Concepts of Object-Oriented Programming
AS 2021

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 13

Ownership Types, Non-Null Types and Object Initialization

December 20 & 21, 2021

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;`

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?

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`.

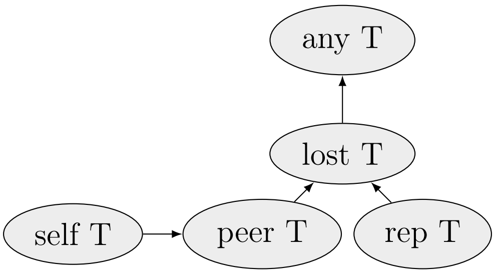
Note that:

- 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.

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 $T_e \blacktriangleright T_f$ 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 $T_e \blacktriangleright T_f$. Note that this applies to field updates as well as field reads.

▶	`peer`	`rep`	`any`	`down`
`self`				
`peer`				
`rep`				
`lost`				
`any`				
`down`				

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?

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?

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)

```
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 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).

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 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.

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.

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 one difference 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 two differences between the treatments of a reference of type `T!` and a reference of type `free T!`, giving illustrative examples.