Prof. Dr. A. Poetzsch-Heffter
Mathias Weber, M.Sc.

University of Kaiserslautern
Department of Computer Science
Software Technology Group

# Advanced Aspects of Object-Oriented Programming (SS 2015)

# Practice Sheet 6 (Hints and Comments)

## Exercise 1 Immutability

a) The class `ImmList` is declared **`final`** so it is not possible to create subclasses that may break immutability. The methods `cons()`, `head()`, `tail()` and `isEmpty()` obviously do not change any state and just return part of the state. The method `length()` calculates the length of the list for each call. But since none of the other methods change the state this can not break the observational immutability. So the class `ImmList` seams to be observational immutable.

The problem is that this class has no implementation of `equals()` and `hashCode()` so it inherits the implementations of `Object`. These implementations compare the identity of the objects. The following code will result in **`false`**:

```
ImmList list1 = ImmList.EMPTY;
ImmList list2 = list1.cons(1);
ImmList list3 = list1.cons(1);
System.out.println("list1.equals(list2):␣" + (list1.equals(list2)));
```

This proofs that the implementation is not observational immutable according to the definition of the lecture. Adding `equals()` and `hashCode()` methods that compare and use only the elements in the list would make this an immutable list implementation.

b) The `CompactString` class is an implementation of a `String` that does not copy the content for `substring()` operations. Instead it keeps the reference to the `data` array of the string the `substring()` operation was called on.

The class is again **`final`** so subclasses can not break immutability. The methods `substring()`, `equals()` and `toString()` do not change the state of the object. The hash code of the object is memoized, meaning it is not computed when creating the instance. Instead it is computed when accessing the hash code and the value of the hash code is not valid (negative). This is not generally a problem, the class could still be immutable.

The problem lies in the relation between what the `compact()` method does and how the hash code is computed. The hash code is computed using the valid indices of the `CompactString`. The method `compact()` allows to get rid of the possibly unused big array of characters of the parent string by copying only the valid characters into an array local to the current `CompactString` object thereby changing the valid indices (initially the start index depends on the position of the substring, afterwards the start index in 0). This change of the internal state does not break immutability since the `equals()` method still returns the same result. The problem is that the hash code is invalidated after changing the state and since the indices have changed and the hash code depends on these indices the new hash code will be different to the old one. The following code shows the problem:

```
CompactString s = new CompactString("Some␣String␣in␣the␣house");
CompactString subs = s.substring(5, 10);
System.out.println("subs.hashCode():␣" + subs.hashCode());
subs.compact();
System.out.println("subs.hashCode():␣" + subs.hashCode());
```

First the hash code will be 626 and after the call to `compact()` the hash code will be 622.

## Exercise 2 Confined Types

| | `ConfinedList` | `ProofTreeNode` | `PTNIterator` |
|---|---|---|---|
| C1 | ok | ok | ok |
| C2 | ok | ok | ok |
| C3 | ok | see below | see below |
| C4 | see below | ok | ok |
| C5 | ok | ok | ok |
| C6 | ok | ok | ok |
| C7 | ok | ok | ok |
| C8 | ok | ok | ok |
| Confined | no | no | no |

**C4 for `ConfinedList`** ConfinedList inherits the method `iterator()`, which creates an object of the inner class `AbstractList.Itr` (in OpenJDK 1.6). This inner class captures the `this`-variable of `AbstractList`, the method `iterator()` can therefore not be anonymous, which breaks rule C4.

**C3 for `ProofTreeNode`** At line 19 in `ProofContainer` a `ProofTreeNode` is passed to a method that expects `Object`; at line 147 in `ProofTreeNode` widening to `Object` occurs.

**C3 for `PTNIterator`** In `ProofTreeNodeIt.getProofTreeNodes() PTNIterator` gets widened to `Enumeration`.

The `kacheck` tool makes the analysis based on the bytecode of the classes. Since the generic type information is erased during compilation, the list `children` in `ProofTreeNode` is taken to be a `ConfinedList<Object>`. The `add()` method takes a parameter of type `Object` then and as such the call in line 125 performs widening on an object of type `ProofTreeNode`.

The capturing of the **`this`** reference for `ConfinedList` is not detected by the tool.

# Exercise 3 Alias Modes

a) No, the annotation in not possible due to the implementation of equals. Line 507: `co.data` accesses the data field but the referenced array is part of the representation of the list `co`. Representation objects are only accessible by the owner and by objects of the same representation domain, but the representation domain of the current `this` is different from the domain of the list referenced by `co`.

b) With alias modes it is possible to control the access to objects which belong to other objects. The alias modes work on object level, i.e. an alias mode is always relative to an object. This allows to hide the representation of one object to another one, even if they are created by the same source (i.e. they are of the same class). The confined types only allow to control that all interactions that may happen with object of a confined type are coded inside the package. This guarantees, that all modifications to these objects are under the control of the package developer. In the Confined Types approach, encapsulation is seen from a static point of view, where as the alias modes (and ownership types in general) have a notion of encapsulation based on dynamic properties.