CS 206

Lecture 16 – Types continued

Type Checking: Basics

- Checks that each operator is applied to arguments of the right type. It needs:
- **Type inference**, to infer the type of an expression given the types of the basic constituents
- **Type compatibility**, to check if a value of type A can be used in a context that expects type B
    - Coercion rules, to transform silently a type into a compatible one, if needed
- **Type equivalence**, to know if two types are considered the same

Type Equivalence

- **Structural equivalence**: unravel all type constructors obtaining type expressions containing only primitive types, then check if they are equivalent
- **Name equivalence**: based on declarations
  - Name equivalence more popular today

```
type R2 = record
    a, b : integer
end;
```

```
type R3 = record
    a : integer;
    b : integer
```

```
type R4 = record
    b : integer;
    a : integer
end;
```

```
type student = record
    name, address : string
    age : integer

type school = record
    name, address : string
    age : integer

x : student;
y : school;
. . .
x := y;                 – – is this an error?
```

Type Compatibility and Coercion

- Type compatibility rules vary a lot
    - Integers as reals OK
    - Subtypes as supertypes OK
    - Reals as integers ???
    - Doubles as floats ???
- Coercion : ?
    - When an expression of type A is used in a context where a compatible type B is expected, an automatic implicit conversion is performed

Type Conversions (Type Casts)

- types are structurally equivalent, but language uses name equivalence
  - types employ the same low-level representation, and have the same set of values
  - conversion is therefore a purely conceptual
- types have different sets of values, but the intersecting values are represented in the same way
  - If the provided type has some values that the expected type does not, then code must be executed at run time to ensure that the current value is among those that are valid in the expected type
- types have different low-level representations
  - Need to change the representations – float to int / vice-versa

Type compatibility and Coercion

- Coercion may change the representation of the value or not
  - Integer → Real *binary representation is changed*

    ```
    {int x = 5; double y = x; …}
    ```
  - A → B subclasses *binary representation not changed*

    ```
    class A extends B{ … }
    {B myBobject = new A(…); … }
    ```
- Coercion may cause loss of information, in general
- In statically typed languages coercion instructions are inserted during semantic analysis (type checking)

Built-in types

- Typical built-in primitive types –
  - Boolean – {True, False}
  - Character -- {…, 'A' , …, 'Z' , …, '0' , …, '9' , …} ← PL- or implementation-defined set of characters (ASCII, Unicode etc.)
  - Integer -- {…, –2, –1, 0, +1, +2, …} ← PL- or implementation-defined set of whole numbers
  - Float -- {…, –1.0, …, 0.0, +1.0, …} ← PL- or implementation-defined set of real numbers
- In some PLs (such as C), booleans and characters are just "small" integers

Some more Terminology

- Discrete types – countable
  - integer, boolean, char
  - enumeration (`enum` day_of_week{Sunday, Monday, Tuesday…, Saturday})

- Scalar types - one-dimensional
  - discrete
  - real

Composite types

- Types whose values are composite, that is composed of other values (simple or composite). Examples
  - records (unions)
  - Arrays (Strings)
  - algebraic data types
  - Sets
  - Pointers
  - lists
- Most of them can be understood in terms of a few concepts:
  - Cartesian products (records)
  - mappings (arrays)
  - disjoint unions (algebraic data types, unions, objects)
  - recursive types (lists, trees, etc.)
- Different names in different languages
- Defined applying type constructors to other types (eg struct, array, record,…)

Overview of Composite types

- Composite type constructors can be mapped to a few mathematical concepts

- Cartesian products (records)
- mappings (arrays)
- disjoint unions (algebraic data types, unions)
- recursive types (lists, trees, etc.)

Cartesian Products

- In a Cartesian product, values of several types are grouped into tuples.
- Let (x, y) be the pair whose first component is x and whose second component is y.
- S × T denotes the Cartesian product of S and T:

$$S \times T = \{ (x,y) \mid x \in S, y \in T\}$$

- Cardinality:

$$\#(S \times T) = \#S \times \#T$$

Cartesian Products

- We can generalize from pairs to tuples

Let $S_1 \times S_2 \times \ldots \times S_n$ stand for the set of all n-tuples such that the i^{th} component is chosen from S_i :

$$S_1 \times S_2 \times \ldots \times S_n = \{ (x_1, x_2, \ldots, x_n) \mid x_1 \in S; x_2 \in S; \ldots; x_n \in S \}$$

- Basic operations on tuples
  - **construction** of a tuple from its component values
  - **selection** of an explicitly-designated component of a tuple

- Records (Ada), structures (C), and tuples (Haskell) can all be understood in terms of Cartesian products

Example: Ada records

- Type declarations:

```
type Month is (jan, feb, mar, apr, may, jun, jul, aug, sep,
oct, nov, dec);
type Day_Number is range 1 .. 31;
type Date is record
    m: Month;
    d: Day_Number;
    end record;
```

- Application Code

Record construction

```
someday: Date := (jan, 1);
 …
put(someday.m+1); put("/"); put(someday.d);
someday.d := 29; someday.m := feb;
```

Component Selection