ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 11

Ownernship Types, Non-null Types, Initialization Types

December 9, 2016

Task 1

Annotate the following program with appropriate ownership type modifiers (according to the topological ownership system) in order to maximize the buffer, the producer, and the consumer encapsulation:

```
class Producer {
 int[] buf;
 int n;
 Consumer con;
 Producer()
 {
  buf = new int[10];
 }
 void produce(int x)
 {
  buf[n] = x;
  n = (n+1)
   % buf.length;
 }
}
```

```
class Consumer {
 int[] buf;
 int n;
 Producer pro;
 Consumer(Producer p)
 {
  buf = p.buf;
  pro = p;
  p.con = this;
 }
 int consume()
 {
  n = (n+1)
   % buf.length;
  return buf[n];
 }
}
```

```
class Context {
 Producer p;
 Consumer c;

 Context(){
  p = new Producer();
  c = new Consumer(p);
 }

 public void run() {
  for(int i=-5; i<=5;
      ++i){
   p.produce(i);
   if(i%2 == 0)
    c.consume();
  }
 }
}
```

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 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 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 class* `U` (but you may assume the existence of a constructor without parameters). You may also add constructors to class `T`.

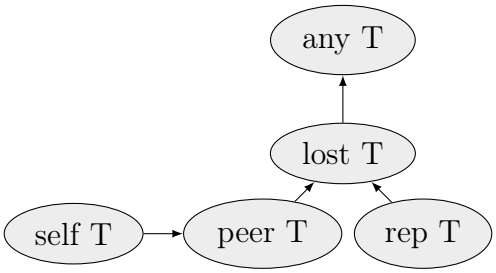
Note that

- 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 most specific (in terms of the context information it conveys) viewpoint adaptation function ▶ 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) 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?

D) In writing your answers for **B** and **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?
   }
}
```

Do your solutions disallow all invalid assignments?

Task 4

Assume the topological ownership framework taught in the course. Suppose that we want to create a class `SortedLinkedList`, with the internal invariant that the values stored in the nodes are sorted in ascending order.

```
package SortedLinkedList;
public class SortedLinkedList {
  private rep Node head;

  /// invariant head != null ==> head.sorted()
  ...
}
private class Node {
  protected peer Node next;
  protected int value;

  /// pure
  boolean sorted() {
    return next!=null ==> value < next.value && next.sorted()
  }
}
```

Suppose that all methods in `SortedLinkedList` are guaranteed to preserve the invariant of the class.

Furthermore, suppose that we want to create iterators for such lists (defined in the same package):

```
public class LinkedListIterator { private any Node current_item; ... }
```

A) Why is the field `current_item` annotated as `any`? What drawbacks would the other possible annotations have?

B) We want the following features:

(i) the invariant of a `SortedLinkedList` object is guaranteed to hold in any program, except when one of its methods executes

(ii) `LinkedListIterator` is a *modifying* iterator, i.e., it may change the value of the object it is pointing to

We can't have both features. Depending on whether or not we impose the "owners as modifiers" discipline, we can have either (i) or (ii). Argue why this is the case.

C) The fact that (i) and (ii) cannot both hold together is not surprising. A modifying iterator can break the invariant of the list it is iterating over. However, the "owners as modifiers" discipline may disallow harmless designs. Write a benign class (perhaps a restricted modifying iterator), which would not break the invariant of any object of `SortedLinkedList`, but still does not compile under "owners-as modifiers".

Task 5

[*From a previous exam*]

This question is about extending the non-null type system to handle arrays (ignoring initialization). Array types can have two type modifiers, declaring independently the nullity expectations for the array itself and the array elements. For any array type `T[]` the corresponding variants are `T?[]?`, `T?[]!`, `T![]?`, `T![]!` (the first modifier applies to the type of objects stored in the array, while the second modifier concerns the reference to the array object itself).

Assuming that we want to guarantee a statically sound approach to subtyping (that is, we want to enforce safety at compile time, without using runtime checks), explain whether or not the following subtype relations are safe. For each relation you consider unsafe, provide a code snippet illustrating that allowing such a subtype relationship would break the safety guarantees of the type system. For these unsafe cases, explain also what runtime checks could be made to restore safety.

- `T?[]! <: T?[]?`
- `T![]! <: T![]?`
- `T![]? <: T?[]?`
- `T![]! <: T?[]!`

Task 6

[*From a previous exam*]

Consider these two different implementations of a cyclic list that use the construction type system taught in the course. The type system rejects both of these implementations:

left

```
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 }
```

right

```
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 }
```

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 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?

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?

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.