

Exercise 10

Aliasing, Readonly Types and Ownership Types

December 2, 2016

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 {
    ///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;}
    private int 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 in 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 `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 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 section 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

[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 3

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;
    }
}
```

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

Program 1 <code>readwrite A obj=new A();</code> <code>readonly B obj2=new B(obj, obj);</code> <code>readwrite StringBuffer v=obj2.y.n1;</code>	Program 2 <code>readwrite A obj=new A();</code> <code>readwrite B obj2=new B(obj, obj);</code> <code>readwrite StringBuffer v=obj2.y.n1;</code>
Program 3 <code>readwrite A obj=new A();</code> <code>readwrite B obj2=new B(obj, obj);</code> <code>readwrite StringBuffer v=obj2.x.n1;</code>	Program 4 <code>readonly A obj=new A();</code> <code>readonly A obj2=new A();</code> <code>readwrite B obj3=new B(obj, obj2);</code> <code>readwrite StringBuffer v=obj3.y.n1;</code>
Program 5 <code>readwrite A obj=new A();</code> <code>readonly A obj2=new A();</code> <code>readwrite B obj3=new B(obj, obj2);</code> <code>readonly StringBuffer v=obj3.y.n1;</code>	Program 6 <code>readwrite A obj=new A();</code> <code>readonly A obj2=new A();</code> <code>readwrite B obj3=new B(obj, obj2);</code> <code>readonly StringBuffer v=obj3.y.n2;</code>

Task 4

In this question assume no type-casts or static variables or fields are used.

The C++ language supports the `const` modifier for types, which tries to model a weak readonly type system.

A) The C++ type system does not ensure transitive readonly structures as the system shown in class. Informally explain which typing rule could be changed and how to ensure transitivity. Your rule should make sure that in the following example, the assignment to `t.b->a` fails.

```
class T {
    public:
        int a;
        T* b;
        T() {
            a = 0;
            b = this;
        }
};
```

```
int main()
{
    const T t;
    t.a = 5;           // Fails in standard C++
    t.b->a = 5;         // Works in standard C++
}
```

B) Considering the changes in the previous part, show an example where the method `n` does modify `x.f`. Is this a problem?

```
void n(const T& x, T& y){...}
```

C) The mutable modifier is used in C++ to denote a field that can be mutated also in `const` methods - meaning that its value does not affect the client visible behaviour of the object (such as caching the results of a time consuming calculation) - consider the following code:

```
class List{
    ...

public:
    ///ensures result >= 0
    int length() const {...}

    ///requires index >= 0 && index <length()
    int at(int index) const {
        if (index == lastSearch)
            return lastSearchResult;
        else
        {
            int result = atHelper(index);
            lastSearch = index;
            lastSearchResult = result;
            return result;
        }
    }

private:
    int atHelper(int index) const {...} //Time consuming
    mutable int lastSearch=-1;
    mutable int lastSearchResult=0;
}
```

In this part assume that the `const` modifier is transitive for both pointers and references. We try to prove correctness of the `at` method by showing that we get the same result regardless of the values of `lastSearch` and `lastSearchResult`. However, this requires a stronger class invariant - give such an invariant, assuming that `atHelper` is pure (and does not modify even mutable fields).

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 relationship (in either direction) between 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] = y[2];           // is this allowed?  
y[1].f = y[2].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 `readwrite readonly 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 relationships (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.

Task 7

[From a previous exam]

Consider the following declarations:

```
class A  
{  
    rep B first;  
    rep B second;  
}  
class B  
{  
    any A obj;  
    peer B sibling;  
}
```

Which of the following programs are allowed in the topological ownership system? For any program that is accepted in the topological system, is it also accepted in the owner-as-modifier

system? Assume that none of the objects involved are null. Briefly explain each of your answers.

Program 1	Program 2	Program 3	Program 4
<code>rep B b; ... b = b.sibling;</code>	<code>peer A a; rep B b; ... a = b.obj;</code>	<code>any A a; ... a.first.obj = a;</code>	<code>peer A a; ... a.first = a.first;</code>