

Genericity Overview

Formal Foundations

F-bound Genericity:

$$S \leq T\langle S \rangle$$

supports binary methods, restricted to one inheritance level

Java, C#

Java: `Integer ≤ Comparable<Integer>`

(subtype relation = relation on bounds)

Higher-order Subtyping:

$$S <\# T \quad \text{if} \quad \forall U : S\langle U \rangle \leq T\langle U \rangle$$

supports binary methods directly, no substitutability

C++, Haskell

statically type-safe as bound in bounded genericity

(subtype relation \neq relation on bounds)

Compilation

genericity is always a compile-time mechanism to support parameterization

homogeneous compilation by **type erasure**: Java

one generic unit translated to one executable unit,
applicable to legacy code, but rather restrictive,
small binaries, not for elementary types, no optimizations

heterogeneous compilation: C++, Haskell

one generic unit translated to several executable units,
deep language integration, rather flexible, efficient, large binaries

combination of both: Scala, C#

Specification of Usability

basic specifications on use site always necessary (if info cannot be inferred),
specifications on declaration site optional,
variance (e.g., $A<? \text{ extends } X>$ and $B<? \text{ super } X>$) optional in OO

minimal decl.-site spec., no variance: C++ so far

no writing effort for declaration,
no variance, difficult to use, bad error handling

decl.-site spec. + decl.-site variance spec.: C#, Scala

user-friendly, good error handling,
large writing effort for server, inflexible variance

decl.-site spec. + use-site variance spec.: Java

good error handling, rather flexible form of variance,
variance difficult to use

Use-Site Variance (Java)

```
class List<A> {
    A get() {...}           // covariant position of A
    void add(A x) {...}    // contravariant position of A
}

void copy(List<? extends Person> from, // covariant use only
          List<? super Person> to)    // contravariant use only
{
    to.add(from.get()); // correct: contravariant use for to
                        //           covariant use for from
    from.add(to.get()); // wrong:  contravariant use for from
                        //           covariant use for to
}

copy(new List<Student>(), new List<Object>());
```

Declaration-Site Variance (C#)

```
interface ReadList<out A> { A get(); } // covariant
interface WriteList<in A> { void add(A x); } // contravariant
class List<A> : ReadList<A>, WriteList<A> {...}

void copy(ReadList<Person> from, WriteList<Person> to) {
    to.add(from.get()); // correct: contravariant use for to
                        //                covariant use for from
    from.add(to.get()); // wrong: contravariant use for from
                       //                covariant use for to
}
```

Templates in C++

Some Properties

constants usable as generic parameters:

```
template<typename T, int n> class buffer { ... };
```

(partial) specialization with “pattern matching”:

```
template<int n> class buffer<bool, n> { ... };
```

```
template<> class buffer<bool, 0> { ... };
```

implicit instantiation if no appropriate specialization available

code for methods produced only if used

Expressiveness

can be used to evaluate expressions statically (Turing-complete):

```
template<int x, int n> struct power {
    static const int r = x * power<x, n-1>::r;
};
template<int x> struct power<x, 0> {
    static const int r = 1;
};
```

algebraic data structures expressible:

```
template<class Head, class Tail> struct Cons { };
struct Nil { };
typedef Cons<int, Cons<float, Nil> > list_of_types;
```

Policy-Based Programming

```
#include <iostream>
template<typename lang> class HelloWorld : public lang {
    public: void Run() { cout << Message() << endl; }
};          // Message inherited from generic parameter

#include <string>
class HelloWorld_Msg_German {
    protected: string Message() { return "Hallo Welt!"; }
};

typedef HelloWorld<HelloWorld_Msg_German> MyHelloWorld;
MyHelloWorld hello;
hello.Run();
```

Alternative: Strategy Design Pattern (Java)

```
interface IMessage { String message(); }
class DMessage implements IMessage {
    public String message() {return "Hallo Welt!";}
}

class HelloWorld {
    private IMessage msg;
    public HelloWorld(IMessage msg) {this.msg = msg;}
    public void run() {System.out.println(msg.message());}
}

class Main {
    public static void main(String[] args) {
        HelloWorld hello = new HelloWorld(new DMessage());
        hello.run();
    }
}
```

Annotations and Reflection

Annotations

annotation is optional parameter that

- can be associated with class, method, parameter, ...

- is set statically in (Java) code – for parameterization,

- can be read by whole tool chain until run-time (reflection).

```
@Override
```

```
public String toString() { ... }
```

```
@BugFix(who="Kaspar", date="2015-04-17", level=3,  
        bug="class unnecessary and maybe harmful",  
        fix="contents of class body removed")
```

```
public class Buggy { }
```

Definition of Annotation

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface BugFix {
    String who() default "me"; // author of bug fix
    String date();           // when was bug fixed
    int level() default 1;   // importance level 1-5
    String bug();           // description of bug
    String fix();           // description of fix
}
```

Annotation at Run-time

@Retention(RUNTIME) → real interface constructed:

```
public interface BugFix
    extends java.lang.annotation.Annotation {
    String who();
    String date();
    int level();
    String bug();
    String fix();
}
```

Access through Reflection

```
String s = "";
BugFix a = Buggy.class.getAnnotation(BugFix.class);
if (a != null) { // null if no such Annotation
    s += a.who() + " fixed a level " + a.level() + " bug";
}
```

```
Annotation[] as = Buggy.class.getAnnotations(); // get all
Method[] ms = Buggy.class.getMethods();
Annotation[] bs = ms[i].getAnnotations();
```


Functional versus Object-oriented Programming

Functional Object-oriented Programming

possible point of view: object-orientation = organisational paradigm, model of computation orthogonal (imperative, functional, etc.)

contradiction: object has state, behaviour, identity;
referential transparency = no state, no identity

in practice: no referential transparency for input/output

→ state and identity also in functional programming,
but most inner program parts are referentially transparent

state of the art: functional part separated from the object-oriented part

pattern matching also usable in object-oriented programming,
genericity related to referential transparency

Delegates in C#

```
using System;
delegate void Sample(string message);
class MainClass {
    static void Method(string message)
        { Console.WriteLine(message); }
    static void Main() {
        // Instantiate delegate with named method:
        Sample d1 = Method;
        // Instantiate delegate with anonymous method:
        Sample d2 = delegate(string message)
            { Console.WriteLine(message); };
        d1("Hello");
        d2(" World");
    }
}
```

Lambda Expressions in Java

```
@FunctionalInterface
```

```
interface X { void anyname(String s); }
```

```
@FunctionalInterface
```

```
interface Y { int op(int a, int b); }
```

```
X x1 = (String p) -> System.out.println(p);
```

```
X x2 = p -> { System.out.println(p); System.out.println(p); };
```

```
Y y1 = (l, r) -> l + r;
```

```
Y y2 = (l, r) -> { return l * r; };
```

```
x1.anyname("Hello world!");
```

```
x2.anyname(y1.op(1,2) + " " + y2.op(3,4));
```

Lambda Expression

is a **closure** encapsulating a method in an object;
local context is kept (in contrast to function pointer);
anonymous, only accessible through object

simple syntax = syntactic sugar + intelligence

environment implicitly created (as inner class)

type inference for parameters (because of no subtyping)

important in applicative programming styles