Types

CS 152 Lecture

Types & Values
A common idea in most PLs

- Values are grouped into types
  - Integers {-2,3,4,-88,...}
  - Characters {'c', 'r', ...|,
  - Boolean {true, false}

- A type is a set of values
  - {-MAX,...0, ... +MAX} i.e. -2147483648 to 2147483647
  - {0,1,2,... 4,294,967,295} = unsigned Int
- Due to representation constraints, it is in practice, a **finite** set in a **programming environment**
- **Can you write programs to find out the ranges of types known to you?**

Sizes of Types: Examples
The width is limited

- No. of Bytes
  - unsigned int: 4
  - short int: 2
  - Int: 4
  - long long int: 8
  - float: 4
  - double: 8
  - Char: 1
- Java byte type: -128 to 128
- Again restating:
  - **Type is a set of values, you can also find the cardinality of the exact set from the representation scheme**

Values and Types

- Value **v** : Member of the set
- Type **T :** The set

we say, $\mathbf{v \in T}$

- **Primitive types** – cannot be decomposed further, standard and user defined
  - e.g. Int, char, enum
  - Enumerated types: enum Weekday {mon,tue,wed,thu,fri,sat,sun};
    - 0 to 6 only, the values are called ***enumerands****, and types are also called* ***enumerations***
- **Composite types** – structures, classes, functions, tables

Type Error

- Occurs during operations
  - Assigning a value of invalid type
    - Incopatible LHS and RHS types
  - Accessing a value that is undefined

```
int main () {
    enum Weekday {mon,tue,wed,thu,fri,sat,sun};
    Weekday d1, d2;
    d1 = wed;
    d2 = sat;
    //  d2 = 11;  // not allowed – typing error, is it detected?
    cout << d1 << "  " << d2 << endl;
    int A[10];
    cout << A[11] << endl; // typing error, but is it detected?
}
```

Composite Types

- Cartesian Products
  - S X R
  - S and R are two types
  - The cardinalities: #(SXR) = #S x #R
  - Struct Person {

    Int id;

    Int age;

    };
  - Int X Int
    - What is the cardinality of type Person?
    - i.e. How many different values can it have?
      - Are they infinite or are they finite?

Composite Types

- Mappings
  - M : S --> T
  - Mapping M maps set S to set T
  - Given a value s of set S, we have a corresponding alue for is t from set T
  - **bool f (bool b) {...}**

    S is boolean, and T is also boolean, we have

    F={ {(t -> t), (f -> t) },

    {(t -> t), (f -> f)},

    {(t -> f), (f -> t)},

    {(t -> f), (f -> f)}

    }

Composite Types

- Mappings: function types
- Cardinality
  - The elements of the value set corresponding to a function type are all possible mappings for a given function signature.
  - $\#(S \rightarrow T) = \#T^{\#S}$
- A function body is merely one of the many possible values for the function type.
- cardinality of a function type is the number of discrete function bodies (i.e. mappings) for the function type.
- Thus we can represent a function body as a value of a function type, or in other words, a program is a value and its specification, a type.

Composite Types

- Mappings: Arrays
- int A[10]
- It can be modeled as a function that maps integers from range 1..10 --> int
- So type of array A is
  - S-->int, where S={1..10}
- Default initializer is the default mapping.
- Cardinality of an array type represents the number of possible values of the array
  - e.g. 1111111111 is one of the many possible value
- With the assignment operator, we change this mapping itself!

Recursive Composite Types

- Some types are defined recursively in order to express the types in terms of closed expressions even if there are infinitely many possible values for them.
- For example,
  - a list L of elements of type T:
    - L = either NULL or T X L
  - or in other words,
    - L = NULL | (T X L), where | defines a disjoint union

Recursive Types:List

Example:

T = {a,b,c,d,e,f,....}

A value L1 of type List<T> = **abcdec**

The above value can be shown to be a valid value of list type by constructing a terminating **recursive expansion** for the value as given below:

L = a X L

X b X L

X c X L

X d X L

X e X L

X c X L

X NULL

Disjoint Union Type

- union U {

  int i;

  char c;

  }
- U = int + char
  - i.e.
    - U = either int or char
  - A value of type U is either a value of type int or a value of type char.
- Note that the union type (either/or) was used in the definition of the list type

When can a value of type T1 be safely treated as a value of type T2?

- Firstly, if T1 and T2 are the same types, then there is no problem.
  - For example as in the following statement
    - int i; int j; ... i = j;
- Further, if T1 and T2 are not the same types, we may still be able to treat ALL values of T1 as values of T2 provided that there is some such relation between the two types.
  - What's that **relation**?

Subtype Relation
A type and its subtype

- **S <: T**
- we say that type S is a subtype of type T
- For primitive types, a subset can be considered as a subtype.
  - Exmples:
    - E1 = {1,2,3,4}
    - E2 <: Int
    - E2 = {'a','b','c','D','E'}
    - E2 <: Char
- What can we **do** with subtypes?

Subsumption due to Subtyping

- We can use a value of a subtype wherever a value of the (super) type is expected.
- This is stated by the following **rule of subsumption**.

$$\frac{t:S,\ S<:T}{t:\ T}$$

The rule states this: if value t is of type S, and S is given as a subtype of type T, then value t is also **quite safely** a value of type T

Subtype relation for primitive types

- For primitive types, subset is subtype.
  - e.g. S={1,2,3}, T={1,2,3,4}, S<:T
- wherever value of a type is expected, a value from the subtype will work safely.
- i.e. a call to function
  - f(T val) {.....}
  - will work correctly with any value of type S sent as a parameter,
  - since all values of type S happen to be valid values of type T

Subtype Relation for Product types: The **width rule**

- R1 = T1 X T2
- R2 = T1 X T2 X T3
- R2 can be considered as a subtype of R1
  - why?
    - Because a value of type R2 can be easily considered as a value of R1 by ignoring the T3 component in it.
- Example:
  - R1 = RollNo X Name
  - R2 = RollNo X Name X Age

Subtype Relation for Product types: The **depth rule**

- R1 = T1 X T2
- R2 = S1 X S2
- R2 can be considered as a subtype of R1, when S1 <: T1 and S2 <: T2
  - R1 = String X String
  - R2 = RollNo X Name

Ssubtyping in Function Types

- Applying the rule to functions g and f,
  - if (type of g) <: (type of f),
    - we can use g safely wherever type of f is expected

```
int f (int x) {..}
Main () {
   int v, x;
    ...
    x = f (v);
     ..
}
```

- In the above program when can we use another function g in place of int f(int) in a **type-safe** manner?

Type Subtype Rule for Functions

- input parameters contravariant
- output result covariant
- T2 f (T1)
- S2 g (S1)

  g <: f

  T1 <: S1

  S2 <: T2
- Why so?
- How much of it is supported in C++/Java/C/other languages?

Overloading

- 10 + 2.3
- 10+ 2
- 2.3 + 10
- 2.3 + 2.4

Operator '+' is a function

- \+ : T1 X T2 --> T3

- What are T1, T2 and T3 types?

Eliminate Overloading by **Coercion**

- A language may use a single function
  float + float --> float

.. and implicitly type-cast integers to floats and back if needed

- The process of implicit type casting: 'coercion'

- int i = 10 + 2 will work correctly as
  - int i = (int) **((float) 10 + (float) 2)** with the typecasts implicitly done.
- But int i = 10 + 2.3 will result in loss of accuracy since the LHS type has been chosen incorrectly as int.

Overloading + coercion

use two overloaded functions

float + float --> float
int + int --> int

.. and implicitly type-cast (coerce) integers to floats and back if needed

Full Overloading

use four overloaded functions, and select the right one

float + float --> float
int + int --> int
int + float --> float
float + int --> float

Subtyping in OO Paradigms

As we already know,
It happens through superclass subclass relationship

Where an object of superclass is expected,
An object of subclass can be used

Accordingly the member function subtyping is also exercised

C++ allows subsumption via pointer types