ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 11

Owner as Modifier, Non-null Types, Initialization Types

December 5, 2014

Task 1

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 2 Topological Ownership

(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

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 in some other class we write the following method to calculate the length of the vector represented by a `Vector` object:

```
public double vectorLength(Vector c) {
    double x = c.x.doubleValue();
    double y = c.y.doubleValue();
    return Math.sqrt(x * x + y * y);
}
```

A) This implementation is unsafe - when executed it may throw exceptions. Why? Is this a reasonable behavior?

B) Add a pre-condition for the method, specifying what is required to be safe.

C) Suppose you are allowed to modify the signature of the method to include non-null type annotations. To what extent can you weaken the necessary pre-condition?

D) Suppose that you are also allowed to upgrade the class `Vector` to include reasonable non-null type annotations. How does this affect your previous answer? Do these changes to the class seem reasonable?

Task 4

(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 5 Cloning a Cyclic List

(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.