ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 10

Object Structures and Aliasing

November 27, 2015

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.

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

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

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);}
```

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`

solution

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

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

```
readwrite A obj=new A();
readonly  B obj2=new B(obj, obj);
readwrite StringBuffer v=obj2.y.n1;
```

Program 2

```
readwrite A obj=new A();
readwrite B obj2=new B(obj, obj);
readwrite StringBuffer v=obj2.y.n1;
```

Program 3

```
readwrite A obj=new A();
readwrite B obj2=new B(obj, obj);
readwrite StringBuffer v=obj2.x.n1;
```

Program 4

```
readonly  A obj=new A();
readonly  A obj2=new A();
readwrite B obj3=new B(obj, obj2);
readwrite StringBuffer v=obj3.y.n1;
```

Program 5

```
readwrite A obj=new A();
readonly  A obj2=new A();
readwrite B obj3=new B(obj, obj2);
readonly  StringBuffer v=obj3.y.n1;
```

Program 6

```
readwrite A obj=new A();
readonly  A obj2=new A();
readwrite B obj3=new B(obj, obj2);
readonly  StringBuffer v=obj3.y.n2;
```

solution

- **Program 1** does not compile since `obj2` is `readonly`, so `obj2.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 `obj2.y.n1` is `readonly`, and we try to assign it to a `readwrite` variable.

- **Program 3** compiles! `obj2` is `readwrite`, `x` is `readwrite` (so `obj2.x` is `readwrite`), `n1` is `readwrite` (so `obj2.x.n1` is `readwrite`), and we assign `obj2.x.n1` to a `readwrite` variable.
- **Program 4** does not compile since `obj` 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 expect 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 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. Show which typing rules could be changed and how to ensure transitivity (consider both pointers and references). Does this ensure that `x.f` is not modified in the method `m`?

```
class C{
   public: int f = 0;
}

void m(const C& x){...}
```

solution

The change that is needed is in the typing of field dereference - so that dereferencing a pointer/reference field of a `const` type gives a pointer/reference to `const`. This would prevent `m` from modifying `x.f` as transitive `const`ness is ensured.

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 C& x, C& y){...}
```

solution

`n` can modify `x.f` through aliasing - for example:

```
void g()
{
   C& c = *new C();
   assert(c.f==0);
   n(c,c);
   assert(c.f==0); //fails
}

void n(const C& x, C& y){
   y.f=1;
}
```

This is not a problem, as the only guarantee the system gives is that no modification is done through `const` objects.

C) The `mutable` modifier is used in C++ to denote a field that can be mutated also in `const` objects - 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).

solution

We could add the class invariant: `lastSearch>=0 ==> (lastSearch<length() && lastSearchResult == atHelper(lastSearch))`.

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[]`?

solution

`int[] readonly` is more restrictive than `readwrite int[]` (fewer operations can be performed with such a reference) so we could have `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] = 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?

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`

  (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`.

- (a) is more restricted than (b), and (b) is more restricted than (c). So the reasonable subtyping relations are (c) `<:` (b) `<:` (a)

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 have that we cannot assign elements in the array but we can write fields accessed via the array elements.

- The subtyping relations already pointed out still work. In addition we could have

  (a) `readonly readwrite <: readonly readonly`

  (b) `readwrite readwrite <: readonly readwrite`

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 developer 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.

solution

The general typing rules are `peer <: any` and `rep <: any` since `any` is more 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

[*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**

`rep B b;`
`...`
`b = b.sibling;`
	`peer A a; rep B b;`
	`...`
	`a = b.obj;`
		`any A a;`
		`...`
		`a.first.obj = a;`
			`peer A a;`
			`...`
			`a.first = a.first;`

solution

- **Program 1** is accepted in both systems.
- **Program 2** is not accepted in the topological system (and neither in the owner-as-modifier system). It attempts the assignment of an `any` reference to a `peer` reference. `peer` is not a super-type of `any`.
- **Program 3** is accepted in the topological system (it assigns `any` to `any`). However, it assigns to the field of a `lost` reference, which means that it is not accepted in the owner-as-modifier system.
- **Program 4** is not accepted in the topological system (and neither in the owner-as-modifier system), because it assigns to a `lost` location.