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.

solution

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

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.

solution

```
/// requires c1 != c2;
void setCells(Cell c1, Cell c2)
...
```

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**.

solution

```
package cell;
class Cell {
   /// ensures get() == newValue
   public Cell(int newValue) { value = new CellInt(newValue); }

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

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

   private Cell(CellInt ci) { value = ci; }

   private CellInt value;

   private class CellInt {
      private int value;

      CellInt(int newValue) { value = newValue; }
      int get() { return value; }
      void set(int newValue) { value = newValue; }
   }
}
```

The `clone` method now creates a new `Cell` that shares the representation (the `CellInt`), and so modifying the cloned or the original `Cell` also modifies the other.

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.

solution

```
   /// requires reach(c1) disjoint reach(c2);
   void setCells(Cell c1, Cell c2)
   ...
```

Now the reach of the arguments `c1` and `c2` are disjoint, so modifying one cannot affect the other in any way.

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`.

solution

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

Strengthening the postcondition of `Cell.clone` like that has the following consequences:

- The implementation of `Cell.clone` from subtask **C** can no longer be verified since it does not guarantee the new postcondition (the `reach` sets won't be disjoint)
- The bodies of the methods `left` and `right` should therefore verify (modularly), and indeed will: `Cell.clone`'s stronger postcondition now establishes the precondition of `setCells`

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.

solution

We can violate the claim by changing the receiver object `this` through the field `spouse`, for instance: `spouse->spouse->money = 0;`

This would only work if the `s` passed in the constructor is non-null.

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`

solution

Number 2 is not allowed - it casts from a `readonly` reference to a `readwrite` reference.

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;
```

solution

- **Program 1** does not compile since `b` is `readonly`, so `b.y.n1` is `readonly`, and we try to assign it to a `readwrite` variable.
- **Program 2** does not compile since field `y` in `B` is `readonly`, so `b.y.n1` is `readonly`, and we try to assign it to a `readwrite` variable.
- **Program 3** compiles! `b` is `readwrite`, `x` is `readwrite`, so `b.x` is also `readwrite`, `n1` is `readwrite`, so `b.x.n1` is also `readwrite`, and we assign `b.x.n1` to a `readwrite` variable.
- **Program 4** does not compile since `a1` is `readonly` and it is passed to the constructor of `B` as the first argument, while the constructor expects a `readwrite` variable.
- **Program 5** compiles! We can always assign something to a `readonly` variable.
- **Program 6** compiles! We can always assign something to a `readonly` variable.

In addition: for all the programs except 4, the first argument passed to the constructor of `B` is `readwrite`, and the second argument can be `readwrite` or `readonly` since a `readonly` argument is expected.

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[]`?

solution

`readonly int[]` is more restrictive than `readwrite int[]` (fewer operations can be performed with such a reference), hence `readwrite int[] <: 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?

solution

Considering `y[1].f` as an access which goes first via `y`, and then `y[1]`, we would obtain that:

- If the first modifier is `readonly`, all the accesses to elements of the array will be treated as `readonly`, since the `readonly` modifier for the array will be considered first. Therefore, the only interesting combinations are:
  (a) `readonly readonly` (equivalent to `readonly readwrite`)
  (b) `readwrite readonly`
  (c) `readwrite readwrite`

Note: the same approach is adopted when we have a `readonly` object variable and we access a `readwrite` field through it: the result would be `readonly`, since any access via a `readonly` reference is `readonly`.

- The reasonable subtyping relations are (b) `<:` (a) and (c) `<:` (a). The case (b) `<:` (a) corresponds to invariant array typing. The (c) `<:` (a) case corresponds to covariant array typing but it is sound since the array type in (a) is `readonly` and, thus, an array element type only appears in covariant position (e.g., `v := a[i]`).

  Note that the relation (c) `<:` (b) would also correspond to covariant array typing but it would not be sound since it would indirectly allow casting a `readonly` reference to a `readwrite` reference, as shown below:

  ```
  class P { String n; }

  class C {
      void client(readonly P p) {
          readwrite readwrite P[] w = new P[1];
          readwrite readonly P[] r = w;
          r[0] = p;         // legal since r[0] and p are readonly
          w[0].n = "...";   // legal since w[0] is readwrite
      }
  }
  ```

  The assignment in the third line of `client` is legal since we have `readwrite` as the first modifier of `r`. Moreover, note that `p` should not be modifiable within the `client` method, as it is passed as `readonly`. However, by allowing the alias in the second line of the method, we enable a way to change `p`. This is undesirable and unsound. The implicit cast from `readonly` to `readwrite` is done on `p` here.

Considering `y[1].f` as a direct access, we would obtain that:

- All the four different combinations have different semantics. With respect to the previous example, we would have that `readonly readonly` will allow only read accesses both on the array and on the elements stored in it, while with `readonly readwrite` we cannot assign elements in the array but we can write fields accessed via the array elements.

- The subtyping relations are:

  (a) `readwrite readonly <: readonly readonly`

  (b) `readonly readwrite <: readonly readonly`

  (c) `readwrite readwrite <: readonly readwrite`

  (d) `readwrite readwrite <: readonly readonly`

  Note that we still have that `readwrite readwrite` $\not<:$ `readwrite readonly`. This subtype relation is not reasonable; if it were allowed, then we could use the example from the first semantic to modify an object through a `readonly` reference.

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

solution

The second solution is more expressive than the first one, since it allows the developers to have more fine-grained control on the read and write accesses on arrays and on their elements. Thus, the second choice seems to be the best. However, it should be carefully considered whether such an approach (that would be different compared to the one adopted for objects and field accesses) may confuse the developers, and eventually create safety problems.

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.

solution

The general typing rules are `peer <: any` and `rep <: any` since `any` is less restrictive than `rep` and `peer`. Following these rules, we obtain that:

- `peer Object foo(any String el)` overrides `any Object foo(peer String el)`
- `rep Object foo(any String el)` overrides `rep Object foo(peer String el)`, that overrides `any Object foo(peer String el)`
- `peer Object foo(any String el)` overrides `peer Object foo(rep String el)`

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.

solution

The following code breaks the acyclicity requirement for the topology:

```
class C {
  rep C down;

  void foo() {
    down.down = this;
  }
}
```

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

solution

None. The assignment is always 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)?

solution

We cannot upcast a reference that is being assigned to. This is illegal according to the subtyping rules.