Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 10

Encapsulation, Aliasing, Readonly types, Ownership

December 9, 2022

Task 1 (from a previous exam)

In answering this task, do not use reflection, inheritance, and static fields or methods.

This task is concerned with reasoning about *non-modification* in a modular setting in the presence of aliasing.

Consider the following code:

```
package cell;
public class Cell {
   private int value;

   /// ensures get() == newValue
   public Cell(int newValue) { value = newValue; }

   /// ensures get() == newValue
   public void set(int newValue) { value = newValue; }

   /// pure
   public int get() { return value; }
}

package client;
import cell.*;
class Client{
   /// requires c1 != null
   /// requires c2 != null
   void setCells(Cell c1, Cell c2) {
      c1.set(1);
      c2.set(2);
      assert(c1.get() == 1);
   }

   void setCellsClient() {
      Cell c1 = new Cell(5);
      Cell c2 = new Cell(5);
      setCells(c1, c2);
   }
}
```

The objective of this task is to make sure that the assertion in the method `setCells` does not fail, using modular reasoning. The potential problem is that of determining whether the call `c2.set(2)` can affect the return value of `c1.get()`.

A) Modify the second line of the method `setCellsClient` (the initialization of `c2`) so that the assertion in method `setCells` fails. The precondition of `setCells` must still be satisfied by the modified version.

B) Add a precondition to `setCells` that will make the call from your version of the method `setCellsClient` illegal. The precondition should be such that the original call is legal. Remember that the precondition can only refer to the arguments of the method and to public fields and methods.

C) We now add a `clone` method to the `Cell` class:

```
/// ensures result != null
/// ensures result != this
/// ensures result.get() == get()
/// ensures get() == old(get())
public Cell clone() { return new Cell(value); }
```

We also add to the client the methods `left` and `right`, which use the `clone` method:

```
void left() {
    Cell c1 = new Cell(5);
    Cell c2 = c1.clone();
    setCells(c1, c2);
}
```

```
void right() {
    Cell c1 = new Cell(5);
    Cell c2a = new Cell(5);
    Cell c2 = c2a.clone();
    setCells(c1, c2);
}
```

Modify only the `Cell` class so that a call to `left` causes the assertion in `setCells` to fail, while a call to `right` does not cause the assertion to fail. You can add private and default access members and methods to the `Cell` class and add private classes to the `cell` package, and also modify the implementation of existing methods, but not change the public interface in any way. Your implementations must satisfy the existing contracts, including the one from task **B**.

D) Strengthen the precondition of the method `setCells` so that, with your modified `Cell`, the call from `left` would fail the precondition check, while the call from the method `right` would satisfy the precondition.

You can use the concept of the reach of an object, where, for an object `x`, `reach(x)` is defined as the the set of objects which are reachable from `x` — the set of objects which can be described by an access path `x.f1.f2. ... .fn` for some n and some sequence of field names `f1..fn` (we do not consider arrays in this task). All fields are considered, regardless whether they are public or private. You can also use set operations in your precondition.

Remember that the precondition of a method can refer only to the `this` object and the method's arguments, dereferencing of public fields, and call public pure methods.

E) In order to prove the correctness of the body of the methods `left` and `right`, when `setCells` has the stronger precondition from task **D**, we would have to strengthen the postcondition of the `clone` method of class `Cell`. Write a stronger postcondition to the method `Cell.clone` so that the bodies of the methods `left` and `right` can be proven modularly — i.e., without knowing the implementation of the `clone` method and other private details of the class `Cell`.

Task 2

Consider the following C++ class:

```
class Person {
    int money;
    Person *spouse;
public:
   Person(int m, Person *s) {
      if (!s) { spouse = NULL; }
      else { spouse = s; s->spouse = this; }
      money = m;
   }
   void f() const;
};
```

The method `f` promises not to make any changes to its receiver object. Provide an implementation for `f` that violates this claim. You are not allowed to use casts, nor to introduce any local variables.

Task 3 (from a previous exam)

In the `readonly`/`readwrite` type system, which of the following assignments is not type correct?

1. `x = y;` where `x` is `readonly` and `y` is `readwrite`
2. `x = y.f;` where `x` is `readwrite`, variable `y` is `readonly` and field `f` is `readwrite`
3. `x = y.f;` where `x` is `readwrite`, variable `y` is `readwrite` and field `f` is `readwrite`
4. `x = y.f;` where `x` is `readonly`, variable `y` is `readwrite` and field `f` is `readwrite`

Task 4

Consider the following classes:

```
class A {
   readwrite StringBuffer n1 = ...;
   readonly StringBuffer n2 = ...;
}

class B {
   readwrite A x;
   readonly A y;
   public B(readwrite A x, readonly A y) {
      this.x = x;
      this.y = y;
   }
}
```

Note that the `readwrite` annotations could have been omitted, since `readwrite` is the default; they are written explicitly here for clarity.

Check which of the following programs typecheck and explain why they do or do not typecheck.

Program 1

```
readwrite A a = new A();
readonly  B b = new B(a, a);
readwrite StringBuffer v = b.y.n1;
```

Program 2

```
readwrite A a = new A();
readwrite B b = new B(a, a);
readwrite StringBuffer v = b.y.n1;
```

Program 3

```
readwrite A a = new A();
readwrite B b = new B(a, a);
readwrite StringBuffer v = b.x.n1;
```

Program 4

```
readonly  A a1 = new A();
readonly  A a2 = new A();
readwrite B b = new B(a1, a2);
readwrite StringBuffer v = b.y.n1;
```

Program 5

```
readwrite A a1 = new A();
readonly  A a2 = new A();
readwrite B b = new B(a1, a2);
readonly  StringBuffer v = b.y.n1;
```

Program 6

```
readwrite A a1 = new A();
readonly  A a2 = new A();
readwrite B b = new B(a1, a2);
readonly  StringBuffer v = b.y.n2;
```

Task 5

Consider how we might extend readonly types to handle arrays. For an array of primitive types, one might want to declare that the array elements cannot be changed, using a declaration such as:

```
readonly int[] x;
```

which, similarly to the rules for accessing fields of objects, would forbid an update such as:

```
x[2] = 2; // error - x is declared with a readonly type
```

A) Should there be a subtyping relation (in either direction) between the types `readwrite int[]` and `readonly int[]`?

B) For arrays of reference types, there are two reasonable questions to consider for readonly typing. Firstly, just as for an array of primitive types, whether or not the array reference can be used for modifications. Secondly, whether the array elements can be used for modifications.

```
y[1] = ...;   // is this allowed?
y[1].f = ...; // is this allowed?
```

In order to express all possibilities, consider having two `readonly`/`readwrite` modifiers for an array type - the first to denote access to the array reference itself, and the second to denote access to the objects stored in its elements, e.g., `readonly readonly T[] y;` is the most restrictive possibility.

Consider what the semantics of `readonly readwrite T[] y;` could be. There are two reasonable choices here, depending on whether we regard accesses of the form `y[1].f` as accesses which go first via the array `y`, and then to the element `y[1]`, or consider them as accesses directly to `y[1]` (in a sense "skipping" `y`, and hence ignoring the first modifier).

For each of these two possible semantics, consider the following:

- Do all four combinations of modifiers express something different from one another?
- What subtyping relations (if any) would be reasonable between the four possible variants of a `T[]` type?

C) In the light of these questions, which of the two semantics seems the best choice?

Task 6

Consider the following method signatures:

```
peer Object foo(any String el);
peer Object foo(rep String el);
rep Object foo(any String el);
any Object foo(peer String el);
rep Object foo(peer String el);
```

Find all the valid pairs of signatures such that one overrides the other. Assume that overriding methods can have covariant return types and contravariant parameter types.

Task 7

In the following question we do not consider the owners-as-modifiers discipline. We are only concerned with the topology of the ownership type system.

Consider the assignment:

```
o.f = p.g;
```

and assume that `o.f` and `p.g` have the same static type.

A) The assignment is forbidden if `o.f` has ownership modifier `lost`. Show an example to demonstrate why we need this rule to preserve the topological invariant.

B) If the ownership modifier of `o.f` is `any`, then what are the requirements for the assignment to be legal?

C) If `o.f` has ownership modifier `lost`, can we upcast `o.f` to an `any` reference and make the assignment legal? Why (not)?