Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Exercise 7

## Linearization, Bytecode Verification, and Parametric Polymorphism

November 18, 2022

## Tasks covered in class

Tasks 1-8 will be covered in the exercise session. The remaining tasks are material for self-study.

### Task 1

Consider the following declarations in Scala:

```
class C
trait T extends C
trait U extends C
class D extends C
```

Find all the types that can be created with or without traits, as well as the subtype relations between them.

solution

We can create the following types:

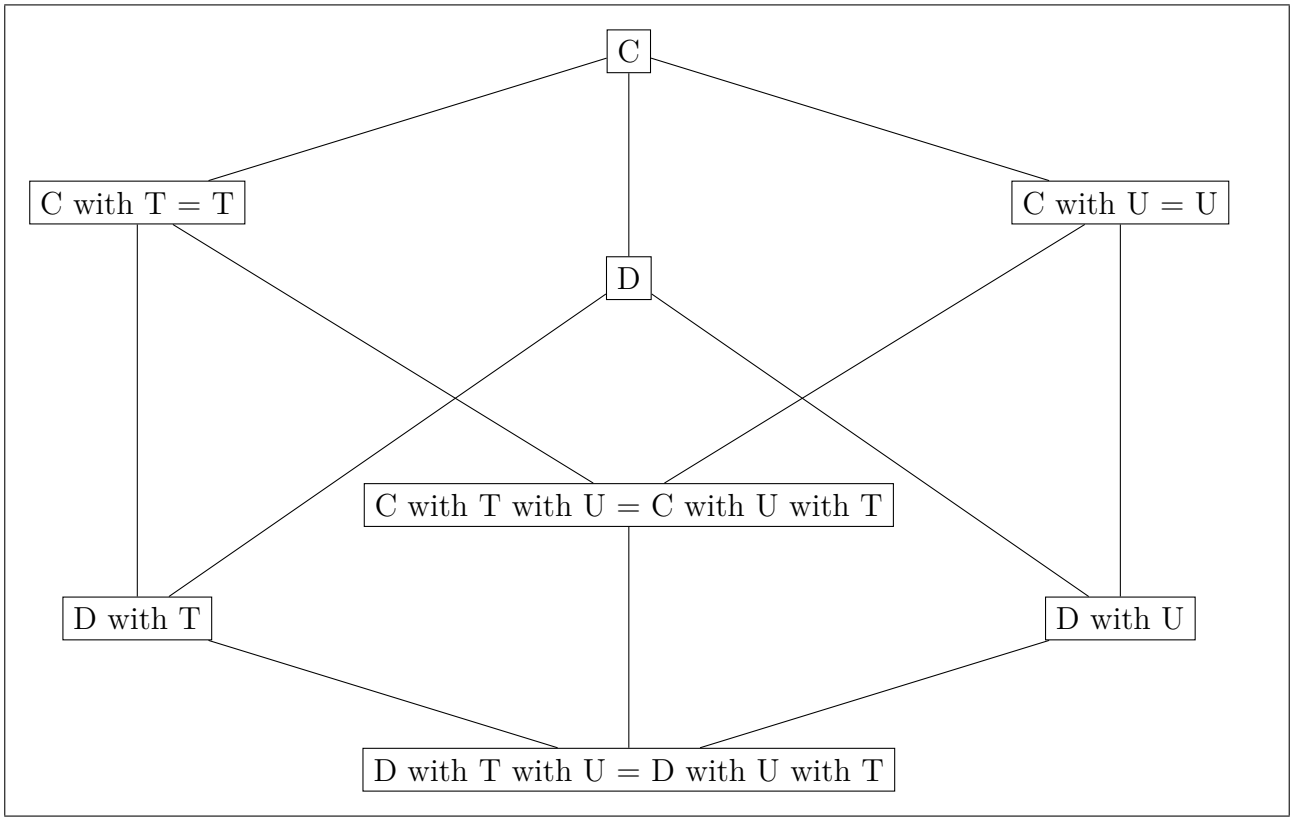
`C`, `D`, `T`, `U`,
`C with T` (same type as `T`, because `T extends C`),
`C with U` (same type as `U`, because `U extends C`),
`C with T with U` (same type as `C with U with T`),
`D with T`,
`D with U`,
`D with T with U` (same type as `D with U with T`).

The subtype relation is reflexive and transitive. Moreover, let $X'$, $Y'$ be the two base classes from which we derive $X$ and $Y$ by mixing in traits. Let $A$ be the set of all traits mixed into the first class and $B$ the set of all traits mixed into the second class. The rule is as follows:

$$X <: Y \text{ if and only if } X' <: Y' \text{ and } A \supseteq B.$$

Note: The above rule applies in our example, but it is not a general rule for subtyping in the presence of traits. Note that even if `D with T with U` and `D with U with T` are equivalent types (subtypes of each other), they can describe different behavior, causing subtle problems for behavioral subtyping.

We can also visualize the types and the subtype relations between them (the edges corresponding to reflexive and transitive subtype relations were omitted):




## Task 2

Consider the following Scala code:

```
class Cell {
   private var x:Int = 0
   def get() = { x }
   def set(i:Int) = { x=i }
}

trait Doubling extends Cell {
   override def set(i:Int) = { super.set(2*i) }
}

trait Incrementing extends Cell {
   override def set(i:Int) = { super.set(i+1) }
}
```

**A)** What is the difference between the following objects?

```
val a = new Cell
val b = new Cell with Incrementing
val c = new Cell with Incrementing with Doubling
val d = new Cell with Doubling with Incrementing
```

solution

Object `a` behaves like a normal cell. Object `b` is also a cell, but it increases the stored value by 1. The interesting difference is between `c` and `d`. They are both cells. They have mixed in exactly the same traits. However, calling `set(i)` has a different effect on them: it stores `2i+1` in the first one and `2(i+1)` in the second one.

**B)** We try to use the following code to implement a cell that stores the argument of the set method multiplied by four:

```
val e = new Cell with Doubling with Doubling
```

Why does it not work? What does it do? How can we make it work?

solution

Trait `Doubling` will not get mixed in twice. Scala rejects this statically.

A possible attempt to bypass the problem is to create a new trait `Doubling2` that behaves exactly like `Doubling`:

```
trait Doubling2 extends Doubling
val e = new Cell with Doubling with Doubling2
```

The code passes through, but dynamically `e` behaves as if it were a `Cell with Doubling`. Scala lets the code go through, because `Doubling2` may introduce new functionalities, but does not include `Doubling` twice in the linearization.

Our last try (the one that works), is to create a whole new trait from scratch, without reusing existing code:

```
trait Doubling3 extends Cell {
   override def set(i:Int) = { super.set(2*i) }
}
val e = new Cell with Doubling with Doubling3
```

And now `e.set` quadruples its argument, as expected.

**C)** Find a modularity problem in the above, or a similar, situation. Hint: a client that is given a class `C` does not necessarily know if a trait `T` has been mixed in that class.

solution

A problem is that a method that accepts `Cell with Doubling with Incrementing` as an argument could also be passed a class of the type `Cell with Incrementing with Doubling` - so what it can actually assume about its inputs is less than would be expected.

**D)** We propose the following solution to support traits together with behavioral subtyping: assume `C` is a class with specification `S`. Each time we create a new trait `T` that extends `C`, we must ensure that `C with T` also satisfies `S`. Show a counterexample that demonstrates that this approach does not work.

solution

Consider the following example:

```
class Cell {
   var x:Int = 0
   // ensures x <= i + 1
   def set(i:Int) = { x=i }
}

trait Incrementing extends Cell {
   override def set(i:Int) = { super.set(i+1) }
}

trait Incrementing2 extends Cell {
   override def set(i:Int) = { super.set(i+1) }
}
```

Both `Cell with Incrementing` and `Cell with Incrementing2` are behavioral subtypes of `Cell`. But `Cell with Incrementing with Incrementing2` is not a behavioral subtype of `Cell`, as the following example shows:

```
val c: Cell = new Cell with Incrementing with Incrementing2
c.set(4)
assert(c.x <= 5) // fails, postcondition of Cell.set not fulfilled
```

## Task 3

Consider a Java class `E`, which has a method `f` with the following signature: `void f();`

The method `f` has one local variable `v` and the following body:

```
0: iconst 5
1: istore 1
2: aload 0
3: astore 1
4: iload 1
5: iconst 1
6: iadd
7: istore 1
8: return
```

The maximal stack size is equal to 1. Can the provided bytecode be verified? If so then verify it, otherwise explain which line of code causes the problem and why.

solution

In the following, we try to verify the bytecode. `T` is an uninitialized register. A state is represented by a pair `(S,R)` where `S` describes the content of the stack and `R` describes the content of the registers.

```
// ([],[E,T]) -- initial state
iconst 5
// ([int],[E,T])
istore 1
// ([], [E,int])
aload 0
// ([E], [E,int])
astore 1
// ([], [E,E])
iload 1
// ERROR!
...
```

The error happens because `iload 1` expects that the local variable has the type integer, but its type is `E`.

## Task 4

The Java bytecode verifier is more permissive than the Java type system. Provide a program that demonstrates this.

solution

Here is an example of such a program:

```
x = true;
x = 5;
```

The type of the variable can change in the bytecode but not in the source code.

## Task 5

*(from a previous exam)*

Assume two Java classes `A` and `B`, where `B` is a subclass of `A`. Consider the following bytecode:

```
0: aload 1
1: astore 2
2: goto 0
```

and assume that the input to the initial node of this code is `([],[A,A,B])`, where the first list indicates the content of the stack and the second list indicates the content of the registers.

After running the bytecode type inference algorithm, what is the inferred input to the initial node?

(a) **CORRECT:** `([],[A,A,A])`

(b) `([],[A,A,B])`

(c) `([],[A,B,B])`

(d) Nothing is inferred – the type inference does not terminate

(e) Nothing is inferred – the type inference rejects the program

solution

Running the bytecode type inference algorithm once from instruction 0 to instruction 2 results in retrieving the object in the second register and storing it in the third register. This object is of type `A`, so the result propagated to instruction 0 after the jump is `([],[A,A,A])`. We now need to join this state with the initial state `([],[A,A,B])`, by computing the pointwise smallest common supertype (`scs`). Since `B` is a subclass of `A`, `(scs(A, B)) = A`. Therefore the resulting input to the next iteration of the algorithm is `([],[A,A,A])`. This is then propagated to the jump instruction, reaching the fixed point. (The inference algorithm runs twice through instructions 0 and 1, and once through instruction 2, before reaching the fixed point.)

## Task 6

Consider the following Java code:

```
interface IFace { void m(); }

class Cl1 implements IFace {
   public void m() { System.out.println("Cl1.m"); }
}
class Cl2 implements IFace {
   public void m() { System.out.println("Cl2.m"); }
}

public class Test {
   public static void main(String[] args) {
```

```
        foo(true);
        foo(false);
    }
    public static void foo(boolean param) {
        IFace iface = null;
        if (param) { iface = new Cl1(); }
        else { iface = new Cl2(); }
        iface.m();
    }
}
```

**A)** What type will be calculated for the variable `iface` of the method `foo` during bytecode verification?

solution

The inference algorithm does not take interfaces into consideration, so the calculated type for the variable `iface` is `Object`.

**B)** When can we decide that `iface.m()` is safe to call, during bytecode verification or during execution?

solution

As the inferred type of the `iface` is `Object`, the decision can be made only during execution.

**C)** Would your answer from **B** be the same if `IFace` were a class instead of an interface? What if `IFace` were an abstract class?

solution

In both cases the inferred type of the `iface` would be `IFace`. The decision about the safety of the call could be made during bytecode verification.

## Task 7

*(from a previous midterm)*

Consider the following Java program, which is rejected by the Java compiler:

```
class Logger<T> {
    public void log(T t) {
        System.out.println(t.loggerString());
    }
}
```

**A)** If the code above were allowed to compile without errors, what could go wrong? To answer, write a brief code sample that uses `Logger` in a way which causes a failure at runtime.

solution

```
Logger<Object> l = new Logger<Object>();
l.log(new Object());
```

**B)** How can we fix the class `Logger` so that it compiles, while preserving its functionality? You should not modify the method `log`, but otherwise can change or add any code. Your solution should include all details required to check that `Logger` is a valid Java class.

solution

```
interface Loggable {
    String loggerString();
}

class Logger<T extends Loggable> { ... }
```

**C)** Assume that class `Logger` has been fixed to resolve the problem from point **A**. Let `A` and `B` be two classes such that `A` is a supertype of `B` and `Logger<A>` and `Logger<B>` are valid instantiations. Consider the following method:

```
void foo(Logger<A> logA) {
  Logger<B> logB = logA;
  logB.log(new B());
}
```

The Java compiler rejects this code. Is the code safe? That is, if it were allowed to compile, would it run without failure?

solution

Yes, the code is safe.

**D)** Suppose we relax the Java type system rules to allow contravariant generics.

- Will the method `foo` compile now?

solution

Yes.

- What are two situations that will require dynamic checks in order to enable contravariant generics in a language, without limiting what a developer can write in a generic class?

solution

  - When calling methods of generic classes, it would be necessary to check whether the dynamic type of the result is a subtype of the static type of the variable where the result is stored.
  - When reading fields of generic classes, it would be necessary to check whether the dynamic type of the field is a subtype of the static type of the variable where the object is stored.

## Task 8

*(from a previous exam)*

**A)** Recall the Java interface `Comparable<T>` that was shown in the lecture:

```
public interface Comparable<T> {
    public int compareTo(T other);
}
```

The method `compareTo` returns $\begin{cases} 1 & \text{if } \texttt{this} \text{ is greater than } \texttt{other} \\ 0 & \text{if } \texttt{this} \text{ is equal to } \texttt{other} \\ -1 & \text{if } \texttt{this} \text{ is less than } \texttt{other} \end{cases}$

Suppose we want to turn `Comparable` into an abstract class with an additional helper method `greaterThan`, that returns true if and only if `this` is greater than `other`.

Assume the following implementation:

```
public abstract class Comparable<T> {
    public abstract int compareTo(T other);

    public boolean greaterThan(T other) {
        return other.compareTo(this) < 0;
    }
}
```

**A.1)** Why does this implementation not type check?

solution

`T` requires an upper bound that provides `compareTo`.

**A.2)** Fix the type error by changing only the body of `greaterThan`, while preserving the intended semantics of the method.

solution

```
return this.compareTo(other) > 0;
```

**A.3)** Fix the type error by changing only the class signature and the signature of the method `compareTo`.

solution

Let the class signature be `Comparable<T extends Comparable<T>>` and the signature of `compareTo` be `compareTo(Comparable<T> other)`.

**B)** Suppose we have the following class:

```
class A<X                    ,Y                    > {
    X a;
    Y b;
}
```

Consider a variable `v` whose type is `A<S,T>` where `S` and `T` satisfy the type bounds that you have to insert above. Your type bounds have to guarantee that for all sequences of `a` and `b` accesses on `v` (e.g., `v.b.a.b.a.a.b.b`) the following two properties hold:

- The static type of a sequence ending in `a` is `S`.
- The static type of a sequence ending in `b` is `T`.

solution

```
class A<X extends A<X,Y>,Y extends A<X,Y>> {
    X a;
    Y b;
}
```

# Tasks not covered in class

## Task 9

*(from a previous exam)*

Consider the following Scala code:

```
class A           {          def bar() =                ""  }
trait B extends A { override def bar() = super.bar() + "B" }
trait C extends B { override def bar() = super.bar() + "C" }
trait D extends B { override def bar() = super.bar() + "D" }

object Main {
  def main() { foo(new A with D with C with B) }
  def foo(x: A with D) { println(x.bar()) }
}
```

What would be the output of the call `Main.main()`?

(a) BDB

(b) BBDBC

(c) BBCBD

(d) DB

(e) **CORRECT:** BDC

(f) BCD

(g) None of the above

solution

The super calls are resolved based on the linear order.
`L(new A with D with C with B) = L(B), L(C), L(D), L(A)` (*)
`L(A) = A`
`L(D) = D, B, A`
`L(C) = C, B, A`
`L(B) = B, A`
We now substitute the linearizations of `A, D, C, B` in (*) (in this order) and make sure the same class/trait is not included twice:
`L(new A with D with C with B) = L(B), L(C), L(D), A`
`L(new A with D with C with B) = L(B), L(C), D, B, A`
`L(new A with D with C with B) = L(B), C, D, B, A`
`L(new A with D with C with B) = C, D, B, A`

The call `x.bar()` corresponds to `C.bar()`, as `C` is the first in the linear order.
`C.super().bar()` is `D.bar()`, as `D` follows after `C` in the linear order.

## Task 10 (from a previous exam)

Consider the following Scala code, which compiles correctly and models some jobs a `Person` may have. To work as a `Lawyer` or as a `TaxiDriver`, one needs to have a valid license. This requirement can be expressed through *self type annotations* added to the traits `Lawyer` and `TaxiDriver` (as in the given code). These annotations are checked by the compiler and allow

the traits `Lawyer` and `TaxiDriver` to be mixed only into subtypes of `PersonWithLicense`. Self type annotations enable code reuse without subtyping, that is, `Lawyer` and `TaxiDriver` $\not<:$ `PersonWithLicense`, but the methods of the class `PersonWithLicense` are available and can be overridden inside these two traits.

```
class Person { def work(): String = { return "working"; }}

class Student { def work(): String = { return "studying"; }}

class PersonWithLicense extends Person {
 def hasValidLicense(): Boolean = { return false; }
}

trait Gardener extends Person {
 override def work(): String = { return super.work() + " in the garden";}
}

trait Lawyer extends Person {
 this: PersonWithLicense => // self type annotation

 override def work(): String = {
   if(this.hasValidLicense()) return super.work() + " in court";
   return "not " + super.work();
 }

 override def hasValidLicense(): Boolean = { return true; }
}

trait TaxiDriver extends Person {
 this: PersonWithLicense => // self type annotation

 override def work(): String = { return super.work() + " in Zurich"; }
}
```

**A)** For each of the following two code fragments (**A.1** and **A.2**), if they compile, write the output of their execution. Otherwise, briefly explain why they are rejected by the compiler.

### A.1

```
val lawyer: Lawyer = new PersonWithLicense with Lawyer with TaxiDriver;
println(lawyer.work());
```

solution

The code compiles: the traits `Lawyer` and `TaxiDriver` are mixed into a subtype of `PersonWithLicense`, as required by the self type annotation and `new PersonWithLicense with Lawyer with TaxiDriver` <: `Lawyer`.

The linearization of `new PersonWithLicense with Lawyer with TaxiDriver` is `TaxiDriver`, `Lawyer`, `PersonWithLicense`, `Person`. When executed, the code prints: working in court in Zurich

### A.2

```
val student: Gardener = new Student with Gardener;
println(student.work());
```

solution

The code does not compile, because `Student` is not a subclass of `Person` (the superclass of the trait `Gardener`)

**B)** Add **one** method to any of the given classes or traits **except** `PersonWithLicense` (explicitly write to which one) and fill in the instantiation from the client code below, such that it compiles and when executed prints `not working in Zurich in the garden`. You are **not allowed** to directly return this string, to use reflection, to define new classes or traits, nor to modify the given code. If this is not possible, briefly explain why.

```
// Client code:
val person = new _____________________________________________________
println(person.work());
```

The following method should be added to:

solution

```
// Client code:
val person = new PersonWithLicense with Lawyer with TaxiDriver with
   Gardener;
println(person.work());

trait TaxiDriver extends Person {
  ...
  // additional method:
  override def hasValidLicense(): Boolean = { return false; }
}
```

Note that if we try to add the method `hasValidLicense` to the trait `Gardener`, the client code does not compile, as `new PersonWithLicense with Lawyer with TaxiDriver with Gardener` inherits two methods with the same signature.

## Task 11

Assume we have two Java classes `A` and `B`. Consider the following Java class `C`:

```
class C {
    void foo(A x) {
        int y = 7;
        this.bar(y, x);
    }

    B bar(int u, A v) {
       ...
    }
}
```

Assume that the method `foo` gets compiled into bytecode as follows:

```
0: iconst 7
1: istore 2
2: aload 0
3: aload 2
4: aload 1
5: invokevirtual C.bar.B(int,A)
```

Can this bytecode be verified? If so, what is the final state (after line 5)?

solution

We assume that the maximal stack size is 3 and that $MR = 3$ (since we have three parameters/local variables): `this`, one argument (`x`), and one local variable (`y`). The initial state is `([], [C, A, T])`, where `C` is the type of `this`, `A` is the type of the argument `x` and the local variable `y` is uninitialized.

```
// ([],[C,A,T])
0: iconst 7
// ([int],[C,A,T])
1: istore 2
// ([],[C,A,int])
2: aload 0
// ([C],[C,A,int])
3: aload 2
// ERROR!
...
```

The error happens because `aload 2` expects that the local variable (from register 2) has a reference type, but its type is `int`.

Let's now assume that we correct the given bytecode, such that in line 3 we have `iload 2`. All the other instructions remain unchanged. We then obtain:

```
... // as before
3: iload 2
// ([int,C],[C,A,int])
4: aload 1
// ([A,int,C],[C,A,int])
5: invokevirtual C.bar.B(int,A)
// ([B],[C,A,int])
```

So the bytecode successfully verifies.

## Task 12

*(from a previous exam)*

Consider an *incorrect* bytecode verifier called *BuggyVerifier*, in which due to a bug the `aload` rule assumes that the loaded element is stored at the bottom of the stack instead of at the top (see the formal description below), while all the other rules are implemented correctly.

$$\begin{array}{l}\text{aload } n: \\ \quad (S, R) \to (\underline{S.R(n)}, R), \\ \quad \text{if } 0 \leq n < MR \ \wedge \ R(n) <: \mathit{Object} \ \wedge \ |S| < MS\end{array}$$

Assume that the initial state (stack and registers) is $([\,], [A, B])$, with the maximum number of registers $MR = 2$ and the maximum stack size $MS = 2$. $A$ and $B$ are classes such that $B <: A$.

**A)** Write a short bytecode program that is accepted by *BuggyVerifier*, but is **not** accepted by a correct bytecode verifier. Clearly mark the line at which the correct verifier detects an error, and briefly describe the error.

solution

```
0: iconst 42
1: aload 0
```

```
2: istore 0 // ERROR: the head of the stack is not an integer
```

You can use in your solution all the bytecode operations seen during the lectures. As a reminder, here are some of them:

- `iconst` $n$: create on the stack a value $n$ of type `int`.
- `iload` $n$: load on top of the stack an element of type `int` from the $n$-th register.
- `astore` $n$: remove an object from the top of the stack and store it in the $n$-th register.
- `goto` $n$: continue the execution from the operation at label $n$.

**B)** Is it possible that *BuggyVerifier* incorrectly accepts a program that overflows the stack, by pushing more than *MS* elements? Write yes or no, then motivate your answer.

solution

No. The rule still correctly checks that the stack size is less than MS.

## Task 13

**A)** Compare dynamic type checking with the `dynamic` keyword to static type inference with `var` in C#:

- Give a correct program which can be realized with `dynamic` but not with `var`.

solution

```
static void Main() {
    dynamic x;
    if(condition()) {
        x = 5;
    } else {
        x = "hello";
    }

    Print(x);
}

static void Print(string str) {
    Console.WriteLine(str);
}

static void Print(int value) {
    Console.WriteLine(value);
}
```

- Give an incorrect program which will be accepted by the compiler with `dynamic` but not with `var`.

solution

```
var x = 3;
x.substring(...);
```

**B)** C#'s most general type is `object`. Similar to `var` and `dynamic`, you can write `object x = ...` with an expression of any type on the right-hand side.

- Given a compiling program using `var`. Can we replace all `var` keywords by `object` and add explicit casts in the right places so that the program compiles and runs as before?

solution

This will be possible in all cases where we know what the type of the variable declared with `var` is. In those cases we can just cast the declared variable in all places where it is used to the most general type fulfilling all static type constraints on the corresponding variable. Since the original program compiled, such a type must exist.

In the case of anonymous types however, we do not know the name of the type to cast to. Consider:

```
var x = new { a = 108, b = "Hello" };
Console.WriteLine(x.b);
```

Here, we could change `var` to `object`, but we will not be able to cast `x` in the second line, because we do not know the type name which the compiler generates for this anonymous type.

- Given a compiling program using `dynamic`. Can we replace all `dynamic` keywords by `object` and add explicit casts in the right places so that the program compiles and runs as before?

solution

Generally we cannot do this, as shown in the following example:

```
static void Main() {
    dynamic x;
    if(condition()) {
        x = 5;
    } else {
        x = "hello";
    }

    Print(x);
}

static void Print(string str) {
    Console.WriteLine(str);
}

static void Print(int value) {
    Console.WriteLine(value);
}
```

To make this code work with `object`, we would need to add explicit type checks and cast the argument to the proper static type.

For both questions, either informally describe how to do the replacement, or give a counter-example where the transformation will always produce a program that does not compile or behaves differently. Note that explicit casts to `dynamic` are not allowed in the transformation.

**C)** Assume now a language like C#, but with covariant return types and contravariant parameter types. Given four classes `A`, `B`, `C` and `D`:

```
class A { int m (int x); }
```

```
class B { void m (dynamic x); }
class C { dynamic m (int x); }
class D { dynamic m (dynamic x); }
```

Develop a subtyping rule for the `dynamic` type annotation and informally explain the reasoning behind it. What are the potential subtypes among the four classes above?

solution

Following the Substitution principle, `dynamic` is equivalent to `object`, in that it accepts any type. Therefore, the usual subtyping rules apply, treating `dynamic` as the most general supertype of all other types. The potential subtyping relations are `A <: C` and `D <: C`.

There are two different ways of looking at class `B`. On the one hand, we could just say that `void` is a special keyword that indicates the absence of a return value, and thus the method `B.m` is unrelated to the other methods. Alternatively, we can allow methods with `void` return type to be overwritten by methods with any return type (assuming the parameter variance rules are satisfied): if a client code is written to expect `void` (no return value), then we could instead use a method which returns an arbitrary value and just discard it. In this second interpretation we will additionally have `D <: B`.

## Task 14

In this task, you have to implement (using three different approaches) a list in Java that supports the following two methods (where `i` represents an index):

```
public void add(int i, Object el)
public Object get(int i)
```

Discuss the advantages and the limitations of the three different approaches below.

**A)** Implement the list using only one class without generics.

solution

```
public class List {
   Object[] elements = new Object[100];
   public void add(int i, Object el) {elements[i] = el;}
   public Object get(int i) {return elements[i];}
}
```

Advantages: short implementation.

Limitations: the return type of the method `get` is `Object`; when using it, we usually have to dynamically cast its return values.

**B)** Implement the list using one abstract class/interface and then (some) subclasses that implement it for different types.

solution

```
public interface List {
   public void add(int i, Object el);
   public Object get(int i);
}

public class IntList implements List {
   Integer[] elements = new Integer[100];
```

```
    public void add(int i, Object el) {elements[i] = (Integer) el;}

    public Integer get(int i) {return elements[i];}
}
```

Advantages: the method `get` returns an `Integer`, thus we do not need dynamic casting of its return values.

Limitations: we have the same limitations as before (if programming against the interface), and in addition code duplication and further type casts/checks in the implementation of concrete list classes, e.g., in `add`. Moreover, we do not have behavioural subtyping, since the method `IntList.add` may not respect the expected contracts of `List` (due to the additional cast). For example, if we invoked `add` passing an object that is not an instance of `Integer`, the runtime environment would raise an exception and the element would not be added to our list.

**C)** Implement the list using generic types.

solution

```
public class List<T> {
    T[] elements = (T[]) new Object[100];
    public void add(int i, T el) {elements[i] = el;}
    public T get(int i) {return elements[i];}
}
```

Advantages: short implementation, statically type safe.

Limitations: none, we have only advantages :)