ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 10

Aliasing and Readonly Types

November 30, 2018

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`.

solution

The invariant can be broken by exploiting the fact that `CList` captures and stores `Coordinates` objects.

```
CList list=new CList();
Coordinates c=new Coordinates(2, 1);
list.add(c);
c.x=0;
```

B) How can we fix this problem? What is the drawback of such a fix?

solution

To fix `CList` we need two things

- We need to clone the `Coordinates` element before storing it.

```
public void add(Coordinates el) {
    if(el.x>el.y) super.add((Coordinates) el.clone());
}
```

- We also need to clone the `Coordinates` element before returning it, as otherwise we leak a reference that could be modified.

```
public Coordinates get(int i) {
    return (Coordinates) super.get(i).clone();
}
```

The drawback of such an approach is that we create a copy of all the elements stored in the list. It is not possible to make sure the invariant is preserved without creating objects that are only in the current `CList` object.

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?

solution

A possible solution would be to have final fields in class `Coordinates`. This would ensure that the invariant cannot be broken, but it requires the allocation of new objects each time we want to modify the fields. For instance, the following code:

```
Coordinates c=new Coordinates(2, 1);
c.x=0;
```

would have to be re-written to

```
Coordinates c=new Coordinates(2, 1);
c=new Coordinates(0,1);
```

which allocates a new object even though this is not necessary (since the object pointed by `c` is not shared, and so changing its fields cannot break the invariants of other objects).

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

solution

The main benefit of alias sharing in data structures is to minimize the consumption of memory. In addition, we may want to share aliases on data structures, for instance, in order to further update the content of an element in a list. The main drawback is that alias sharing does not allow us to reason locally about the objects stored in the data structure, since clients could retain references to objects they store in the data structure, and might therefore modify the contents of these objects after they were stored.

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.

solution

The body of the `malicious` method could look like this:

```
void malicious(Login l) {
   User u = new User("user","pass");
   l.registerUser(u);
   u.name = null;

   try {
      l.login(u);
   }
   catch(LoginException e) {
      boolean success = l.login(e.problemUser);
      //Logged in as the user that was registered before user u
   }
}
```

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?

solution

- We could make both fields of `User` have the default (package) access:

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

Therefore, code outside the package will not be able to change existing `User` objects and the `malicious` method could not cause the exception as before.

- only modifying the `LoginException` class?

solution

The `LoginException` class currently captures the value of the problematic user. Instead it could create a new user that has the same name as `problemUser` but hides the password.

```
public class LoginException extends RuntimeException {
   public User problemUser;
   public LoginException(String message, User problemUser) {
      super(message);
      this.problemUser = new User(problemUser.name, "****");
    }
}
```

This way, even if an exception is thrown, that refers to the wrong user name, the user's password will not be leaked.

- only modifying the `registerUser` method?

solution

We can change the `registerUser` method so that it does not capture its argument:

```
public void registerUser(User u) {
   if (u == null || u.name == null || u.password == null
         || u.name.isEmpty() || u.password.isEmpty()) return;
   users.add(new User(u.name, u.password));
}
```

Now we would not be able to modify the internal structure of the `Login` class by modifying the user we just registered in the `malicious` method.

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

solution

This for loop actually contains a bug which allows the exploit to work. To fix it we must move the assignment to the `current` variable to the beginning of the loop:

```
for(User registered : users) {
   current = registered;
   boolean nameEqual = registered.name.equals(u.name);

   ...
}
```

In the original code we were able to cause an exception regarding a particular user, but report the previous user as an invalid, since `current` was not updated yet. This is no longer the case.

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.

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);}
```

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 disjoined)
- 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 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`

solution

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

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;
```

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 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 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++
}
```

solution

The problem is that `t.b` is a constant pointer `T * const`, not a constant pointer to a constant `T const * const`. The change that is needed is in the typing of field dereferences, so that dereferencing a pointer field of a `const` type gives a pointer to `const`. This would prevent `main` from modifying `t.a` through `t.b` as transitive `const`ness is ensured.

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){...}
```

solution

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

```
void g()
{
   T c = T();
   assert(c.a==0);
   n(c, c);
   assert(c.a==0); //fails
}

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

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

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).

solution

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

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[]`?

solution

`readonly int[]` 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 `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 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`.

- 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 that it would not be sound since it would indirectly allow casting a `readonly` reference to a `readwrite` reference:

```
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;
        w[0].n = "...";
    }
}
```

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.