Types equivalence

Based on the slides of Maurizio Gabbrielli

Equivalence and compatibility between types

- Two types T1 and T2 are equivalent if each object of type T1 is also an object of type T2 and vice versa
- T1 is compatible with T2 if an object of T1 can appear where a T2 would be expected
- Equivalent?

```
struct {
        int a, b;
    }
```

```
struct {
        int a;
        int b;
    }
```

```
struct {
        int b;
        int a;
    }
```

```
array[1.. 10] of char;
```

```
array [1..2*5] of char;
```

```
array [0..9] of char;
```

Structural equivalence

- Used in ALGOL 68, Modula-#, C, ML
- Two types are equivalent if they have the same structure:
  - Examples struct 1 and 2 are equivalent
  - Example 3: depends on the language
- Structural equivalence: low level, does not respect the abstraction that the programmer uses with the name:

```
type stud = record
   age: integer;
   rating: string:
end
```

```
type wine = record
   age: integer;
   rating: string:
end
```

```
var s:stud;
    v:wine;
...
s:= v;
```

Structural Equivalence Definition

- Structural equivalence between types is the (minimum) equivalence relationship that satisfies the following three properties:
  - A name type is equivalent to itself;
  - If a type T is introduced with a definition

    type T = expression,

    T is equivalent to expression;
  - If two types are constructed by applying the same type constructor to equivalent types, then they are equivalent

Equivalence by name

- Two types are equivalent if they have the same name
- Used in Pascal, Ada, Java
- Loose equivalence by name (Pascal, Modula-2)
  - A declaration of a type alias does not generate a new type, but only a new name:

```
type A = record..... end
type B = A;
```

    - A and B are two names of the same type.

Compatibility

- T is compatible with S when T objects can be used in a context where S values are expected
    - Example: int n; float r; r = r + n;
- The definition depends critically on the language! T is compatible with S if
    - T and S are equivalent;
    - The values of T are an underset of the values of S (range);
    - All operations on S-values are also possible on the values of T (records defined differently);
    - The values of T correspond canonically to some S values (int and float);
    - The values of T can be matched to some S values (float and int with truncation);

Type conversion

- If T compatible with S, however, some type conversion is needed. Two main mechanisms
  - Implicit conversion (also called **coercion**): The abstract machine inserts the conversion, without any trace of it at the linguistic level;
  - Explicit conversion, or cast, when the conversion is indicated in the program text.

Compatibility and coercion

- Coercion is used to indicate a compatibility situation and what the implementation should do.
- Three possibilities. The types are different but:
  - With same values and same representation.

    Example: Structurally equal types, different names
    - conversion to compile-time only; no code
  - Different values and representation.

    Example: integers and reals.
    - Conversion code
  - Different values, but same representation at the intersection.
    Example: ranges and integers
    - code for dynamic control over membership at the intersection

Cast

- In certain contexts the programmer must insert explicit type conversions (cast in C and Java)
  – Annotations in the language that specify that a value of a type should be converted to another type.

```
S s = (S) t
```

```
r = (float)n;
n = (int) r;
```

- Not all explicit conversion allowed
  – Only those which the language knows how to implement the conversion.
  – You can always insert a cast where there is a compatibility (useful for documentation)
- Modern languages tend to favor casts than coercions

Type inference

- Infer the type of an expression from the type of the components
- In principle simple, but compatibility, equivalence, casting make things difficult