ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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 `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). 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 `SortedLinkedList`, but still does not compile under "owners-as modifiers".

Solution

(a) If `current_item` were annotated as `rep`, then the owner of the node it refers to is the iterator itself. In this case, the iterator cannot iterate over a `SortedLinkedList` object l, because l also owns its nodes. The ownership topology allows at most one owner per object.

If `current_item` were annotated as `peer`, then, assuming that `current_item` has a list owner l, the owner of the iterator must also be l. This may be OK in topological ownership. However, if we add "owners as modifiers", the iterator's methods that traverse l cannot be called directly from an object outside l, which defeats the purpose of iterators.

(b) If we don't have "owners as modifiers", an object may get hold of an `any` reference to a node of the list, modify its `value` field, and break the invariant: (i) is not achieved. If we do have "owners as modifiers", then the iterator may not modify the value of the node it is pointing at, because it holds an `any` reference to it: (ii) is not achieved.

(c) We could have an iterator that performs the requested modification iff this does not violate the invariant:

```
public class LinkedListIterator {
  private any Node f;

  ... // some non-modifying methods

  public void modifyCarefully(int x) {
    if(f.value <= x && (f.next == null || x < f.next.value))
      f.value = x;
    // benign but does not type check
    // 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:

	rep(m)	rep	any
rep(n)			
rep			
any			
lost			
self			

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?

Solution

A Viewpoint Adaptation

	`rep(m)`	`rep`	`any`
`rep(n)`	`rep(n+m)`	`lost`	`any`
`rep`	`rep(1+m)`	`lost`	`any`
`any`	`lost`	`lost`	`any`
`lost`	`lost`	`lost`	`any`
`self`	`rep(m)`	`rep`	`any`

B Subtype Relations

Any `rep(n)` type is a subtype of itself, `lost`, and `any`. `lost` is a subtype of `any`. `self` is a subtype of `rep(0)`. `rep` is a subtype of `rep(1)`. There is no other subtype relation; in particular there is no relation between `rep(n)` and `rep(m)` if `n!=m`.

C Field Assignments

The usual subtyping rules, plus that the assignment to a `lost` reference is illegal.

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?[]!`

Solution

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

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

```
Object![]? x = new Object![1]?;
Object?[]? y = x;
if(y!=null) y[0]=null;
if(x!=null) x[0].toString();
```

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

```
Object![]! x = new Object![1]!;
Object?[]! y = x;
y[0]=null;
x[0].toString();
```

In both the last two cases, we need to check at runtime if a value stored in an array with dynamic non-null type for the elements stored in the array is not the `null` value. Alternatively, we can check at runtime if a value read from an array with dynamic non-null type is not the `null` value.

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?

Solution

- If `c` were `null`, the field dereferences `c.x` and `c.y` would generate exceptions. Furthermore, if `c.x` were `null` then method call `c.x.doubleValue()` would generate an exception. Similarly if `c.y` were `null`. There is no reasonable answer for the method to

return if it encounters null values - any attempt to deal with these cases would have to return some arbitrary value, since the question the method is meant to answer is undefined in these cases.

- `requires: c≠null ∧ c.x≠null ∧ c.y≠null`
- `public double vectorLength(Vector! c)` would make the following pre-condition sufficient: `requires: c.x≠null ∧ c.y≠null`
- By changing the types of the fields `x` and `y` to be `Number!` we could guarantee that no pre-condition would be required. This seems a reasonable change, since a `null Vector` doesn't seem to be meaningful anyway.