Exercise 11
Ownership Types, Non-Null Types and Object Initialization
December 16, 2022

Note: Please wait until the lecture on the 15th of December before attempting tasks 6 and 7 because they exercise Construction Types which were not covered in the lecture yet.

Task 1   (from a previous exam)

Consider the following declarations:

```
class A {
    rep B first;
    rep B second;
}

class B {
    any A obj;
    peer B sibling;
}
```

Which of the following programs are allowed in the topological ownership system? For any program that is accepted in the topological system, is it also accepted in the owner-as-modifier system? Assume that none of the objects involved are `null`. Briefly explain each of your answers.

Program 1	Program 2	Program 3	Program 4

`rep B b;`
`...`
`b = b.sibling;`
	`peer A a; rep B b;`
	`...`
	`a = b.obj;`
		`any A a;`
		`...`
		`a.first.obj = a;`
			`peer A a;`
			`...`
			`a.first = a.first;`

> **solution**
>
> - **Program 1** is accepted in both systems.
>
> - **Program 2** is not accepted in the topological system (and neither in the owner-as-modifier system). It attempts to assign an `any` reference to a `peer` reference. `peer` is not a supertype of `any`.
>
> - **Program 3** is accepted in the topological system (it assigns `any` to `any`). However, it assigns to the field of a `lost` reference, which means that it is not accepted in the owner-as-modifier system.
>
> - **Program 4** is not accepted in the topological system (and neither in the owner-as-modifier system), because it assigns to a `lost` location.

Task 2   (from a previous exam)

The topological ownership system guarantees the following property: if a reference `a.f` to an object `b` is of ownership type `rep C`, then the object `a` is the *owner* of `b`. Moreover, each object has at most one owner.

The topological ownership system has a weakness: it does not support ownership transfer, which is desirable in many situations. Let us try to remedy this situation. Consider the following incomplete definition of a class T:

```
class T {
  public rep U f, g;
  ...
}
```

and the following program P, which, in addition to the field assignments, *implicitly also changes the owner* of the object e2.g from e2 to e1:

```
// implicitly: e2.g.owner = e1;
e1.f = e2.g;
e2.g = null;
```

where e1, e2 are two non-null objects of type T.

A) The code P is not allowed in the topological ownership system. Which rule disallows it?

> **solution**
>
> Assuming e1 is not syntactically equal to this, then e1.f must be lost and can therefore not be assigned to.

B) Write a code snippet C, such that executing $C; P$ is *guaranteed* to break the property described in the first paragraph of this task, *after P has finished executing. Do not rely on any specific implementation of the class* U (but you may assume the existence of a constructor without parameters). You may also add constructors to the class T.

Note that:

- you can assume that P is accepted by the compiler.

- *all* the code that *you* write *must respect the topological ownership system. P is the only code that breaks the rules.*

- you may *not* use reflection in your solution.

- you may *not* use P anywhere in the code that you write.

> **solution**
>
> We can add the following constructor to T:
>
> ```
> T() {
>    f = new rep U();
>    g = f;
> }
> ```
>
> and use the following code C:
>
> ```
> e1 = new peer T();
> e2 = new peer T();
> ```
>
> The invariant is broken after $C; P$, because e1 is the owner of e1.f, but the rep field f of a different object (e2) points to it.

Task 3

The ownership type system allows the following ownership modifiers: peer, rep, self, lost and any - to structure the object store and to restrict how references can be passed and used.

We want to extend the ownership type system by adding one more modifier, `down`. This modifier is introduced to denote references to objects in the same context as `this` or in the context (transitively) owned by an object in the same context as `this`.

A) Redraw the subtype relation diagram below to include the newly introduced type of the universe type system.
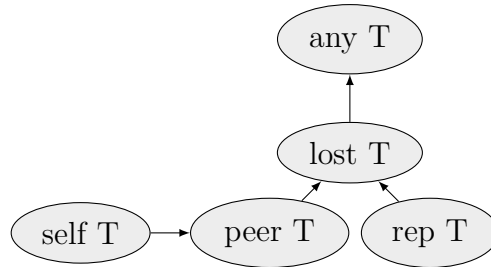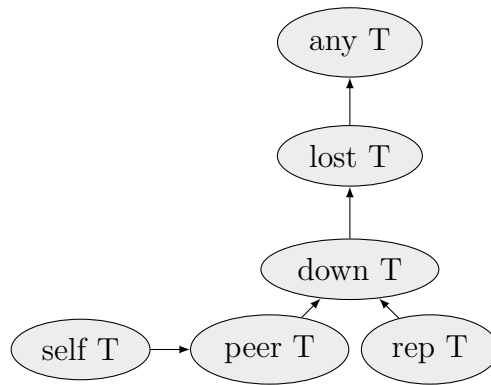
B) Define the viewpoint adaptation function ▶, such that it is the most specific in terms of the context information it conveys (i.e., it conveys as much context information as possible), by filling the table below (for a combination T_e ▶ T_f the modifier T_e specifies the row and the modifier T_f the column of the table used).

Recall that the viewpoint adaptation function ▶ is used, in particular, to determine the owner of an object referenced by a field access. More exactly, if the ownership modifier of `e` is T_e and the ownership modifier of a field `f` is T_f, then the ownership modifier assigned to the field access `e.f` is determined as T_e ▶ T_f. Note that this applies to field updates as well as field reads.

▶	peer	rep	any	down
self				
peer				
rep				
lost				
any				
down				

solution

Here is the table that defines the viewpoint adaptation as describing the most precise information possible about where such a reference may belong in the heap topology:

►	peer	rep	any	down
self	peer	rep	any	down
peer	peer	down	any	down
rep	rep	down	any	down
lost	lost	lost	any	lost
any	lost	lost	any	lost
down	down	down	any	down

Note that in the table above we over-approximate entries, in cases where we cannot describe precisely what we want. For example, rep►rep can be down, because down over-approximates the objects which can actually be stored in such a field. This is a true approximation - rep►rep is not allowed to store all the objects which can be referred to via down, only some of them. This means that we need to add extra restrictions on field assignments in the cases where we use down to over-approximate in this way.

If we *relax the requirement to have a most specific viewpoint adaptation function*, we can take an alternative approach which does not allow this kind of over-approximation; the modifier chosen could reflect precisely the requirements for a reference to be allowed to be stored in such a location, and thus avoid the need for extra requirements on the field assignment rule. Here is the table with this approach:

►	peer	rep	any	down
self	peer	rep	any	down
peer	peer	lost	any	down
rep	rep	lost	any	lost
lost	lost	lost	any	lost
any	lost	lost	any	lost
down	lost	lost	any	lost

In this case, perhaps surprisingly, cases such as rep►rep and down►down result in lost. This is because, choosing the answer down is not restrictive enough. In general, we have no way to express what is safe to assign to the down field of a rep receiver (down from our viewpoint includes objects above the rep, which should not be included), and similarly for a down receiver. This second approach is not very flexible; only rep and peer objects can ever be typed as down (via subtyping).

C) Consider the following example:

```
public class Node {
    rep Node c;
    down Node d;

    public void foo() {
        this.d.d = this;   // should this line typecheck?
        this.c.d = this.d; // should this line typecheck?
    }
}
```

Which of the assignments above should be allowed by the type system? Why?

---

solution

The example code shows two cases where the field updates should not be allowed, because we would allow a down field to point upwards (to this) in the ownership topology, and in the second, because we would allow a down field to point to some object which is considered

D) Assuming that you only need to enforce the topological constraints of the type system, how should the field update rule from lecture 6 slide 64 be adapted to the system extended with the `down` modifier? Do you need to make any changes?

> **solution**
>
> With the first (most precise) variant of the viewpoint adaptation function from **B** we have to make sure that the examples from part **C** do not type-check, as those field updates are unsafe. Therefore, we need to require that the result of the viewpoint adaptation is not `down`, except in the special case of the receiver being `self` or `peer`, and the field type being `down` (in these cases, the `down` result expresses precisely what is safe to assign to the location; it is not an over-approximation).
>
> With the second (avoiding over-approximation) variant of the viewpoint adaptation function from **B**, we do not need to make any changes to the field assignment rule, to guarantee the topological constraints of the type system.

Task 4

Consider the following classes, 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;
    }
}

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; }
}
```

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)

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

Task 5

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).

```
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 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 in a type-safe language.

> **solution**
>
> We have to pick suitable method signatures so that the implementing methods have valid overriding signatures in both classes above. This 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 6

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 the 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 one difference 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.

D) Explain at least two 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!`.

Task 7   (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 your answer.

> **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.