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:

```
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()`:

```
 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)

```
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 the 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 that the method `volume` is undefined. 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 that the method `volume` is undefined. 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 type 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;
      this.next = null;
   }

   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 type 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

(*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?

solution

left: 15, 26

Note that furthermore, line 29 does not type-check according to the non-null type system (not required as an answer here).

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

solution

It is not possible for a committed object to become not locally initialized.

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

solution

left: 15, 26

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.

solution

```
20: next = new Node(this, this, other.next);
33: next = new Node(first, this, other.next);
```

```
 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 $\texttt{e}_1$ has a committed type then $\texttt{e}_1\texttt{.f}$ has a committed type as well. Similarly, if $\texttt{e}_1$ has an unclassified type then $\texttt{e}_1\texttt{.f}$ also has an unclassified type. However, if $\texttt{e}_1$ has a free type then $\texttt{e}_1\texttt{.f}$ 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.

solution

Because anything (in terms of construction type annotations) can be stored in the fields of a free reference, when we read something back from such a field we do not have any guarantees. In particular, it is possible to store a committed reference in the field of a free reference, and if we could then read it back as free, this would be unsound. For example, the following code would type-check:

```
public class C {
   C! f, g;

   public C(C! x) {                    // x is committed, this is free
      this.f = x;                      // assigning committed to free - ok
      free C? y = this.f;              // this.f is free - ok
      if (y != null) {
          y.f = this;                  // y.f is free, this is free - ok
          this.g = x.f.g;              // what happens here?
      }

   }

   public C() { f = this; g = this; }
}

void client() {
   C! c = new C( new C() );
   c = c.g.g; // NullPointerException
}
```

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.

solution

For all solutions below, let us suppose that class `T` has the following field declarations:

```
T! f;
T? g;
```

If `x` is a reference of type `T!` then `x.f` is a permitted field read (without any if-checks /dataflow analysis), but if `x` is a reference of type `T?` then it is not.

Also, `x` can only be assigned to the `f` field of an object in the former case and not the latter (`T!` is a subtype of `T?` but not vice versa).

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.

solution

Suppose `y` is a reference of type `free T!`. If `x` is also a reference of type `free T!` then `x.f = y;` is a permitted field update, but if `x` is a reference of type `unc T!` then it is not.

Also, `free T!` is a subtype of `unc T!` but not vice versa.

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.

solution

If `x` is a reference of type `T!` then `x.f.f` is a permitted field read, since `x.f` also has the type `T!`. But if `x` is a reference of type `unc T!` then it is not permitted, since `x.f` has the type `unc T?`.

If `y` is a further reference of type `unc T!` then `y.f = x` is allowed when `x` has the type `T!` but not when `x` has the type `unc T!`.

Also, `T!` is a subtype of `unc T!` but not vice versa.

Furthermore, a constructor call `new C(x)` will return a committed type if `x` is committed, but a `free` type if `x` is unclassified.

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.

solution

If `x` is a reference of type `T!` then `x.f.f` is a permitted field read, since `x.f` also has the type `T!`. But if `x` is a reference of type `free T!` then it is not permitted, since `x.f` has the type `unc T?`.

If `y` is a further reference of type `unc T!` then `y.f = x` is allowed when `x` has the type `T!` but not when `x` has the type `free T!`.

Similarly, `x.f = y` is allowed when `x` has the type `free T!` but not when `x` has the type `T!`.

Furthermore, a constructor call `new C(x)` will return a committed type if `x` is committed, but a `free` type if `x` is `free`.

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.

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);
   }
}
```

The body of the first constructor could be replaced by a call to `this(d,0,this)`, assuming the definite assignment rule correctly determines that calling another constructor guarantees that all fields are assigned (in that other constructor). This question is, however, not addressed in the lecture, so the above solution is the "safe" solution.

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 7

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 be the output of running the following program?

```
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
```