A Summary of Type Checking Rules and Method Lookup in Java

Prakash Panangaden
School of Computer Science
McGill University

March 31, 2019

The following is a brief summary of the

Rules for Method Lookup and Type Checking.

In particular we discuss the Guassian integers example from class.

First the rules. Remember that there are two phases, compile time, which is when type checking is done and run time, which is when method lookup happens. Compile time is before run time.

- The type checker has to say that a method call is OK at compile time.
- All type checking is done based on what the declared type of a reference to an object is.
- Subtyping is an integral part of type checking. This means if B is a subtype of A and there is a context that gets a B where A was expected there will not be a type error.
- Method lookup is based on actual type of the object and not the declared type of the reference.
- When there is overloading (as opposed to overriding) this is resolved by type-checking.

1 Analysis of the `gaussInt` Example

```
Name        Declared Type   ActualType


a:      myInt       myInt
z       gaussInt    gaussInt
w       gaussInt    TBD
b       myInt       gaussInt
d       myInt       myInt
c       myInt       TBD


myInt a = new myInt(3);
gaussInt z = new gaussInt(3,4);
```

```
gaussInt w;
myInt b = z;

System.out.println("the value of z is"+ z.show());

> real part is 3 imag part is 4
```

this prints out the above line because z is declared to be of type `gaussInt`. It passes the type checker as there is a `show` method defined in the `gaussInt` class. At run time it uses the `show` method of `gaussInt` to display the above line.

```
System.out.println("the value of b is :" + b.show());

> real part is 3 and imag part is 4.
```

b is declared to be of type `myInt`. There is a method called `show` in the `myInt` class. The type checker sees that and because of that it passes the type checker, **but** the actual type of b is `gaussInt`. Method lookup is based on actual types of objects and therefor b uses the `show` method in the `gaussInt` class and displays what a `gaussInt` object would have shown.

```
myInt d = b.add(b)

System.out.println("the value of d is:"+ d.show());

> 6
```

b is declared to be of the type `myInt`, the type checker checks to see whether there is an `add` method in the `myInt` class. **Yes** there is one; it takes a `myInt` object and returns a `myInt` object as the result. At run time b's actual type is `gaussInt` the run-time system checks to see if there is an `add` method in the `gaussInt` class which matches the type that it was told by the type-checker. There are two add methods - one that takes a `myInt` and returns a `myInt` (This method has been inherited from the `myInt` class). The other takes a `gaussInt` and returns a `gaussInt`; this is the method that is explicitly defined in the `gaussInt` class. However the latter method does not match what the type-checker told the run-time system to expect.

NOW WHICH ADD METHOD DO WE USE?

since "When there is overloading, it is resolved by typechecking" the method which takes an object of the type `myInt` will be used. This is the method that has been inherited. It takes in a `myInt` and returns a `myInt`. Hence `b.add(b)` returns a `myInt` object and therefor NOW the actual type of d is `myInt`.

```
//w= z.add(b) -----(i)
//w = b.add(z)-----(ii)
```

These two will not type check

1. z is declared to be of the type `gaussInt`. There are two methods in the `gaussInt` class, the one that takes in a `myInt` object and returns a `myInt` object is used. Why? Once again

overloading is resolved by typechecking. Since b is declared to be a `myInt` object it will pick the `add` method that it inherited.

z is a `gaussInt` which is a subtype of `myInt` and hence is added to b and returns a `myInt`. w is declared to be a `guassInt`. Since `myInt` is not a subtype of `gaussInt` the assignment statement will not accept this for the right hand side, and hence would cause an error.

2. b is declared to be of the type `myInt`. The type checker checks if there is an `add` method in the `myInt` class there is one which expects a `myInt` object and returns a `myInt`object z is a `gaussInt` and since `gaussInt` is a subtype of `myInt`, b is added to z to produce a `myInt` object

   w is declared to be a `gaussInt` Since `myInt` is not a subtype of `gaussInt` it will not accept it and hence would cause an error.

```
w = ( (gaussInt) b).add(z)
```

This does type check as it is just a little modification to case 2 above. Now since w is a `gaussInt`, it better get a `gaussInt` on the right hand side. However, now, because of the cast, the typechecker knows that b is really a `gaussInt`. Thus, it now has to choose between two possible `add` methods. To resolve the overloading it uses the declared types; z has declared type `gaussInt`. Thus when it resolves the overloading of the `add` method it figures out to use the `gaussInt` to `gaussInt` version.