Structuring the Computation

1. Objectives

- To control flow among the different constructs in a program.
-

- This is key to understanding the ***semantics*** of the language

- Type error can be classified into two categories:
    - Application errors
    - Language errors.

- Language errors:
    - Here we only address ***language errors***, as opposed to ***application errors***
    - Language errors are **programmer errors** in how a language's syntax and semantics are defined.

    - There are two categories
        - Syntactic
        - Semantic

- **Type Safety**: guarantee no type errors

- Another Classification:

    - Static:
        - During compilation
        - It is preferred.

        - Not all errors can be detected at compile time, e.g., memory availability and division by zero.

    - Dynamic:
        - At run time.
        - Slow down program execution.

2. Strong Typing & Type Checking

- A type system is said to be strong if it guarantees not to generate type errors.

- **Strongly typed languages**: Languages that have strong type systems.

- **Weakly typed languages**: languages that are not strongly typed.

- Coercion rules weakens the value of strong typing

- Static Type System:

    - The type of each expression must be known at compile time.
    - Requirements of a static type system include:
        - Only built-in types can be used
        - All variables are declared with an associate type.
        - All operations are specified by stating the types of the required operands and the type of the results.

- **Statically Typed** Languages are **Strongly Typed** Languages.

3. Type Compatibility

- A compatible type is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler-generated code, to a legal type.

- Type compatibility is also called conformance or equivalence.

- There are two type compatibility methods:
    - **Name compatibility** (also called strict compatibility):
        - Two variables can have compatible types only if they are in either the same declaration or in declarations that use the same type name.
        - It is highly restrictive.
        - It is easier to implement.
        - Ada uses name compatibility.

    - **Structure type compatibility**:
        - Two variables have compatible types if their types have identical structures.
        - It is more flexible.
        - It is difficult to implement: compare the whole structure instead of just names.
        - Two types are structurally compatible if:
            1. T1 is name compatible with T2; or

2. T1 and T2 are defined by applying the same type constructor to structurally compatible corresponding type components.

- Examples:
    1. Two records or structure types compatible if they have same structure but different field names?
    2. Two single-dimensioned array types in a Pascal or Ada program are compatible if they have the same element type but have different subscript ranges?
    3. C uses structural compatibility except for structures.

4. Type Conversion

- It is needed when the types of an expression are not compatible.
- Some programming languages allow automatic conversions (Also called **coercion)**:

What would the following C program output?

```
#include <stdio.h>

main(){

    char c1 = 'a';
    int i = 10;
```

```
        i += c1;
        printf(" The value of i is:%d\n",i);
}
```

- Formally, the conversion of a type T_1 to another type R_1 is defined as follows:

$$func: T_1 \rightarrow T_2$$

For example:

$T_1\ x_1;$
$T_2\ x_2;$

$x_2 = func(x_1);$

- How to use func to convert from another type T_2 to another type R_2?

Requirements: Two new functions t_{21} and r_{12}:

$t_{21}: T_2 \rightarrow T_1$
$r_{12}: R_1 \rightarrow R_2$

Let $\mathbf{x}_2$ be a variable of type $\mathbf{T}_2$ and $\mathbf{y}_2$ be a variable of type $\mathbf{R}_2$, then apply $\mathbf{t_{21}}$ to $\mathbf{x}_2$, evaluate **func**, apply $\mathbf{r}_{12}$ to the result, and assign it to $\mathbf{y}_2$:

$y_2 = r_{12}\ (\ func(\ t_{21}(\ x_2\ )\ )\ );$

5. Type and Subtypes

- A ***type*** is defined by:
  - A set of values
  - A set of operations

- A ***subtype*** ST of a basic type T (also called parent type or supertype) can be defined as
    - A subset of the values of T
    - Assume that operations of T are ***inherited*** by ST

- A language supporting subtypes must provide:
    - A way to define subsets of a given type, and
    - Compatibility rules between a subtype and its supertype
    - Example: Ada subtype

```
SUBTYPE Day_Range  IS Integer RANGE 0..31;
SUBTYPE Year_Range IS Integer RANGE 1901..2099;
SUBTYPE natural IS integer range 0..integer'last;

SUBTYPE natural IS integer RANGE 0..integer’last;
SUBTYPE positive IS integer RANGE 1..integer’last;
```

6. Generic Types

- It allows functions and ADT to have generic types

- How type-checking is done to guarantee type safety?

    - Bind generic types to concrete types at compile-time instantiation.

- Example:

push: stack(T)xT $\rightarrow$ stack(T)
pop: stack(T) $\rightarrow$ stack(T)xT
length: stack(T) $\rightarrow$ int

7. Monomorphic versus Polymorphic

- Monorphic: It is a strict type system.

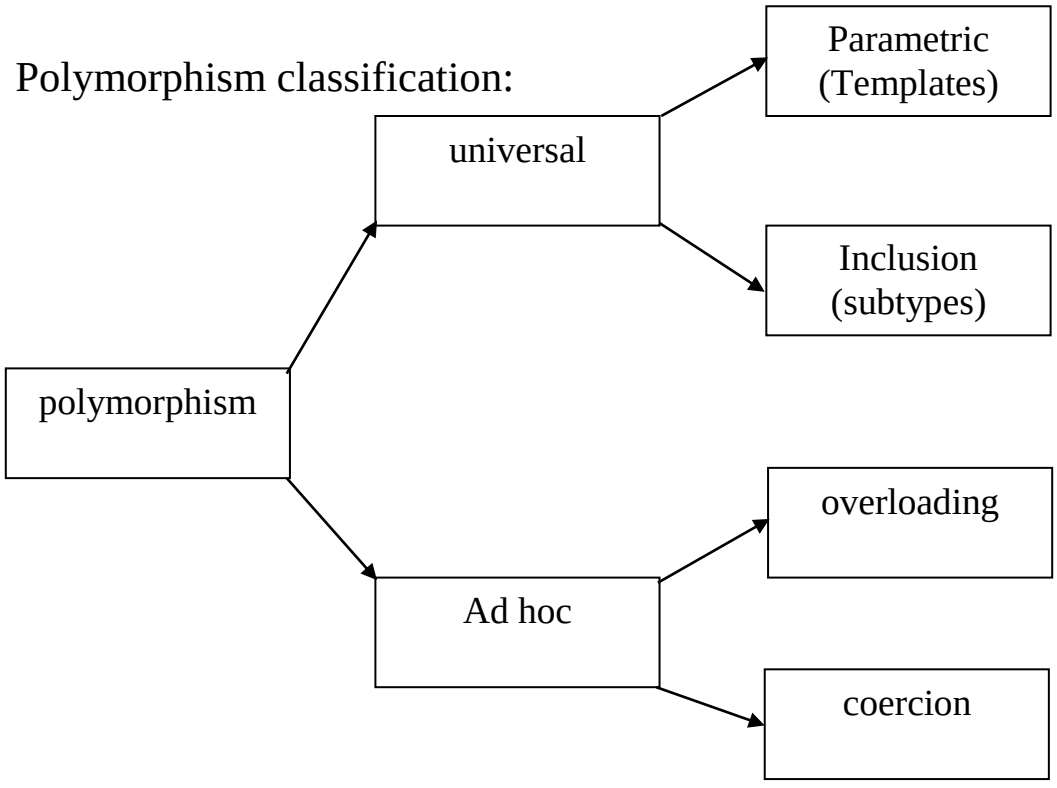
- Polymorphic: It is the case where an object can have more than one type

- Most practical programming languages have some degree of polymorphism.

- Languages deviates form strict monomorphic if they include one of the following:

    - Type equivalence
    - Coercion
    - Subtyping
    - Overloading

- Polymorphism classification:



- **Ad hoc** : functions work on a finite small set of types and may behave differently for each:
  - Overloading
  - Coercion

- **Universal**: work uniformly for an infinite set of types, all of which have common structure
  - Parametric
  - Inclusion

8. The Type Structure of Representative languages

- Java:
  - Computation unit is a class
  - User class construct to create new types

- C++
  - Uses the **class** construct to declare new types
  - Templates

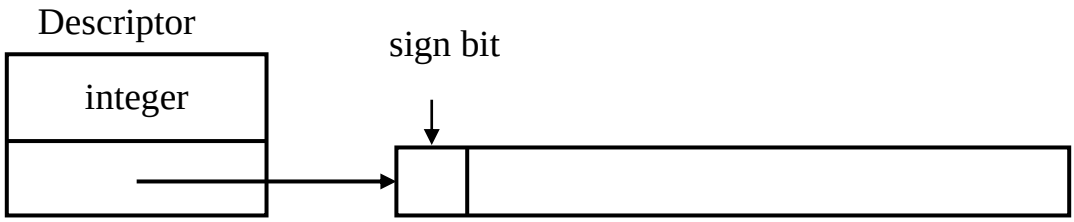
- Ada
  - Subtypes do not constitute new types, so parent-type operations still apply
  - Generic packages

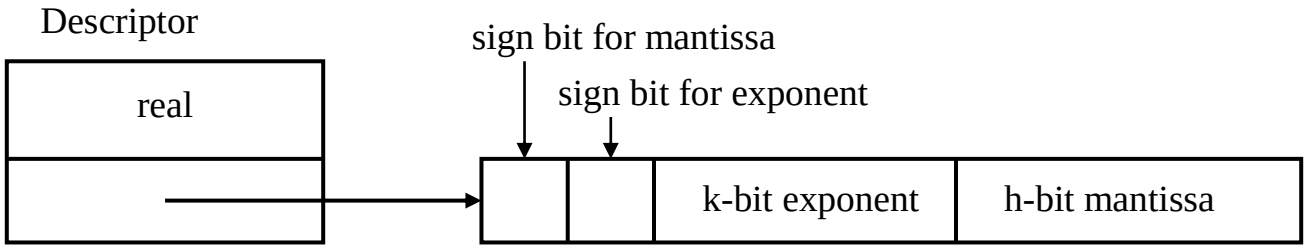
9. Implementation Models

- How to represent data objects inside the machine
- The implementation of each data object requires two elements:

- Descriptor: structures used to store the attributes of the data
- Data object: memory locations to store the actual values

- Built-in Primitive types:
  - Integers and reals are supported on most computers
  - They may also provide different sized versions of these types
  - Representations:

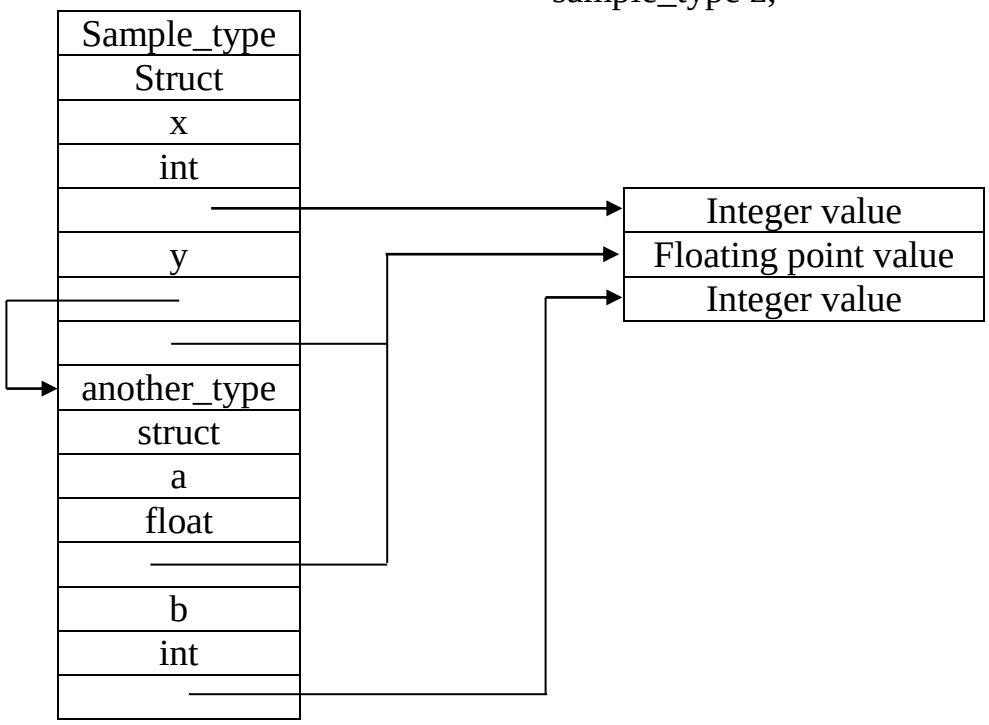




10. Implementation of Structured Types

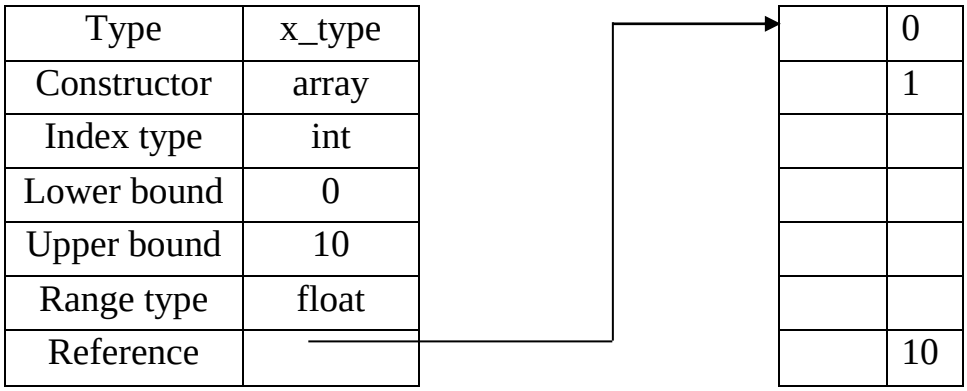
- Cartesian product:
  - It is a sequential layout of components.
  - The descriptor contains the following:
    - The Cartesian product type name and
    - For each field:
      - Name of the selector
      - Type of the field
      - Reference to the data object
  - Example:

```
type another_type is struct{
        float a;
        int b;
};
type sample_type is struct{
        int x;
        another_type y;
};
sample_type z;
```



- Finite Mapping
  - Allocate storage units for each element component.
  - The descriptor contains the following:
    - The mapping type name
    - The name of the domain type
    - Values for the upper and lower bounds.
    - The name of the range type
    - A reference to the first location of the mapping
  - Example:

    type x_type is float array[0..10];
    x_type x;

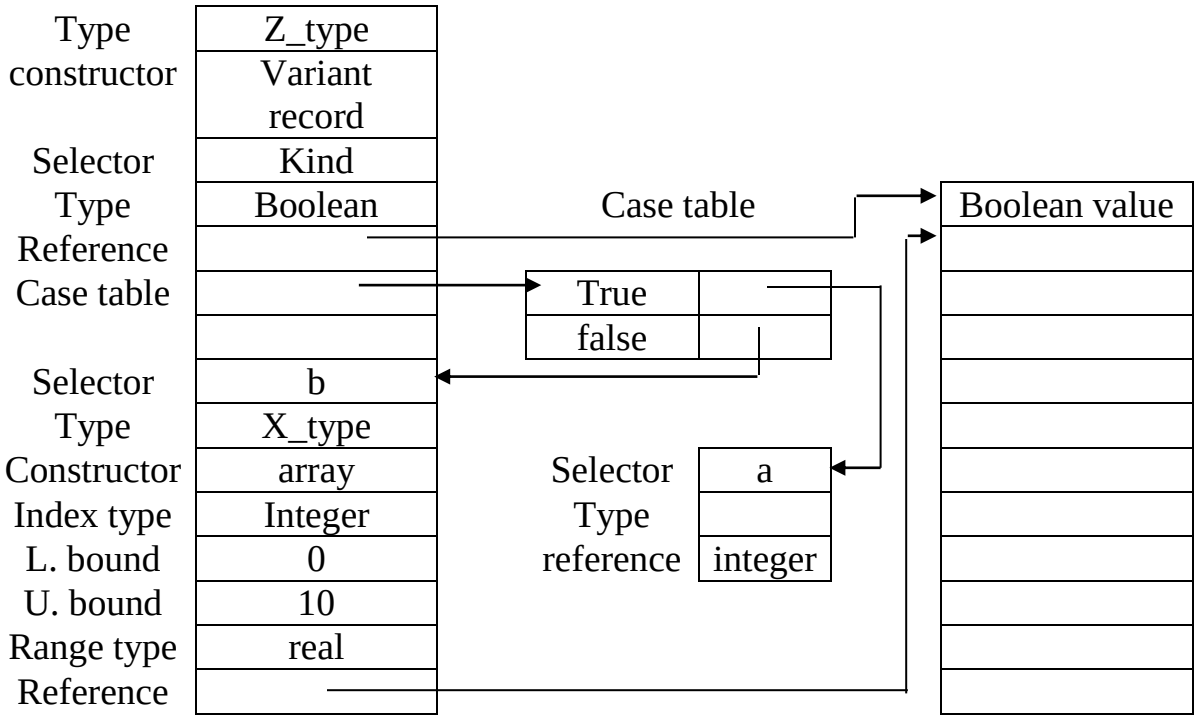


Type	x_type
Constructor	array
Index type	int
Lower bound	0
Upper bound	10
Range type	float
Reference	

	0
	1
	10

- Union and discriminated union
  - The descriptor of a variable of a union type is the sequence of the descriptors of each component.
  - Example:

```
type x_type = array[0..10] of real;
type z_type = record
                case kind: boolean of
                    true: (a:integer);
                    false: (b:x_type);
                end;
```



- Powerset
  - Use bit-mapping to represent a set
  - Use AND for Intersection and OR for union
  - Example:

S = {1, 3, 10, 16}

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	10	0	0	0