

Exercise 11

Owner as Modifier and Non-null Types

December 6, 2013

In-class Assessment: A subset of the questions from this sheet will be used for the in-class assessment. No notes are allowed during the assessment.

Task 1

Assume the topological ownership framework taught in the course. Suppose that we want to create a class `SortedListLinkedList`, 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 `SortedListLinkedList` 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 `SortedListLinkedList` 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). Demonstrate with an example what doesn't work in each 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 `SortedList`, but still does not compile under “owners-as modifiers”.

Task 2

We want to extend the ownership type system introduced in the course, to track how deep an object is within the ownership hierarchy of another. In particular, we introduce the ownership annotation `rep(n)` where `n` is a constant integer. The annotation `rep(0)` is equivalent to `peer`. If reference `o.f` is of type `rep(n)` and `g` is a field of type `rep`, then `o.f.g` is of type `rep(n+1)`. Conversely, if `g` is of type `rep(-1)` then `o.f.g` is of type `rep(n-1)`.

A Viewpoint Adaptation

What is the viewpoint adaptation relation in this system? Complete the following table:

	<code>rep(m)</code>	<code>rep</code>	<code>any</code>
<code>rep(n)</code>			
<code>rep</code>			
<code>any</code>			
<code>lost</code>			
<code>self</code>			

B Subtype Relations

What are the subtype relations between the ownership modifiers in this system?

C Field Assignments

What should the typing rules be for a field read `v = e.f` and for a field write `e.f = v` in the new system?

Task 3

(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 4

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);
}
```

- This implementation is unsafe - when executed it may throw exceptions. Why? Is this a reasonable behavior?
- Add a pre-condition for the method, specifying what is required to be safe.
- 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?
- 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?