. (To assign a value to a component of an array, use whatever is appropriate for the component type.)

• compute an expression with complete arrays (this is possible in extendible or specialized languages):

```
V := W + U;
```

If `V`, `W`, `U` are arrays, this may denote addition of arrays. All three arrays must be compatible (the same index and component type), and addition is probably carried out element by element.

Subscript binding

• static:	fixed size, static allocation (this is done in older Fortran).
• semistatic:	fixed size, dynamic allocation (Pascal).
• semidynamic:	size determined at run time, dynamic allocation (Ada).

```
type day is
   (mo,tu,we,th,fr,sa,su);
type days is
   array(day range <>) of boolean;
cutoff := th;
-- ...
A: days (mo..cutoff);
B: days (cutoff..sa);
```

Ada instantiates the range dynamically. The two arrays are compatible.

• dynamic:	size fluctuates during execution, flexible allocation required (Algol 68, APL—both little used...)

Array-type constants and initialization

Many languages allow initialization of arrays to be specified together with declarations:

C
```
int vector [] = {10,20,30};
```

Ada
```
vector: array(0..2)
   of integer := (10,20,30);
```

Array constants in Ada

```
temp is array(mo..fr)of -40..40;
T: temp;
T := (5,12,8,30,25);
T := (mo=>5, we=>8, tu=>12,
      fr=>25, others=>30);
T := (5,12,8, fr=>25, th=>30);
```

Attributes of arrays in Ada

```
TM: temp;
TM'FIRST     mo
TM'LAST      fr
TM'LENGTH    5
TM'RANGE     mo..fr
```

Implementing arrays

The only issue is how to store arrays and access their elements—operations on the component type decide how the elements are manipulated.

An array is represented during execution by an array descriptor. It tells us about

• the index type,

• the component type,

• the address of the array, that is, the data.

Specifically, we need:

• the lower and upper bound (for subscript checking),

• the base address of the array,

• the size of an element.

We also need the subscript—it gives us the offset (from the base) in the memory area allocated to the array.

A multi-dimensional array will be represented by a descriptor with more lower-upper bound pairs.

Mapping multi-dimensional arrays into one-dimensional memory

(we'll do it in 2 dimension, for simplicity)

Row-major (the second subscript changes faster)

Column-major (the first subscript changes faster)

11	12	13	14	15
21	22	23	24	25
31	32	33	34	35

array [1..3, 1..5]

11	12	13	14	15	21	22	23	24	25	31	32	33	34	35

row-major

11	21	31	12	22	32	13	23	33	14	24	34	15	25	35

column-major

Suppose that we have this array:

```
A: array [LOW1..HIGH1,
          LOW2..HIGH2] of ELT;
```

where the size of `ELT` is SIZE.

The calculation is done for row-major (column-major is analogous). We need the base—for example, the address LOC of `A[LOW1, LOW2]`.

We can calculate the address of `A[I,J]` in the row-major order, given the base.

Let the length of each row in the array be:

ROWLENGTH = HIGH2 - LOW2 + 1

The address of `A[I,J]` is:

(I - LOW1) * ROWLENGTH * SIZE +

(J - LOW2) * SIZE + LOC

A final word on arrays: they are not supported by Prolog, Scheme, ML. An array can be *simulated* by a list (the basic data structure in Scheme and a very important data structure in Prolog).

- Assume that the index type is always `1..N`.
- Treat a list of N elements:

  $[x_1,\ x_2,\ \ldots,\ x_N]$ (Prolog)

  $(x_1\ \ x_2\ \ \ldots\ \ x_N)$ (Scheme)

  as the (structured) value of an array.

Records

A record is a *heterogeneous* collection of fields (components)—as opposed to *homogenous* arrays.

Records are supported by a majority of important languages, from Cobol through Pascal, PL/I, Ada, C (where they are called structures), Prolog (!) to C++. The syntax varies. This is Ada:

```
type date is
record
  day: 1..31;
  month: 1..12;
  year: 1000..9999;
end record;

type person is
record
  name: record
          fname: string(1..20);
          lname: string(1..20);
        end record;
  born: date;
  gender: (F, M);
end record;
```

Fields are distinguished by names rather than indices. While iteration on elements of an array is natural and very useful, iteration on fields of a record is not possible (why?).

A field is indicated by a qualified name, e.g.

```
X, Y: person;
X.born.day := 15;
  X.born.month := 11;
  X.born.year := 1964;
Y.born := (23, 9, 1949);
Y.name.fname(1..8) := "Smithson";
```

The name of the record variable can be omitted if we know what record is referred to, e.g. inside Pascal's `with` statement:

```
with X do
  born.day := 15;
  born.month := 11;
  born.year := 1964;
end;
```

Another style of qualified names (in Cobol):

```
day of born of X
```

Operations on records

Selection of a component—by field name.

Construction of a record from components—either from separate fields, or as a complete record (a structured constant).

```
D := (month => 10, day => 15,
      year => 1994);
D := (day => 15, month => 10,
      year => 1994);
D := (15, 10, 1994);
D := (15, 10, year => 1994);
```

Note that an array can also be assigned such a constant. Interpretation depends on context.

```
A: array(1..3)of integer;
A := (15, 10, 1994);
```

Assignment of complete records.

Comparison of records for equality (there is no standard ordering of records).

Ada allows default values for fields:

```
type date is record
  day: 1..31; month: 1..12;
  year: 1000..9999 := 1994;
end record;
D: date;   -- D.year=1994 now
```

There aren't many restrictions on field types. Any combination of records and arrays (any depth) is usually possible. A field could also be a file or even a procedure!

Records (actually, terms) in Prolog carry their type and components around:

```
date(day(15), month(10), year(1994))
person(name(fname("Jim"), lname("Berry")),
       born(date(day(15), month(10),
            year(1994)),
       gender(m))
```

If we can assure the correct use, this can be simplified by dropping one-argument functors:

```
date(15, 10, 1994)
person(name("Jim", "Berry"),
       born(date(15, 10, 1994),
       m)
```

Discriminated union

A union of types means a set of objects coming from those types—and operations on them. Though the types may be incompatible, this causes problems. While incompatible types cannot be bound to an object at the same time, this is possible at different moments in the lifetime of the object.

A discriminated union is a union in which the current type of the object can always be known. This is achieved by adding a special component to a union, called a tag (or discriminant). It indicates the type. An example of variant records in Ada:

```
type person (tag: status) is
record
  name: string(1..40);
  case tag is
    when married =>
      spouse: string(1..40);
    when single => null;
    when divorced =>
      alimony: integer;
  end case;
end record;
```

We assume that this type has been defined:

```
type status is
  (married, single, divorced);
```

Type `person` is a union of three types:

```
record
  name: string(1..40);
  tag: status;    -- tag=married
  spouse: string(1..40);
end record;

record
  name: string(1..40);
  tag: status;     -- tag=single
end record;

record
  name: string(1..40);
  tag: status;   -- tag=divorced
  alimony: integer;
end record;
```

We can operate correctly on a record with variants if we test the tag field.

```
if tag = divorced
then child_payment := alimony;
else child_payment := -1;
end if;
```

Incorrect use of variant records may lead to inconsistencies:

```
X: person;
X.name := "Smithson";
X.tag := married;
X.alimony := 1200;   -- ??!!
```

Ada makes such inconsistency impossible: only aggregates are allowed in assignments. For example, this structured constant would be considered incorrect:

```
("Smithson",married,1200)
```

That is, it's possible to check the types of all components against the types in the variant indicated by the tag value `married`.

This constant is correct:

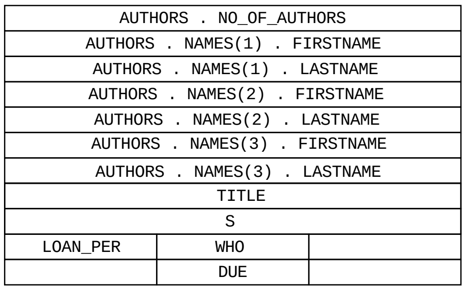
```
("Smithson",divorced,1200)
```

and can be assigned:

```
X := ("Smithson",divorced,1200);
```

Implementation of variant records

```
type book_status is (REFRNCE, CHCKD, REG);
type reader is ......
type book (S: book_status) is record
   AUTHORS: record
      NO_OF_AUTHORS: 1..3;
      NAMES: array(1..3) of
        record FIRSTNAME: string(1..20);
               LASTNAME: string(1..20);
        end record;
   end record;
   TITLE: string(1..80);
   case S is
      when REFRNCE => LOAN_PER: integer;
      when CHCKD => WHO: reader; DUE: date;
      others => null;
   end case;
end record;
```


AUTHORS . NO_OF_AUTHORS
AUTHORS . NAMES(1) . FIRSTNAME
AUTHORS . NAMES(1) . LASTNAME
AUTHORS . NAMES(2) . FIRSTNAME
AUTHORS . NAMES(2) . LASTNAME
AUTHORS . NAMES(3) . FIRSTNAME
AUTHORS . NAMES(3) . LASTNAME
TITLE
S
LOAN_PER
WHO
DUE

Sets

Sets (available only in Pascal and Modula-2) are restricted to scalar base types with few objects.

Declaration—an example:

```
set of char;
```

Operations on sets:
assignment, union, intersection, difference, membership, equality, inequality, inclusion.

Operators are `:=`, `+`, `*`, `-`, `in`, `=`, `<>`, `<=`, `>=`

Implementation of sets: as a bit map.

`set of day;`

`[mo, we]` $\longrightarrow$	**`1010000`**
`[tu, we, th]` $\longrightarrow$	**`0111000`**
`[mo, we] + [tu, we, th]` $\longrightarrow$	**`1111000`**
`[mo, we] * [tu, we, th]` $\longrightarrow$	**`0010000`**

Size restriction is usual, e.g., to 256 elements.

Pointers

[We're skipping Sections 5.10.3, 5.10.7, 5.10.10.]

A pointer variable has addresses as values (and a special address **`nil`** or **`null`** for "no value"). They are used primarily to build structures with unpredictable shapes and sizes—lists, trees, graphs—from small fragments allocated dynamically at run time.

A pointer to a procedure is possible, but normally we have pointers to data (simple and composite). An address, a value and usually a type of a data item together make up a variable. We call it an anonymous variable: no name is bound to it. Its value is accessed by dereferencing the pointer.

$p \longrightarrow \boxed{\quad \alpha}$ `value(p) = p^ = α`

$\alpha \longrightarrow \boxed{\quad 17}$ `value(p^) = 17`

Note that, as with normal named variables, in
`p^ := 23;`
we mean the address of `p^` (the value of `p`). In
`m := p^;`
we mean the value of `p^`.

A pointer variable is declared explicitly and has the scope and lifetime as usual. An anonymous variable has no scope (because it has no name) and its lifetime is determined by the programmer. It is created (in a special memory area called heap) by the programmer, for example:

```
   new(p);                  in Pascal
   p = malloc(4);           in C
```

and destroyed by the programmer:

```
   dispose(p);              in Pascal
   free(p);                 in C
```

If an anonymous variable exists outside the scope of the explicit pointer variable, we have "garbage" (a lost object). If an anonymous variable has been destroyed inside the scope of the explicit pointer variable, we have a dangling reference.

```
      new(p);
      p^ := 23;
      dispose(p);
      ......
      if p^ > 0 {???}
```

Producing garbage, an example in Pascal:

```
    new(p);  p^ := 23;  new(p);
```

{the anonymous variable with the value 23 becomes inaccessible}

Garbage collection is the process of reclaiming inaccessible storage. It is usually complex and costly. It is essential in languages whose implementation relies on pointers: Lisp, Prolog.

Pointers in PL/I are typeless. In Pascal, Ada, C they are declared as pointers to types, so that a dereferenced pointer (`p^`, `*p`) has a fixed type.

Operations on pointers in C are quite rich:

```
   char b, c;
   c = '\007';
   b = *((&c - 1) + 1);
   putchar(b);
```

Summary