

Subtyping in Object orientation

CS 329 Lecture 6

Aug 8, 2007

Rushikesh K. Joshi



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 overloadings 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?

