ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Exercise 12

Initialization

December 11, 2015

## Task 1

Consider a Java class `Vector`, representing a 2 dimensional vector:

```
public class Vector {
   public Number! x; // Remark: Number is a super-interface for
   public Number! y; //   Integer, Double, etc.

   public Vector (Number! x, Number! y) {
      this.x = x;
      this.y = y;
   }
}
```

Suppose that we add a subclass `Vector3D` which has a third `Number` field `z` and a new method `volume()`:

```
 public class Vector3D extends Vector {
   public Number! z;

   double volume() {
      return x.doubleValue()*y.doubleValue()*z.doubleValue();
   }
}
```

Which of the following method definitions compile (assuming that the data-flow analysis for non-null types doesn't consider the semantics of `instanceof`)? Which would always run safely (if compiled without typechecking)? Explain your answers.

### A)

```
double getVolume1(Vector? c) {
   if(c instanceof Vector3D) {
      return c.volume();
   } else { return 0.0; }
}
```

solution

`getVolume1` won't compile for two reasons - Java will complain that `c` is of (class) type `Vector` for which method `volume` is not defined, and a non-null type checker would complain that it cannot determine that `c` is non-null when the call is made. However, the program would run safely - the if-condition not only guarantees that the method is defined for the call, but implicitly that the expression `c` is non-null when the call is made (because Java defined that (`null instanceof T`) always evaluates to `false`.

**B)**

```
double getVolume2(Vector? c) {
   if(c instanceof Vector3D) {
      return ((!) c).volume();
   } else { return 0.0; }
}
```

solution

`getVolume2` won't compile for the first reason above - Java will complain. The code would still be safe.

**C)**

```
double getVolume3(Vector? c) {
   if(c instanceof Vector3D) {
      return ((Vector3D!) c).volume();
   } else { return 0.0; }
}
```

solution

`getVolume3` will compile - the cast satisfies all the necessary constraints to be checked. The code will still be safe (in particular, the cast always succeeds).

**D)**

```
double getVolume4(Vector? c) {
   if(c!=null && (c instanceof Vector3D)) {
      return c.volume();
   } else { return 0.0; }
}
```

solution

`getVolume4` won't compile for the first reason above - Java will complain. The code would be safe though. Note that the non-null type checker won't complain in either case, because of the new if-condition.

**E)**

```
double getVolume5(Vector? c) {
   if(c!=null && (c instanceof Vector3D)) {
      return ((!) c).volume();
   } else { return 0.0; }
}
```

solution

`getVolume5` won't compile, but is safe for the same reasons as `getVolume4`.

**F)**

```
double getVolume6(Vector? c) {
   if(c!=null && (c instanceof Vector3D)) {
      return ((Vector3D!) c).volume();
   } else { return 0.0; }
}
```

solution

`getVolume6` will compile and run safely.

## Task 2

Consider the following abstract class, representing a node of a singly-linked list:

```
public abstract class ListNode<X> {
   public abstract void setItem(X x);
   public abstract X getItem();
   public abstract ListNode<X> getNext();
}
```

Consider now the following implementation using a simple (acyclic) list:

```
public class AcyclicListNode<X> extends ListNode<X> {
   protected X item;
   protected AcyclicListNode<X> next;

   public AcyclicListNode<X> (X item) {
      this.item = item;
      this.next = null;
   }

   public void setItem(X x) { item = x; }
   public X getItem() { return item; }
   public AcyclicListNode<X> getNext() { return next; }
}
```

In this implementation, suppose that an empty list is represented simply by a `null` reference. Suppose that a further design intention of this implementation is that each node is guaranteed to store an `X` object in its item field.

**A)** Annotate the class `AcyclicListNode<X>` with appropriate non-null type annotations to express these design intentions as far as possible. You do not need any annotations from the Construction Types system (`free` or `unc` annotations).

solution

(Side note: the interaction of generic types and non-null types, e.g., the interpretation of a type `X!` if `X` can be instantiated with types that themselves include non-nullity expectations, is beyond the scope of the course, but in case you are worried, you can assume that the explicitly visible annotation `!` overrides any annotation in the instantiation for `X`, i.e., `X!` can still be safely assumed to always store a non-null value) The following class definitions express the design expectations:

```
public class AcyclicListNode<X> extends ListNode<X> {
   protected X! item;
   protected AcyclicListNode<X>? next;

   public AcyclicListNode<X> (X! item) {
      this.item = item;
   }

   public void setItem(X! x) { item = x; }
   public X! getItem() { return item; }
   public AcyclicListNode<X>? getNext() { return next; }
}
```

**B)** Now consider an alternative implementation using a cyclic list:

```
public class CyclicListNode<X> extends ListNode<X> {
   protected X item;
   protected CyclicListNode<X> next;

   public CyclicListNode<X> (X item) {
      this.item = item;
      this.next = this;
   }

   public void setItem(X x) { item = x; }
   public X getItem() { return item; }
   public CyclicListNode<X> getNext() { return next; }
}
```

In this implementation, the design intention is that every node will always have a next object in the list (sometimes itself). In this design, we choose to represent an empty list by a single node whose `next` field points to itself, but whose `item` field is `null`. All non-empty lists will be represented using only nodes whose `item` fields are non-null.

Annotate the class `CyclicListNode<X>` with appropriate non-null type annotations to express these design intentions as far as possible. You do not need any annotations from the Construction Types system (`free` or `unc` annotations).

solution

```
public class CyclicListNode<X> extends ListNode<X> {
   protected X? item;
   protected CyclicListNode<X>! next;

   public CyclicListNode<X> (X? item) {
      this.item = item;
      this.next = this; // default - maybe changed later
   }

   public void setItem(X? x) { item = x; }
   public X? getItem() { return item; }
   public CyclicListNode<X>! getNext() { return next; }
}
```

Note that we may decide to pass a non-null reference to `setItem`.

**C)** Now consider how to annotate the method signatures in `ListNode<X>` so that both implementations can be accommodated. Your solution should be compatible with the usual co/contra-variance rules for subclass method signatures.

solution

We have to pick suitable method signatures so that the implementing methods have valid overriding signatures in both classes above. This typically means strengthening the argument types and weakening the return types:

```
public abstract class ListNode<X> {
   public abstract void setItem(X! x);
   public abstract X? getItem();
   public abstract ListNode<X>? getNext();
}
```

## Task 3

In the Construction Types system, a field assignment $e_1.f = e_2$ is permitted if the usual subtyping holds, and if, in addition either $e_1$ has a free type, or $e_2$ has a committed type.

In particular (in terms of Construction Types), it is ok for an expression with committed type to be assigned to the field of an expression with committed type, and it is also ok for an expression of free type to be assigned to the field of an expression of free type. However, it is not permitted for an expression of unclassified type to be assigned to the field of an expression of unclassified type. Explain why not, giving an example of what would go wrong if we were to allow this.

solution

Because unclassified references are supertypes of the corresponding free and committed references, then if we were to allow this, we might "disguise" the assignment of a free reference to the fields of a committed reference. For example, the following code would then type-check, which is not sound:

```
public class C {
   C! f, g;
   public C(C! x) {    // x is committed, this is free
      unc C! y = x;     // cast committed to unclassified - ok
      unc C! z = this; // cast free to unclassified - ok
      y.f = z;          // assign unc to field of unc (?)
      this.g = x.f.g;   // what happens here?
      this.f = this;
   }
}
```

## Task 4

Consider the following three classes (declared in the same package):

```
public class Person {
   Dog? dog; // people might have a dog

   public Person() { }
}

public class Dog {
   Person! owner; // Dogs must have an owner
   Bone! bone;    // Dogs must have a bone
   String! breed; // Dogs must have a breed

   public Dog(Person owner, String breed) {
      this.owner = owner;
      this.bone = new Bone(this);
      this.breed = breed;
   }
}

public class Bone {
   Dog! dog;         // Bones must belong to a dog..

   public Bone(Dog toOwn) {
      this.dog = toOwn;
   }
}
```

**A)** Annotate the code with non-null and Construction Type annotations where they are necessary. Explain why the code now type-checks according to Construction Types.

solution

Here are the annotations for the first version of the code:

```
public class Person {
   Dog? dog;    // a person might have a dog

   public Person() { }
}

public class Dog {
   Person! owner;  // A dog must have an owner
   Bone! bone;     // A dog must have a bone
   String! breed;  // A dog must have a breed

   public Dog(unc Person ! owner, unc String ! breed) {
      this.owner = owner;
      this.bone = new Bone(this);
      this.breed = breed;
   }
}

public class Bone {
   Dog! dog;         // Bones must belong to a dog..

   public Bone(unc Dog ! toOwn) {
      this.dog = toOwn;
   }
}
```

Note that we choose the parameter to the construction of `Bone` to be unclassified - since it is public then it probably should be callable with a committed parameter from client code, but it is also called inside the body of the constructor of `Dog`, with a free parameter. Note that the returned reference from these two kinds of call will be different - committed in the former case, and free in the latter. For the `Dog` constructor, we can also choose to make the parameters unclassified. Although in this case we do not directly need to permit "free" arguments being passed to the constructor, we may as well be as permissive as possible. In general, if it is possible to type a constructor body using "unclassified" argument types then this should be the preferred choice of signature as it is the most permissive. Note that the same does not apply for method signatures, since any overriding method definitions are then also be forced to cope with unclassified arguments, which may be much less convenient than using committed ones.

**B)** Could we provide constructors for classes `Dog` and `Bone` with no parameters?

solution

It isn't reasonable to have constructors for `Dog` and `Bone` without parameters, since we need some way of initialising their non-null fields. Although it would be possible to do this by calling e.g., the `Person` constructor from the `Dog` constructor, this doesn't seem very intuitive (nor would it be easy to establish the intuitive invariants of the code - that a `Dog`'s owner refers back to the same `Dog`, etc.). In particular, if all of the constructors need to take no parameters, they would need to call each other infinitely. This is because, we can't set up a cyclic object structure without some kind of mutual initialization (in this case we

can only build an infinite object structure to satisfy the non-null requirements of all the objects).

**C)** Now, suppose a (possibly mad) scientist wants to extend the implementations of these classes with some genetic engineering. Firstly, we want to be able to "clone" a bone. We can add the following method to class `Bone` to make a copy of an existing bone, and assign it to another `Dog`:

```
public Bone clone(Dog toOwn) {
      return new Bone(toOwn);
   }
```

However, our scientist would like to go further, and be able to clone dogs. A cloned `Dog` should also have its bone cloned along with it, but may be assigned to a new owner: we add the following extra constructor and method to class `Dog`:

```
Dog(Dog toClone, Person newOwner) {
   this.owner = newOwner;
   this.breed = toClone.breed;
   this.bone = new Bone(this);
}

public Dog clone(Person toOwn) {
   return new Dog(this, toOwn);
}
```

However, our scientist would like to go still further, and be able to clone people. A cloned `Person` should also have its dog (if any) cloned along with it: we add the following extra constructor and method to class `Person`:

```
Person(Person toClone) {
   Dog? d = toClone.dog;
   if(d!=null) {
      this.dog = new Dog(d, this);
   }
}

public Person clone() {
   return new Person(this);
}
```

Annotate this extra code with appropriate non-null and Construction Types annotations. You should guarantee that each of the `clone` methods (which belong to the public interface) return a committed reference. You should ensure that your answers guarantee that all of the code type-checks - explain your choices.

Hint: think carefully about how constructor calls are typed, and what happens if the constructors are called in more than one situation.

solution

Here is the fully annotated code for the cloning case:

```
public class Person {
   Dog? dog;   // A person might have a dog

   public Person() { }

   Person(Person! toClone) {
```

```
        Dog d? = toClone.dog;
        if(d != null) {
            this.dog = new Dog(d, this);
        }
    }

    public Person! clone() {
        return new Person(this);
    }
}
public class Dog {
    Person! owner; // A dog must have an owner
    Bone! bone;    // A dog must have a bone
    String! breed; // A dog must have a breed

    public Dog(unc Person ! owner, unc String ! breed) {
        this.owner = owner;
        this.bone = new Bone(this);
        this.breed = breed;
    }

    Dog(Dog! toClone, unc Person! newOwner) {
        this.owner = newOwner;
        this.breed = toClone.breed;
        this.bone = new Bone(this);
    }

    public Dog! clone(Person! toOwn) {
        return new Dog(this, toOwn);
    }

}
public class Bone {
    Dog! dog;         // A bone must belong to a dog..

    public Bone(unc Dog ! toOwn) {
        this.dog = toOwn;
    }

    public Bone! clone(Dog! toOwn) {
        return new Bone(toOwn);
    }
}
```

Note that all parameters to the new constructors and methods need to have non-null type annotations, since they are each either dereferenced, used to initialize non-null-declared fields or passed on as further parameters to calls that require non-null parameters.

The `toClone` parameter of the new constructor of `Person` needs to be a committed parameter, otherwise when we dereference `toClone.dog` we will obtain an unclassified value, which will not be suitable to use as a parameter for the new `Dog` constructor.

The `toClone` parameter of the new constructor of `Dog` needs to be a committed parameter, since when a field is read from it, we need to obtain a result with a non-null type. However, the `newOwner` parameter of the new constructor of `Dog` needs to be an unclassified parameter. This is because this parameter is sometimes supplied from a free reference (in the new constructor of `Person`), and sometimes from a committed reference (in the clone method of `Dog`).

For similar reasons, the `toOwn` parameter of the constructor of `Bone` needs to be an unclas-

sified parameter (as was suggested for the previous part of the question). This is because this parameter is sometimes supplied from a free reference (in the new constructor of `Dog`), and sometimes from a committed reference (in the clone method of `Bone`).

This is an important usage of the unclassified types in the Construction Types system - they are useful for constructors which get called sometimes with free and sometimes with committed parameters. Recall that the type of a new expression is determined from the static types of the actual parameters at a particular call, and not from the formal parameters in the constructor signature. For example, in the `clone` method of the `Bone` class, the new expression `new Bone(toOwn)` is given a committed type because the actual parameter `toOwn` has a static type which is committed, despite the fact that the constructor argument type is declared as unclassified in its signature. This means that the same constructor can produce committed/free results depending on the particular arguments provided in each call (`new` expression). In particular, the return type of the `clone` method can be a committed reference, as required in the question (the same applies to all of the clone methods in the code, since they each call constructors with only committed arguments).

## Task 5

*(From a previous exam)*

Consider the following code in a Java-like language enriched with the non-null types system of the course:

```
class Node
{
   int depth;
   public Node! parent;
   public Node! left;
   public Node! right;

  Node(int d)
  { ... }

  ...
}
```

The constructor shown above, when invoked with a positive integer, as in

```
new Node(d)
```

must create a complete binary tree (type `Node!`) of depth `d` containing exactly $2^{d+1} - 1$ nodes. The root node has depth 0. The `depth` field of every node in the constructed tree must be initialized to the depth of that node in the tree. The `parent` field of the root node should point to the root node itself. Similarly the `left` and `right` fields of leaf nodes should point to the leaf nodes themselves.

**A)** Write the body of the constructor. You may introduce other constructors and methods. Make sure that you adhere to the rules of the non-null types system including construction types.

solution

Here is a possible implementation

```
Node(int d)
{
```

```
    depth = 0;
    parent = this;
    if(d == 0) {
        left=this;
        right=this;
    } else {
        left = new Node(d, 1, this);
        right = new Node(d, 1, this);
    }
}

Node(int goal, int d, free Node! p)    // can be unc Node!
{
    depth = d;
    parent = p;
    if(d == goal) {
        left=this;
        right=this;
    } else {
        left = new Node(goal, d+1, this);
        right = new Node(goal, d+1, this);
    }
}
```

**B)** Consider the following method:

```
void foo(unc Node! o)
{
    unc Node! x = new Node(2);
    free Node! y = new Node(2);
    Node! z = new Node(2);
    o.right = new Node(2);
}
```

Which of these assignments would typecheck? Explain.

solution

The type of `new Node(2)` is committed. This can be shown trivially, because no references are passed to the constructor.

From the assignments to local variables, the second one is not allowed because it violates the subtyping rules. The other two are allowed.

The fourth assignment is allowed. By the rules for assignments to fields, we know that a committed reference can be assigned to non-null fields of unclassified, free and committed objects.

## Task 6

Consider the following Java classes:

```
public class A {
    public static final int value = B.value + 1;
}

public class B {
    public static final int value = C.value + 1;
}
```

```
public class C {
   public static final int value = A.value + 1;
}
```

**A)** Will these classes compile? If not, how could we modify them so that they do?

solution

The classes will compile.

**B)** What would the output of running the following program be?

```
public class Program {
   public static void main(String[] args) {
      System.out.println(A.value);
      System.out.println(B.value);
      System.out.println(C.value);
   }
}
```

solution

When the program is run, the output will be:

```
3
2
1
```

This is because, starting to initialize `A` causes `B` to start being initialized which causes `C` to start being initialized (at which point Java realizes `A` has already started initialization and just carries on initializing `C`). When `C.value` gets assigned, `A.value` still contains the default value 0.

**C)** In what ways can you change the output of the program by reordering the statements?

solution

The class we first mention will always get loaded first, and so complete initialization last. By changing the order of the second two classes, we can vary the output between the one above, and:

```
3
1
2
```