ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 12
Initialization
December 14, 2018

Task 1

Consider a class `Vector`, representing a 2 dimensional vector, written in a Java-like language with non-null type annotations:

```java
public class Vector {
    public Number! x; // Remark: Number is a supertype 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()`:

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

    public Vector3D (Number! x, Number! y, Number! z) {
        super (x, y);
        this.z = 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)

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

B)

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

C)

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

D)

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

E)

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

F)

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

Task 2

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

```java
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:

```java
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 type system (`free` or `unc` annotations).

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 type system (`free` or `unc` annotations).

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.

Task 3

(From a previous exam)

Consider the following two different implementations of a cyclic list that use the construction type system taught in the course. The type system rejects both of them. The constructors are used to clone an existing list. In both cases we establish a link between a node and its clone.

A) Are there lines of code where we are trying to incorrectly assign to a field of a committed object? If so, in which implementation (*left* or *right*) and on which lines?

B) If we allowed these implementations to run, is it possible that a committed object would become not <u>locally initialized</u>, while a constructor is executing? If so, in which implementation (*left* or *right*) and on which line is there an assignment where this happens?

C) If we allowed these implementations to run, is it possible that a committed object would become not <u>transitively initialized</u>, while a constructor is executing? If so, in which implementation (*left* or *right*) and on which line is there an assignment where this happens?

D) Without changing the constructor signatures in any way, which two lines of the implementation on the *right* can you change and how, so that it typechecks in the construction type system and achieves the expected result? Write the line numbers and the new content of the lines.

```
1  class Node {
2   Node! next; // cyclic
3   Node? copy;
4   int value;
5
6   Node(int x)
7   {
8    next = this;
9    value = x;
10  }
11
12  Node( Node! other )
13  {
14   value = other.value;
15   other.copy = this;
16
17   if(other.next == other)
18    next = this;
19   else
20    next = new Node(other, other.next);
21  }
22
23  Node( Node! first, Node! other )
24  {
25   value = other.value;
26   other.copy = this;
27
28   if(other.next == first)
29    next = other.next.copy;
30   else
31    next = new Node(first, other.next);
32  }
33 }
```

```
1  class Node {
2   Node! next; // cyclic
3   Node? original;
4   int value;
5
6   Node(int x)
7   {
8    next = this;
9    value = x;
10  }
11
12  Node( Node! other )
13  {
14   value = other.value;
15   original = other;
16
17   if(other.next == other)
18    next = this;
19   else
20    new Node(this, this, other.next);
21  }
22
23  Node( free Node! first,
24        free Node! prev, Node! other )
25  {
26   value = other.value;
27   original = other;
28   prev.next = this;
29
30   if(other.next == first.original)
31    next = first;
32   else
33    new Node(first, this, other.next);
34  }
35 }
```

Task 4

In the construction type system, when we read from the field of an expression of committed type, we obtain a reference of committed type, i.e., if e_1 has a committed type then $e_1.f$ has a committed type as well. Similarly, if e_1 has an unclassified type then $e_1.f$ also has an unclassified type. However, if e_1 has a free type then $e_1.f$ does not have a free type, but instead has an unclassified type. Explain why the alternative choice would be unsound (given the existing rules of the system), giving an example of what would go wrong.

Task 5

With non-null types, any class type T can be annotated to explicitly declare non-nullity ($T!$) and possible-nullity ($T?$). In the construction type system, further variants of these types are

introduced, for `free`, "committed" (the default), and "unclassified" (`unc`) types. These types are all treated differently by the type system taught in the lectures.

A) Explain at least one difference between the treatments of a reference of type `T!` and a reference of type `T?`, giving an illustrative example.

B) Explain at least one difference between the treatments of a reference of type `free T!` and a reference of type `unc T!`, giving an illustrative example.

C) Explain at least two differences between the treatments of a reference of type `T!` (a committed reference) and a reference of type `unc T!`, giving illustrative examples.

D) Explain at least three differences between the treatments of a reference of type `T!` and a reference of type `free T!`, giving illustrative examples.

Task 6

(From a previous exam)

Consider the following code in a Java-like language enriched with the non-null type 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 type system including construction types.

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.

Task 7

Consider the following Java classes:

```java
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?

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

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

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