ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 12

Initialization

December 16, 2016

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; }
}
```

B)

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

C)

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

D)

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

E)

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

F)

```
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:

```
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).

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).

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

In the Construction Types 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. Similarly, if e_1 has an unclassified type then e_1`.f` 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 4

With non-null types, any class type `T` can be annotated to explicitly declare non-nullity (`T!`) and possible-nullity (`T?`). In the Construction Types 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 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.

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 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?

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);
   }
}
```

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