

CS 329 Principles of Programming Languages

Slides-- Part I

CS 329 Lecture Series

Rushikesh K. Joshi



Types, Type Constructors, Subtyping



Types and Values

Types can be considered as sets

The members of the set that represents a type represent all possible values of the type



Value Assignment

```
T var;  
var = v1;  
var = v2;
```

A variable of type T can be assigned a value that is a member of the set defining type T

Assignment statement can be used to change the assignment of a value to a variable of given type; only that the value should be from the set defining the type.

An example

`bool = {true, false}`

`b1 = true;`

`b2 = false;`

`...`

`b1=b2`

The type is *bool*

`b1, b2` are the only possible values of this
type

Cardinality of a type

The count of all possible discrete values

`bool = {true, false}`
`#bool = 2`

`Week = {Mon, Tue, Wed, Thu, Fri, Sat, Sun}`
`#Week = 7`

Primitive Types

Sets of discrete values

To specify a type, simply enumerate all
its values

e.g. $\text{int} = \{-\text{MAX}, \dots, 0, \dots, +\text{Max}\}$

e.g. bool, int, float, char, short int,
unsigned int, enumerated data types
etc.

Language definitions provide some standard primitive types from
which composite types such as structures, functions, lists can be
constructed

Composite Types

These are constructible from other types

```
e.g. struct xyz {  
    int i;  
    char c;  
}
```

A structure or a record is thus a composite formed by taking a cross product of multiple types

Composite Types: Product types

```
Record R1 {  
    T1 v1;  
    T2 v2;  
}
```

$R1 = T1 \times T2$

$\#R1 = \#T1 \times \#T2$

Example: if $T1=T2=bool$, $\#R1 = 4$.

$R1 = \{ (t,t), (t,f), (f,t), (f,f) \}$

The cardinality again represents the count of all possible values of the given type.

Composite Types: Function types

A function $T2 \ f(T1)$ is a mapping from set $T1$ to set $T2$. i.e. f computes a value of type $T2$ given a value of type $T1$ as input parameter.

$f: T1 \rightarrow T2$

If $T1$ is boolean, and $T2$ is also boolean, we have

$f = \{ \{ (t \rightarrow t), (f \rightarrow t) \}, \{ (t \rightarrow t), (f \rightarrow f) \}, \{ (t \rightarrow f), (f \rightarrow t) \}, \{ (t \rightarrow f), (f \rightarrow f) \} \}$

how many different function bodies can you write against a function signature $T2 \ f(T1)$?

ans: $\#f = (\#T2)^{(\#T1)}$

Cardinality and values of a function type

The elements of the set corresponding to a function type are all possible mappings for a given function signature.

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: Array types

int A[10]

It can be modeled as a function that maps integers from range 1..10 to int

so type of array A is $T1 \rightarrow \text{int}$, where $T1 = \{1..10\}$

default initializer is the default mapping.

Cardinality of an array type represents the number of possible valuations of the array

e.g. 1111111111 is one of the many possible valuations.

Any other mapping can be used as a value of A, if the mapping is a valid value of the type that defines A.

A function type represents an array more naturally than a product type since we have the associated operation of indexing. Record elements are accessed by their names, whereas array elements are accessed by their indices.

Type Errors

Consider Type
int A[10]

In 'C', if you access element A[10], it constitutes an error. Since the type is undefined on index=10, such an access is called type error, or type violation.

Depending on the design of the programming language, a type error may get detected at compile time, or at runtime, or go undetected by the language's runtime environment, and may eventually get trapped inside the operating system such as through a segmentation fault.

More Composite Types



Recursive Types: Lists

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 :

$L = \text{either NULL or } T \times L$

or in other words,

$L = \text{NULL} + (T \times L)$, where $+$ defines a
disjoint union

The set defining the list type contains all possible lists of type T , but we have a closed recursive expression for the list type L .

An example list value

L = abcdec

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

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;  
}
```

$U = \text{int} + \text{char}$

i.e.

$U = \textit{either int or char}$

A value of type U is either a value of type int or a value of type char.

The union type (either/or) was used in the definition of the list type.

example: A union type defined in C

The Subtype Relation

Subsumption



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, there is no problem. For example as in the below program::

```
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 relation between the two types. What's that relation?

Subtype Relation

$$S <: T$$

we say that type S is a subtype of type T

For primitive types, a subset can be considered as a subtype.

Exmples:

$R1 = \{1,2,3,4\}$

$R1 <: \text{Int}$

$R2 = \{a,b,c,D,E\}$

$R2 <: \text{Char}$

What can we do with subtypes?

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

Subsumption Rule

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

The rule states that:
if value t is of type S and S is given as a
subtype of type T ,
then value t is also 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 .

However, subtype relation is not symmetric.

For example, the below function will not work correctly for all values of type T when sent as input parameter to $f()$.

```
f(S val) { A[3]; return A[val]; }
```

 For which case does it not work?

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

$R1 = T1 \times T2$

$R2 = S1 \times S2$

$R2$ can be considered as a subtype of $R1$, when $S1 <: T1$ and $S2 <: T2$

$R1 = \text{String} \times \text{String}$

$R2 = \text{RollNo} \times \text{Name}$

Subtype Relation for Product types: The combined rule

$R1 = T1 \times T2$

$R2 = S1 \times S2 \times S3$

R2 can be considered as a subtype of R1, when $S1 <: T1$ and $S2 <: T2$

$R1 = \text{String} \times \text{String}$

$R2 = \text{RollNo} \times \text{Name} \times \text{Age}$

Subtype Relation for Product types: Record Permutation Rule

$R1 = T1 \times T2$

$R2 = T2 \times T1$

$R2$ can be considered as a subtype of $R1$, and vice versa by the record permutation rule.

$R1 = \text{Name} \times \text{RollNo}$

$R2 = \text{RollNo} \times \text{Name}$

$R1 <: R2$, and $R2 <: R1$

The rule is at conceptual level, and its implementation in a programming language may require manipulating with the memory layouts for correct implementation of the rule.

Properties of subtype relation

Reflexive ✓

Symmetric ✗

Anti-symmetric ✗

transitive ✓



Function Subtypes



Function Types

float f (int x) {..} has type
int-->float

int g (int x) { ...} has type
int --> int

When can we say that a function type is a
subtype of another function type?

The Subsumption rule revisited


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

By the above rule of subsumption associated with subtype relation $<:$, the values of a subtype can be used safely as values of the (super)type.

Applying the rule to functions, if $(\text{type of } g) <: (\text{type of } f)$, we can use g safely wherever f is expected. Consider the program given below:



An example

```
int f (int x) {...}
main () {
    int v;
    int 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?

Consider g to be one of the following and find out which of these will be safe replacements for f in the above program?:

int --> int
float --> int

int --> float
float --> float

An example ..

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

We can see that if `g` defines its input parameter to be `float` or `int`, there will be no problem in accepting an input parameter `v` which is defined as `int` in the program.

However, if `g` returns a `float` type, it will result in loss of information when the return value gets assigned to variable `x` which has type `int`.

An example ..

```
int f (int x) {...}
main () {
    int v;
    int x;

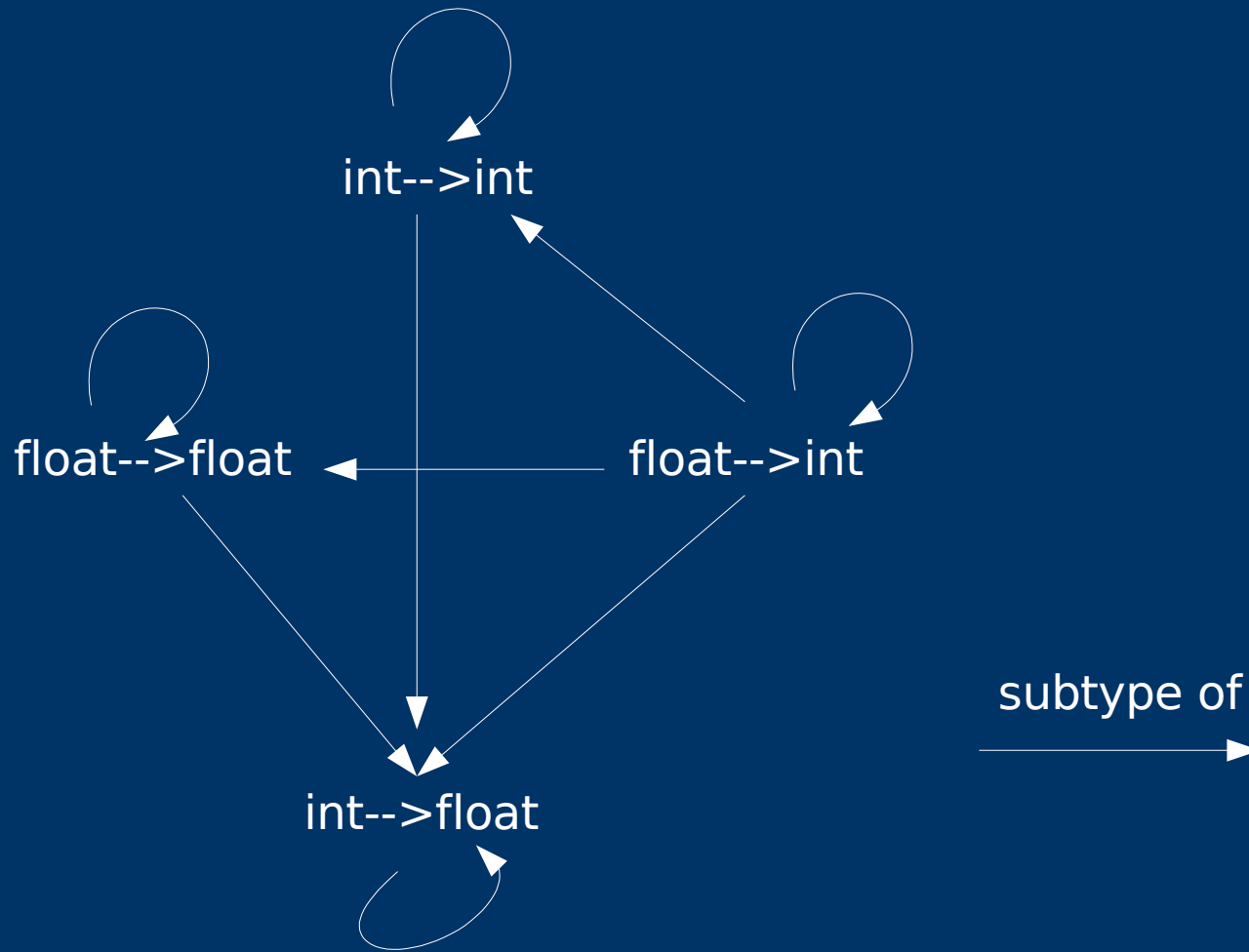
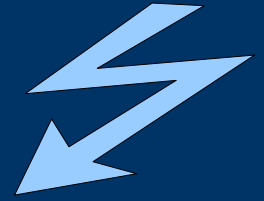
    ...
    x = f (v);
    ..
}
```

Both the below functions will be type-safe substitutions for `int f(int)`.

```
int g (int)
int g (float)
```

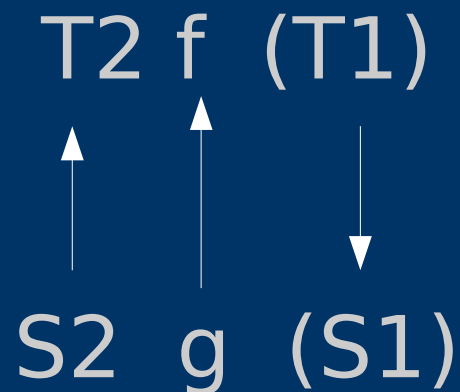
you can see that as long as there is no assignment of a value of a supertype to a variable of subtype, the usage is type safe. This safety condition can be observed in the case of types of input parameters and return results in the above two functions when they are used in place of `int f(int)`.

Type-subtype relations among four functions



The function subtyping rule

input parameters contravariant
output result covariant



Overloading, coercion



Overloading

10 + 2.3

10 + 2

2.3 + 10

2.3 + 2.4

Operator '+' is a function

$+: T1 \times T2 \rightarrow T3$

The implementation of operator + is internally provided by the language environment.

Considering the above four possible usages, what can we say about the Type of this internal function '+'. In other words, what should be the signature of this internal function '+'?

What's overloading?

The multiple apparent definitions of a function results in overloading of the function.

The name of the function is the same, but the same name can work with multiple signatures. That is to say that the function name is overloaded.

In the above case, `+` is overloaded with four possible signatures. However, the language may resolve overloading by using one of the plans discussed below.

Resolving overloading, Plan A

The 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 is called '**coercion**'

Note that this function is not really a super-type of all other functions.
Its
working relies on implicit coercion, and on correct use of types in the
program for coercion to work correctly.

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 lvalue type
has been chosen incorrectly as int.

Thus, overloading of 4 signatures can be completely eliminated with the
help of just one signature and the use of coercion wherever required.

Resolving overloading, Plan B

use two overloaded functions

float + float --> float

int + int --> int

and implicitly type-cast (coerce) integers to floats and back if needed, if one of the parameters is an int value

Thus, in this case, overloading of 4 signatures is resolved into overloading of 2 signatures with the help of coercion

Resolving overloading, Plan C

use four different functions

float + float --> float

int + int --> int

int + float --> float

float + int --> float

and select the one with exact matching signature.

Thus, in this case, there is no coercion, and full overloading is carried forward into implementation

Top type, Bottom Type



Top Type

The type of which every other type is a
subtype

example: Object type in Java

A value of any type can be used wherever
a value of the Top type is expected

Bottom Type

A type of which the values can also be used as values of all other types.

e.g. Type NULLT having a single value NULL.



Inheritance, Subtyping, Dynamic Binding in OOPs



Revisiting the function subtyping rule

$$g: S1 \rightarrow S2, T1 <: S1, S2 <: T2$$

$$g: T1 \rightarrow T2$$

or in other words,

$$g: S1 \rightarrow S2, (S1 \rightarrow S2) <: (T1 \rightarrow T2)$$

$$g: T1 \rightarrow T2$$

ok, what is the relation of g with 'f' then,
with signature of f as $T2 \rightarrow f(T1)$?

Note that f does not appear in the above formula. Why?

The answer is that f is just a value of type $T1 \rightarrow T2$,
and g a value of type $S1 \rightarrow S2$. By applying the above rule, we can say that
where a value f having type $T1 \rightarrow T2$ is expected, value g can be given,
as type of g is a subtype of $T1 \rightarrow T2$.

Subtyping induced by Subclassing

A obj;

obj = new A(); // a correct assignment

obj = new B(); // this will be correct if B is a subclass of A

We can use an instance of B where a type A is expected.
variable obj has type A, but the instance of B is being used.

Subclass defines a subtype.

Now we will address the problem of relating member functions in classes which are related through the subclass relationship.

Should the overriding function defined in subclass be a subtype of the corresponding function defined in the superclass, or should it be the other way?

Types in Inheritance

```
main () {  
  A obj;  
  J v;  
  K x;  
  
  read choice from the user;  
  if (choice==0) obj = new A();  
  else obj = new B();  
  
  x = obj --> f (v);  
  
}
```

Problem 1

what rules should be applied to ensure type safety of invocation `obj--> f (v)` in the main program?

```
class A {  
  public T2 f (T1 x) {....}  
}  
  
class B extends A {  
  public L2 f(L1 x) {....}  
}
```

Problem 2

What rules should be applied to permit `B::f()` the status as an overridden function that overrides `A::f()`?

***Towards Type Rules for (1) Member Function
Invocation, and for (2) Member Function Definition***

Problem 1 in the earlier slide relates to
type safety of a member function
invocation

Whereas Problem 2 relates to typing
restrictions on member function
definitions in order to establish
overriding

But What's the benefit of overriding?

```
main () {  
  A obj;  
  J v;  
  K x;  
  
  read choice from the user;  
  if (choice==0) obj = new A();  
    else obj = new B();  
  
  x = obj --> f (v);  
}
```

The benefit is
dynamic binding.

In the program on the left, an invocation to `obj->f()` gets bound to either `A::f()` or to `B::f()` depending on the class that is instantiated against variable `obj`. In this program, this user choice occurs at runtime, but that is fine for the invocation. The binding to the actual member function to be called also happens at runtime if overriding is used.

Dynamic Binding of member functions

A member function invocation statement is checked against the static type signatures, but the member function implementation that gets actually invoked is decided at runtime.

The function that is defined in the creation class of the object that is being used is picked up.

Solving Problem 1: Type checking of the invocation statement

```
main () {
A obj;
J v;
K x;

    read choice from the user;
    if (choice==0) obj = new A();
        else obj = new B();

    x = obj --> f (v);
}
```

```
class A {
    public T2 f (T1 x) {...}
}

class B extends A {
    public L2 f(L1 x) {...}
}
```

Problem 1

what rules should be applied to ensure type safety of invocation $\text{obj} \rightarrow f(v)$ in the main program?

We can see that f is being invoked through instance variable obj . Variable obj has static type A . Depending on the choice, obj may contain an instance of either A or B . However, the call to $\text{obj} \rightarrow f()$ can be type-checked wrt the static type of obj variable, which is A .

So we need to only ensure that $v:T1$ AND $x:T2$ by asserting $J<:T1$ AND $K<:T2$

And answer to question 2 (next slide) will ensure that this type-checking wrt the static signatures will be enough for the invocation statement to work correctly for all overloads of f in all possible subclasses of A .

Solving Problem 2: Ensuring type safety during dynamic binding, which is a property associated with overridden functions

```
main () {  
  A obj;  
  J v;  
  K x;  
  
  read choice from the user;  
  if (choice==0) obj = new A();  
    else obj = new B();  
  
  x = obj --> f (v);  
}
```

```
class A {  
  public T2 f (T1 x) {...}  
}  
  
class B extends A {  
  public L2 f(L1 x) {...}  
}
```

Problem 2

What rules should be applied to permit B::f() the status as an overridden function that overrides A::f()?

As seen from the program on the left, we are looking forward to correct working of overridden functions where a signature from the superclass is expected. This is achieved if we simply apply the function subtyping rule making $f::B <: f::A$, i.e.

$T1 <: L1$ AND $L2 <: T2$

Example of correct overriding

```
main () {  
  A obj;  
  int v;  
  int x;  
  read choice from the user;  
  if (choice==0) obj = new A();  
    else obj = new B();  
  
  x = obj --> f (v);  
}
```

v: int, x: int
=> Acceptable for invocation obj->f()

```
class A {  
  public int f (int x) {...}  
}  
  
class B extends A {  
  public int f(float x) {...}  
}
```

int A::f (int) <: int B::f(float)
=>Acceptable for B::f() to
be overriding A::f()

The above program is type-safe

Another Example of correct overriding

```
main () {  
  A obj;  
  nonnegativeint v;  
  float x;  
  read choice from the user;  
  if (choice==0) obj = new A();  
    else obj = new B();  
  
  x = obj --> f (v);  
}
```

```
class A {  
  public int f (int x) {...}  
}  
  
class B extends A {  
  public int f(float x) {...}  
}
```

v: nonnegativeint, x: float
nonnegativeint <: int
float <: int
=> Acceptable for invocation obj->f()

We have nonnegativeint <: int <: float
so v will work correctly as parameter to B::f
Also, value returned from B::f will get assigned correctly (i.e. safely) to x, a value of type float.

The above program is type-safe

So here are the rules

```
main () {
  A obj;
  J v;
  K x;

  read choice from the user;
  if (choice==0) obj = new A();
    else  obj = new B();

  x = obj --> f (v);
}
```

```
class A {
  public T2 f (T1 x) {...}
}

class B extends A {
  public L2 f (L1 x) {...}
}
```

The rule for type safe invocation

We make sure that $J <: T1$ AND $T2 <: K$

The rule for type safe overriding

Here we make sure that $T1 <: L1$ AND $L2 <: T2$

How do these two rules together make sure that all Js and Ks following the rule for type safe invocation will work correctly with all possible L1s and L2s following the rule for type safe overriding?

Fortunately Subtyping is Transitive. So we get $J <: T1 <: L1$, and $L2 <: T2 <: K$. This makes it possible for $v:J$ to work safely as parameter into $B::f()$, and value returned by $B::f()$ gets assigned safely to variable $x:K$.

What if the rule of type safe invocation is not followed?

```
main () {
A obj;
int v;
char x;
    read choice from the user;
    if (choice==0) obj = new A();
        else obj = new B();

    x = obj --> f (v);
}
```



Check the rule for type safe invocation

The rule fails!

float, the return type of A::f is not a subtype of char

```
class A {
    public float f (int x) {...}
}

class B extends A {
    public int f (float x) {...}
}
```




Check the rule for type safe overriding

Here it's fine!

- The compiler which guarantees static type checking can refuse to compile such a program, as it cannot guarantee type safety at compile time for all possible object value assignments to variable obj.


What if the rule of overriding is not followed? Carefully observe all the types

```
main () {  
  A obj;  
  int v;  
  float x;  
  read choice from the user;  
  if (choice==0) obj = new A();  
    else obj = new B();  
  
  x = obj --> f (v);  
}
```



Check the rule for type safe invocation
The rule is followed!

```
class A {  
  public float f (int x) {...}  
}  
  
class B extends A {  
  public char f (float x) {...}  
}
```




Check the rule for type safe overriding
Here it's not!

In this case, B::f can be permitted to exist as an independent function that has no subtyping relation with A::f

But since they both happen to use the same name 'f', they form a set of overloaded functions.


What if the rule of type safe invocation is not followed, but there exists an overloaded function somewhere down the chain?

```
main () {  
A obj;  
int v;  
char x;  
    read choice from the user;  
    if (choice==0) obj = new A();  
        else obj = new B();  
  
    x = obj --> f (v);  
}
```



Check the rule for type safe invocation
The rule is not followed!

```
class A {  
    public float f (int x) {...}  
}  
  
class B extends A {  
    public char f (float x) {...}  
}
```



Check the rule for type safe overriding
Here also the rule is not followed!
The two functions are considered overloaded

In this case, though there is an overloading available in the subclass B, the type safety of `x=obj-->f()` cannot be guaranteed at compile time since the instance can be created either from A or from B. So a compile time type error can be generated.

What if the rule of type safe invocation is not followed, but there exists an overloaded function in the static type of the variable through which the invocation is being made?

```
main () {  
  A obj;  
  int v;  
  char x;  
  read choice from the user;  
  if (choice==0) obj = new A();  
    else obj = new B();  
  
  x = obj --> f (v);  
}
```



Check the rule for type safe invocation
The rule is not followed!

```
class A {  
  public float f (int x) {...}  
  public char f (float x) {...}  
}  
  
class B extends A {  
  public float f (int x) {...}
```



Check the rule for type safe overriding
Here also the rule is followed for one pairing,
and there is also one overloaded definition in A

Solve it.

Do Java, C++ implement really these rules? Find out by writing programs.

Dynamic Binding in presence of multiple overridings within a single inheritance chain

The search for the implementation starts from the creation class of the object, and it continues up the inheritance chain. The first function that is found to be the subtype of the static type signature expected is picked up for dispatch. This binding happens during runtime.

what additional problem can occur with multiple inheritance?

