Type Systems

- As we saw previously, any programming language that has some complexity to it allows us to create syntactically correct statements that semantically do not make any sense:

```
declare z (x) return x+1;

put z+1; // ???
```

Chap 11

- The error in the expression can easily be caught by an interpreter or compiler by tagging the operands with *type names*: z.{function} + i.{int}
- Now it is simple for the language processor to find the problem: it is only allowed to apply addition to {int} terms, e.g., j.{int} + i.{int}

Type Systems

- A principled approach to tagging terms and expressions with type names is called a *type system*
- Every modern programming language has one

Why do we use type systems?

- Types allow the language system to assist the developer in writing *better programs*. *Type mismatches* in a program usually indicate some sort of *programming error*.
  - Static type checking – check the types of all statements and expressions at compile time.
  - Dynamic type checking – check the types at runtime.
- Languages with a static type system can be type checked dynamically and statically
- Languages with a dynamic type system can only be type checked dynamically
- New research: gradual typing – type check as much as possible statically and then do the rest dynamical.

Types

A Type is a Set of Values

Consider the statement:

```
int n;
```

Here we declare n to be a variable of *type* int; what we mean, n can take on any value from the *set of all integer values*.

Also observe that the elements in a type share a common representation: each element is encoded in the same way (float, double, char, etc.)

Also, all elements of a type share the same operations the language supports for them.

Types

Def: A *type* is a set of values.

Def: A *primitive type* is a type a programmer can use but not define.

Def: A *constructed type* is a user-defined type.

Example: Java, primitive type

float q;

type float ⇒ set of all
possible floating point values

q is of type float, only
a value that is a member
of the set of all floating point
values can be assigned to q.

Types

Example: Java, constructed type

```
class Foobar { int i; String s; };

Foobar c = new Foobar();
```

Now the variable c only accepts values that are members of type Foobar;
☞ *object instantiations* of class Foobar; objects are the values of type Foobar..

Types

Example: C, constructed type

int a[3];

the variable a will accept values
which are arrays of 3 integers.

e.g.: int a[3] = {1,2,3};
int a[3] = {7,24,9}

We will have more to say about this later on.

Subtypes

- We saw that the notion of a type as a set of values is a nice model for explaining variable declarations and object-oriented structures
- But it is also essential to developing the notion of a *subtype*

Subtypes

Def: *a subtype* is a *subset* of the elements of a type.

Example: Java

'Short' is a subtype of 'int', that is, all the values in set 'short' are also in set 'int': short $\subset$ int

Example: Java

'Float' is a subtype of 'double' (all the values in set 'float' are also in set 'double)': float $\subset$ double

Observations:

(1) converting a value of a subtype to a value of the supertype is called a *widening* type conversion. (safe)

(2) converting a value of a supertype to a value of a subtype is called a *narrowing* type conversion. (not safe - information loss)

Subtypes

Consider this example in Java with an implicit *narrowing* conversion:

```
int i = 33000;
short j = i;        //problematic, short is only 2 bytes, overflow!
```

On the other hand this example in Java with an implicit *widening* conversion has no problems:

```
short i = 20000;
int j = i;
```

☞ Compilers/interpreters will often insert widening conversions but will flag errors when a supertype needs to be converted to a subtype.

Subtypes

- An important implication of subtypes in programming languages is the notion of *type hierarchies*
- Here the types of a language are ordered along the subtype relation, e.g. in Java
  - int $\subset$ float $\subset$ string

Type Equivalence

I. Name Equivalence – two objects are of the same type of and only if they share the same *type name.*

Example: Java

```
Class Foobar {
  int i;
  float f;
}
```

```
Class Goobar {
  int i;
  float f;
}
```

```
Foobar o = new Goobar();
```

Error; even though the types look the same, their names are different, therefore, Java will raise an error.

☞Java uses *name equivalence*

Type Equivalence

II. Structural Equivalence – two objects are of the same type if and only if they share the same *type structure*.

Think of this as:

```
class Person {
  int age;
  int weight;
  String mstatus;
  String profession;
}
```

Example: ML

```
- type person = int * int * string * string;
- type mytuple = int * int * string * string;
- val joe:person = (38, 185, “married”, “pilot”):mytuple;
```

Even though the type names are different, ML correctly recognizes this statement.

☞ ML uses *structural equivalence*.

Polymorphism

- An interesting implication of type systems is *polymorphism*:
  - Function overloading
  - Subtype polymorphism

Def: A function is *polymorphic* if it has at least two possible types.

polymorphism ≡ comes from Greek meaning 'many forms'

Polymorphism

<u>Function Overloading</u>

<u>Def:</u> An *overloaded function* is one that has at least two definitions, all of different types.

Example: In Java the ‘+’ operator is overloaded.

String s = “abc”.{String} + “def”.{String} ;

int i = 3.{int} + 5.{int} ;

Polymorphism

Subtype Polymorphism – essential for OO programming!

Def: A function exhibits *subtype polymorphism* if one or more of its formal parameters has subtypes.

Polymorphism

Example: Java

void g (double a) { ... }

int $\subset$ double
float $\subset$ double
short $\subset$ double
byte $\subset$ double
char $\subset$ double

} all legal types that can be passed to function 'g'.

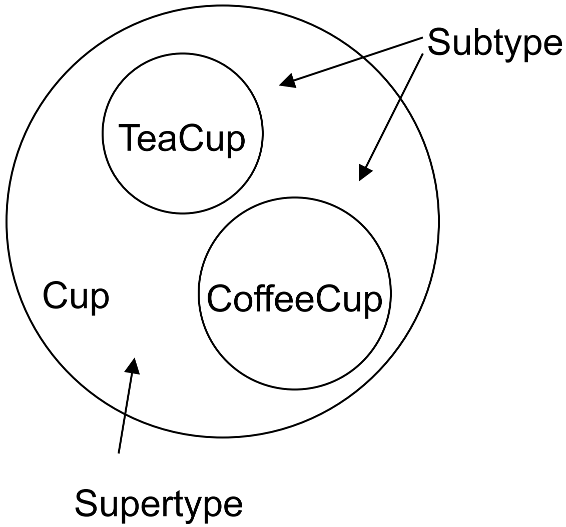
int i = 10;
g(i);

} Legal because of subtype polymorphism

Polymorphism

Example: Java

```
class Cup { ... };
class CoffeeCup extends Cup { ... };
class TeaCup extends Cup { ... };
```




```
void fill (Cup c) {...}

TeaCup t = new TeaCup();
CoffeeCup k = new CoffeeCup();

fill(t);
fill(k);
```

} subtype polymorphism

```
TeaCup t = new TeaCup();
Cup c = t;
```

← widening type conversion: TeaCup → Cup

safe!