ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 10

Aliasing and Readonly Types

December 1, 2017

Task 1

Data structures often intentionally share aliases. For instance, consider the following Java class:

```
class ArrayList<T> {
   private T[] elements=...;
   private int lastEl=0;
   public T get(int i) {return elements[i];}
   public int size() {return lastEl;}
   public void add(T el) {elements[lastEl++]=el;}
}
```

Imagine that this class is extended as follows

```
class Coordinates {
   int x, y;
   public Coordinates(int xx, int yy) {x=xx; y=yy;}
}

class CList extends ArrayList<Coordinates> {
   /// invariant ∀ i:int | 0 ≤ i ∧ i < size() ⇒ get(i).x > get(i).y
   public void add(Coordinates el) {
      if(el.x>el.y) super.add(el);
   }
}
```

A) Write a program that breaks the invariant of `CList`.

B) How can we fix this problem?

C) Is it possible to fix it without allocating new objects (either directly or indirectly), that is, without consuming additional memory? What new problems might arise from your changes?

D) Discuss the benefits and the drawbacks of using alias sharing in data structures.

Task 2

The following Java classes, all part of the `security` package, were written by an unexperienced programmer and contain a number of issues:

```
package security;

public class User {
   public String name;
   public String password;
```

```
    public User(String name, String password) {
        this.name = name;
        this.password = password;
    }
}

public class LoginException extends RuntimeException {
    public User problemUser;
    public LoginException(String message, User problemUser) {
        super(message);
        this.problemUser = problemUser;
     }
}

public class Login {
    private List<User> users = new LinkedList<User>();
    public void registerUser(User u) {
        if (u == null || u.name == null || u.password == null
              || u.name.isEmpty() || u.password.isEmpty()) return;
        users.add(u);
    }

    // Returns true if the user 'u' was successfully logged in.
    // Otherwise returns false or throws an exception.
    public boolean login(User u) throws LoginException {
        if (u == null) return false;
        User current = null;
        try{
            for(User registered : users) {
                boolean nameEqual = registered.name.equals(u.name);
                current = registered;

                if (nameEqual) {
                    if (registered.password.equals(u.password))
                        return true;
                }

                if (nameEqual)
                    throw new LoginException("Invalid password for user",u);
            }

            return false;
        }
        catch(Exception e) {
            throw new LoginException("Invalid user",current);
        }
    }
}
```

The `malicious` method is in a different package:

```
void malicious(Login l) { ... }
```

Assume the `Login` object that is passed into the method already has registered users.

A) Complete the body of the `malicious` method so that you manage to log-in as an already existing user. You do not know any names or passwords of existing users. Do not use reflection.

B) Is it possible to fix the problem under the following restrictions? In each of these cases, explain how you can prevent the malicious login or why it is not possible.

- only modifying the `User` class?

- only modifying the `LoginException` class?
- only modifying the `registerUser` method?
- only modifying the body of the `for` loop inside the `login` method?

Task 3

[*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 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 4

[*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 5

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 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;
```

Task 6

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) Unlike the type system shown in class, the C++ type system does not ensure transitive readonly structures. Informally explain which typing rule could be changed and how in order 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.a`. 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 (e.g., fields used for 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 7

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?