Types

Based on the slides of Maurizio Gabbrielli

Data Type

Type: collection of values (homogeneous and effectively presented) equipped with a set of operations to manipulate these values

Types:

- Integer +, *, div,...
- Bool and, or,...
- Strings concat
- Records field sel/upd
- Int → bool ...

Not types:

- {3, True}
- True reals

What is a type and what is not strongly dependent on the programming language

What are the types for?

- Design level: organizing the information
  - Different type for different concepts
  - Comments/info on the intended use of identifiers
- Program level: identify and prevent errors
  - Types (and not comments) can be controlled automatically

    3 + "foo" is wrong
- Implementation level: allow some optimizations
  - Bool requires less bits than real
  - Pre-calculation of record/struct access offsets

Type systems

- Each language has a type system:
  - Predefined types
  - Mechanisms for defining new types
  - Rules for type checking:
    - Equivalence, compatibility, inference
    - Control: static (compilation) or dynamic
- High Level languages
  - Systems with expressive types
  - Elaborate checks
- Machine languages
  - Simple Systems
    - Integers and floating-point
  - No control (semantic errors may not be detected)

Type systems

- Type equivalence
  - When two expressions have the same type
- Type compatibility
  - When a type is allowed in a different context
    - 3 + 4.6 is it legitimate?
- Type inference
  - How to derive the type of a composite expression
- Type checking
  - What errors are detected and when
  - Language Strongly Typed: no unreported errors resulting from a type error can occur at run time

Static or dynamic checking

- Dynamic checking (e.g. Lisp)
  - Each data has a type descriptor
  - Before each operation you check the compatibility
    - (head x) → interpreter checks before x is a list
  - An error causes the execution to stop
- Static checking (compilation) (e.g. Pascal, Java)
  - More binding syntax
    - x = 4.6 illegal in Java if x is an int
  - Execution without types
  - An error causes the non-compilation

Static or dynamic checking

- Better static or dynamic?
  - Both prevent type errors
  - Dynamic control inefficient at run time
  - Static checking restricts flexibility
    - In static checking

```
int x;
x = if true then 3 + 4 else "foo";
```

is always rejected, while it is dynamically corrected

- In General it is impossible (undecidable) to statically know if a running error will occur
- Static checking is always "conservative"

Categories of types

- Scalar types (or simple types): Values not composed of aggregations of other values
- Composite types: obtained by combining other types with appropriate constructors

Scalar types

- Boolean
  - Values: True, False
  - Operators: or, and, not, conditionals
  - Repr: 1 bit
  - Note: C does not have a bool type

- Characters
  - Values: a, A, B, B,..., is, é, ë,; , ', ...
  - Operators: equality Code/decode; language-dependent
  - Repr: 1 byte (ASCII) or two bytes (UNICODE)

Scalar types

- Integers
  - Val: 0,1,-1.2,-2,..., maxint
  - Op: +,-, *, mod, Div,...
  - Repr: some bytes (2 or 4)
  - Notes: integers and long integers (also 8 bytes) → limited portability problems when length is not specified in language definition

- Real
  - Val: Rational values in a certain range
  - Op: +,-, *,/,...
  - Repr: some bytes (4); Floating
  - Notes: reals and long reals (8 bytes) → serious portability problems when the length is not specified in the language definition

Scalar types

- Complex
  - Val: ....
  - Op: ...
  - REPR: Two Reals
  - Notes: Scheme, Ada

- Fixed Point (for reals)
  - Val: Rational values in a certain range
  - Op: +,-, *,/,...
  - Repr: some bytes (2 or 4)

Scalar types

- The type `Void`
  - has only one value
  - No operation
  - Serves to define the type of operations that change the state without returning any value

```
void f (...) {...}
```

  - The type of f must have a value (and not none) otherwise we could not define such a f!
  - The value of f of type `Void` is always the same (and therefore does not interest us)

Enumerations

- Introduced in Pascal

```
Type days = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
```

- Easier to understand programs
- Ordered values: `Tue < Fri`
- Iterating over values: `For i: = Mon to Sat...`
- succ, pred
- represented as short integers (one byte)
- In C:

```
enum days = {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

  But it's equivalent to

```
typedef int days;
const days Mon = 0, Tue = 1, ..., Sun = 6;
```

- Pascal distinguishes instead `Tue` And `1`

Intervals (subrange)

- Introduced in Pascal
- Values are a range of values of an ordinal type (the *base type* of the interval)
- Example:

```
Type LessThanTen = 0..9;
Type WorkingDays = Mon..Fri
```

- Represented as the base type
- Why use a range type instead of its base type:
  - "Controllable" documentation
  - Efficient code generation

Composite types

- Record
  - Collection of fields, each of a (different) type
  - A field is selected with its name
- Varyant records
  - Records where only some (mutually exclusive) fields are active at a given instant
- Array
  - function from one index type (scalar) to another type
  - Array of characters are called strings (special operations)
- Set
  - Subset of a base type
- Pointer
  - Reference to an object of another type

Records

- Manipulate heterogeneous data in a unified way
- C, C++, Commonlisp, Algol68: `Struct`
- Java: had no record types, subsumed by classes
  - introduced in Java 16 (https://openjdk.java.net/jeps/395)
- Example, in C:

```
struct student {
    char name[20];
    int id; };
```

- Field selection:

```
student s;
s.id = 343536;
```

- Records can be nested

Varying records

- In a variation record, some fields are alternative: only one of them is active at any given instant

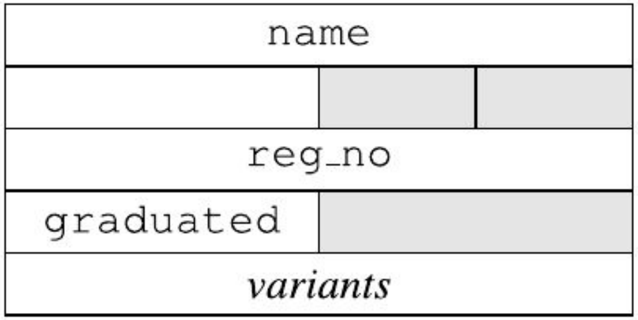
```
type Stud = record
    name : array [1..6] of char;
    reg_no : integer;
    case graduated : boolean of
        true: (lastyear : 2000..maxint);
        false:(major : boolean; year :
          (first,second,third))
end;
s.graduated := true
s.lastyear := s.lastyear+1
...
```

The two fields `lastyear` and `year` can share the same memory location

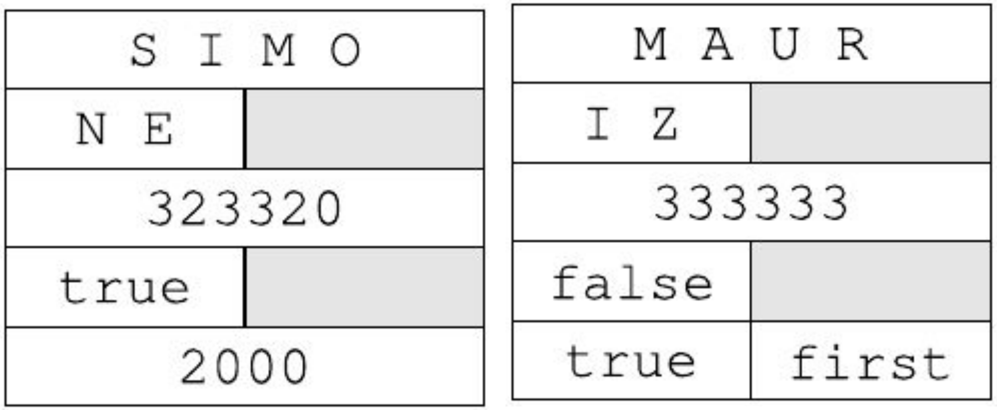
The type of `graduated` can be any ordinal type

Variant Records: Memory layout

```
type Stud = record
    name : array [1..6] of char;
    reg_no : integer;
    case graduated : boolean of
        true: (lastyear : 2000..maxint);
        false:(major : boolean;
         year : (first,second,third))
    end;
```



Two variables of type `Stud`



Varying records

- Possible in many languages
  - C: Union + struct

```
struct student {char name [6];
  int reg_no;
  bool graduated;
  union {
     int lastyear;
     struct {int year;
         bool major;} variantfields;
         }
  };
```

- Pascal (Modula, Ada) use unions and records

Variant Records: Issues

- The discriminating tag is an editable field with an ordinary assignment

  The following program is legal

```
var s: stud;
...
s.graduated := true;
s.lastyear := 2001;
s.graduated := false;
println(s.year);       /* Ohi, Ohi, Ohi! */
...
```

Array

- Collections of homogeneous data:
  - function from an index type to the type of the elements
  - index: generally discrete
  - element: "any type" (rarely a functional type)
- Statements
  - C: `int arr[30];` Index type: between 0 and 29
  - Pascal: `var arr: Array [0..29] of integer;`
- Multidimensional arrays
  - function from index type to array type
  - In Pascal the following are equivalent

```
var mat: array[0.. 29, ' a '.. ' Z '] of real;
var mat: array[0.. 29] of array [' a '.. ' Z '] of real;
```

  - But they are not equivalent in Ada: the second allows *slicing*
  - C: merges arrays and pointers (see later)

Array: Operations

- Main operation allowed:
  - Selecting an item: arr[3] mat[10,' C ']
  - Note that edit is not an operation on the array, but on the editable location that stores an element of the array
- Some languages allow slicing:
  - Selecting contiguous parts of an array
  - Example: in Ada, with

    `mat: array(1..10) of array(1..10) of real;`

    `mat(3)` refers to the third row of the square matrix `Mat`

Array: Shape

- Shape: The number of dimensions and range of the index. When is it fixed?
  - Static form.
    - All decided at compilation time.
    - Fixed memory that can be allocated at the entrance of the block.
    - Access to the element similar to local-variable access.
  - Form fixed at the time the processing of declaration
    - No longer known at compile time the offset on the activation record to access the variables
    - Activation record divided into fixed and variable part
  - Dynamic form.
    - The array can change shape after its creation
    - It is not possible to use the stack (activation record should dynamically change its size)
    - Heap used, with pointer in the activation record at the beginning of the array

Pointers

- Languages with modifiable variables introduce the possibility for a variable to "refer" to a given datum
    - In reference model languages (CLU, ML, Java) not needed: everything is a reference!
- Operations: dereferencing
    - C: * prefix   * a
    - Pascal: ^ Postfix   a^
- Operations: Allocate/deallocate objects
    - C: malloc
        - System call, size as argument;
    - Pascal: new

Pointers: High or low level?

- The type of "pointers to" is an abstraction over the locations
  - In many languages it remains (rightly) such
    - Pascal does not allow you to directly access or modify a pointer (new or assignments only)
- C allows direct access to the representation
  - Pointer arithmetic

```
Int *X;
X = (int *) malloc (sizeof (int));
X = X + 1;
```

After the increment X points to the next word to the one allocated (in this case + 4 bytes): The compiler knows the size of the object pointed

Pointers and Arrays in C

- Arrays and pointers are interchangeable in C

```
int n;
int *a;        // pointer to ingegers
int b[10];     // array of 10 integers
…
a = b;          // a points to the first element of b
n = a[3];       // n takes the value of b[3]
n = *(a+3);     // the same
n = b[3];       // the same
n = *(b+3);     // the same
```

- But remember: `a[3]=a[3]+1;`

  will also change `b[3]` (it's the same thing!)

Recursive type

- a value of the type can contain a reference to a value of the same type

```
type int_list: { int:val;
  int_list next;};
```

- usually represented as structures in the heap