ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 14

Self-Study Exercise Sheet

NOTE: This exercise sheet will not be discussed in an exercise session. We publish it together with the solution to help you better prepare for the exam. If you have any questions, please submit them for the Q&A session.

Subtyping and Behavioral Subtyping

Task 1

Assume the following class definitions in a nominally typed language:

```
class A {...}
class B extends A {...}
```

Consider now the following two classes:

```
class Super {
  B foo(B x) { return x; }
}

class Sub extends Super {
  A foo(A x) { return x; }
}
```

According to the rules presented in the lecture, this subtyping is illegal. Briefly explain why this is the case. However, considering the substitution principle, this subtyping is *safe*. Why?

Task 2

Consider the class `X` and its only method `foo`, where `ZZZ` is a placeholder for a class name:

```
class X {
   /// requires x > 0 ∧ (¬∃i,j: int | 2 ≤ i,j ≤ x ∧ i * j = x)
   /// ensures result > 0 ∧ result % 2 = 0
   int foo(final int x){ return (new ZZZ()).foo(x); }
}
```

Which of the four classes below could be substituted for `ZZZ` such that no contracts will be violated?

(a)
```
class A {
      /// requires x ≥ 0
      /// ensures result = x + 1
      int foo(final int x) {...}   }
```

(b)
```
class B {
      /// requires true
      /// ensures result % 2 = 0
      int foo(final int x) {...}   }
```

(c)
```
class C {
      /// requires x % 2 = 1
      /// ensures result = x + 1
      int foo(final int x) {...}    }
```

(d)
```
class D {
      /// requires true
      /// ensures result = x * (x + 1)
      int foo(final int x) {...}    }
```

Inheritance, Dynamic Method Binding, Multiple Inheritance, and Linearization

Task 3

Consider the following Java classes and interfaces:

```
public interface IA { IA g(IA x); }

public interface IB extends IA { IB g(IA x); IA g(IB x); }

public interface IC extends IA { IC g(IB x); }

class B implements IB {
   public IB g(IA x) { System.out.print("B1 "); return null; }
   public IC g(IB x) { System.out.print("B2 "); return null; }
}

class C implements IC {
   public IC g(IA x) { System.out.print("C1 "); return null; }
   public  C g(IB x) { System.out.print("C2 "); return null; }
}


class Main{
   public static void main(String[] args) {
      B  b  = new B();
      C  c  = new C();
      IA a1 = b;
      IA a2 = c;

      IA r1 = a1.g(a2);
      IA r2 = a2.g(b);
      IC r3 = b.g(b);
      IA r4 = c.g(a2);
      C  r5 = c.g(b);
   }
}
```

What is the output of the execution of the `Main.main` method? Explain your answer.

Task 4

Consider the following C++ code:

```
class Person {
   bool likesDiamonds;

   public: Person (bool l) { likesDiamonds = l; }
};

class Programmer : virtual public Person {
```

```
    public: Programmer () : Person (false) {}
            // diamonds are a programmer's worst enemy
};
```

It is expected that `!likesDiamonds` is an invariant for the class `Programmer`. Use inheritance to break this invariant, without altering the above code.

Task 5

Write three C++ classes:

- A class `Queue` that represents a queue of integers and has an `enqueue` and a `dequeue` method
- A subclass of `Queue` that also stores (and allows clients to retrieve) the current sum of all items in the queue, using the `enqueue` and `dequeue` methods
- A subclass of `Queue` that also stores (and allows clients to retrieve) the current product of all items in the queue, using the `enqueue` and `dequeue` methods

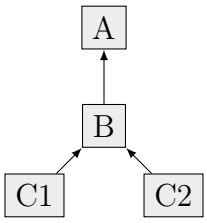
We would now like to have a class that supports both functionalities (i.e., stores and allows clients to retrieve both the sum and the product of all the items in the queue).

- Suppose that we use multiple inheritance and override the `enqueue` and `dequeue` methods of the new class, such that the new methods call the `enqueue` and `dequeue` methods of both of the old classes. Are there any problems with this approach?
- How can you solve this problem in Scala, using traits? Does this fix the above-mentioned problems from C++?

Bytecode Verification

Task 6

Consider the following type hierarchy:



Suppose that the method `f` of class `E` has the following signature:

```
A f(boolean b1, boolean b2);
```

and there are three local variables `x`, `y`, `z`. The maximal stack size is equal to 1.

The method `f` contains the following code snippet:

```
 0: iload 1
 1: ifeq  22
 4: iload 2
 5: ifeq  12
 8: aload 3
 9: goto 14
12: aload 4
14: astore 3
15: aload 5
```

```
17: astore 4
19: goto 0
22: aload 3
23: areturn
```

It is known that the state at the beginning of the snippet is:

```
([], [E,boolean,boolean,C1,C2,A])
```

Note: In this example, `ifeq x` pops an integer from the stack and jumps to line `x` if the integer is equal to zero.

A) Verify that the code snippet is type safe.

B) Provide the minimal type information that enables the type checking algorithm (i.e., the algorithm that does not perform a fixpoint computation) to verify the bytecode.

Task 7

The bytecode type inference algorithm rejects a verified program if there are different stack sizes for input values of a join point.

A) Provide a bytecode program that is rejected because of this limitation but that does not cause runtime errors.

B) Is it possible to construct a bytecode verification algorithm that avoids this limitation? If yes, then provide an updated algorithm. If no, then show that it cannot be done.

C) How serious is this restriction from a pragmatic perspective?

Parametric Polymorphism

Task 8

Consider the following Java code:

```
class Box<T extends Number> {
   private T t;

   public void set(T t) { this.t = t; }
   public T get() { return t; }
}

class Main {
   public static void main(String[] args) {
      Box<Number> b = new Box<_______>();
      b.set(new _______);
      _______ c = b.get();
      System.out.println( c );
   }
}
```

and recall that `Integer <: Number <: Object`. How can you fill in the blanks in the `Main.main` method so that the code compiles and executes successfully?

(a) `Integer, Integer(9), Integer`

(b) `Integer, Integer(9), Object`

(c) `Number, Integer(9), Integer`

(d) `Number, Integer(9), Object`

(e) None of the above

Task 9 (extended version of a previous exam question)

Consider the following Java code:

```
interface Food {}
interface Grass extends Food {}
interface Meat extends Food {}

abstract class Animal<F extends Food> implements Meat {
   abstract void eat(F food);
   F getLunchBag(){ return lunchBag; };
   F lunchBag;
}

final class Sheep extends Animal<Grass> { void eat(Grass f) {} }
final class Wolf extends Animal<Meat>   { void eat(Meat  f) {} }

class Cage { //You are allowed to modify this class
   Cage(Animal<?> animal){ this.animal = animal; }
   Animal<?> getAnimal() { return animal; }
   Animal<?> animal;
}

class Zoo {
   void feedAnimal(Cage cage){ /*code given in each section*/ }

   <F extends Food> void feed(F food, Animal<F> animal) {
      animal.eat(food);
   }

   void manage(){ /*your code here*/ }
}
```

Clearly a `Wolf` can eat a `Sheep` but not the other way around. In the following subtasks we explore if relaxing some of the Java type rules can lead to a situation where a `Sheep` can eat a `Wolf` - that is, the method `eat` is called on an object of the dynamic type `Sheep` with an argument object of the dynamic type `Wolf`. All the code you give in the answers to the following sections is in the same package as the code above, and must type check in standard Java. The top method that is called in all sections is `Zoo.manage`. You can assume that method call arguments are evaluated left to right, and that the program is sequential. You are not allowed to use reflection, raw types, or type-casts.

A) Assume the following body of `Zoo.feedAnimal(Cage cage)`, which is rejected by the Java type checker:

```
{ feed(cage.getAnimal().lunchBag, cage.getAnimal()); }
```

Make a `Sheep` eat a `Wolf` assuming the body of `feedAnimal` is exempted from the type checker.

Show all necessary code. You are only allowed to change the `Cage` class and provide the body of the `Zoo.manage` method.

B) Assume the following body of `Zoo.feedAnimal(Cage cage)`, which is rejected by the Java type checker:

```
{ feed(cage.animal.getLunchBag(),cage.animal); }
```

Can you make a `Sheep` eat a `Wolf` if the body of `feedAnimal` is exempted from the type checker? If so, show all necessary code, otherwise explain why not. You are only allowed to change the `Cage` class, provide the body of the `Zoo.manage` method, and add new classes.

C) Answer the question in the previous section, assuming the field `Cage.animal` is `final`. Explain your answer. Reminder: Java's `final` fields can be assigned to only in the constructor of the class that declares them.

D) Assume the following body of `Zoo.feedAnimal(Cage cage)`, which is rejected by the Java type checker:

```
{ feed(cage.animal.lunchBag, cage.animal); }
```

Can you make a `Sheep` eat a `Wolf` if the body of `feedAnimal` is exempted from the type-checker? If so, show all necessary code, otherwise explain why not.

E) Which of the above cases, that is safe in the sequential case, is unsafe in a multithreaded program? For each such case, explain what can happen in the multithreaded case that cannot happen in the sequential case.

Information Hiding and Encapsulation

Task 10

Suppose that we have a language with the information hiding rules of Java, but with structural subtyping. What should be the subtyping relations between the following three classes?

```
class A { int foo(); }
class B { protected int foo(); }
class C { public int foo(); }
```

Task 11

Consider the class `Hour`, defined as follows:

```
public class Hour {
   protected int h = 0;
   /// invariant h >= 0 && h < 24

   public void set(int h) {
      if (h >= 0 && h < 24) this.h = h;
   }
}
```

What is the external interface of `Hour`?

Can we extend the code, without changing the class, so that the invariant is broken? If yes, provide an example, and propose how to fix the class.

Task 12 (from a previous exam)

Consider the following Java program consisting of two packages `BTC` and `B2X`:

```
1  package BTC;
2
3  public class Chain {
4
5      /// ensures result <= 2
6      _______ int max_size() {
7          return 2;
8      }
9  }
10
11 package B2X;
12 import BTC.*;
13
14 public class Chain2x extends Chain {
15
16     /// ensures result <= 4
17     protected int max_size() {
18         return 4;
19     }
20 }
```

A) What is the *most permissive* access modifier for the method `max_size()` in class `Chain` such that class `Chain2x` is a *behavioral subtype* of `Chain`? Assume that we *do not use* specification inheritance. **Fill the blank above with your answer**. Explicitly write `default` for a default access modifier. Write `none` if no choice of access modifier allows `Chain2x` to be a behavioral subtype of `Chain`.

B) We now add a class `Block` and a subclass `Block2x` to package `BTC`:

```
1  package BTC;
2
3  public class Block {
4
5      protected int num;
6      /// invariant: 1 <= num
7
8      public Block(int n) {
9          num = (n < 1 ? 1 : n);
10     }
11
12 }
13
14 public class Block2x extends Block {
15
16     /// invariant: 2 <= num
17     protected Block pred;
18     /// invariant: pred != null ==> pred.num < num
19
20     public Block2x(int n, Block b) {
21         super(n < 1 ? 2 : 2*n);
22         pred = (b != null && 2 <= b.num && b.num < num ? b : null);
23     }
24
25 }
```

B.1) Do the invariants in `Block` and `Block2x` satisfy the requirements of *behavioral subtyping*? Assume that we *do not use* specification inheritance. **Briefly explain your answer**.

B.2) A class `C` is *correct* with respect to its invariants if all constructors of `C` establish the invariants *of the new object* and all exported methods `m` of `C` preserve the invariants *of the receiver object*, that is, the invariant holds in the poststate of `m` provided that it held in the prestate of `m`. Are classes `Block` and `Block2x` correct with respect to their invariants? **Briefly explain your answer**.

C) We now want to extend the code in part **B** with *methods that preserve the invariants of the class in which they are declared* but that make it possible to violate the invariants of `Block2x` from client code in another package.

C.1) Extend the code in part **B** with a method that preserves the invariants of the class in which it is declared but makes it possible to violate the *first invariant* of class `Block2x` (i.e., `2 <= num`) from client code in package `B2X`. **Specify in which class you want to declare the method, write the method, and write the client code that violates the invariant.**

C.2) How can you prevent the code that you wrote in part **C.1** from violating the invariant by further extending the code in part **B**? *You are not allowed to modify existing code.* **Write the code that fixes the specific problem you exploited in part C.1.**

C.3) Extend the code in part **B** with a method that preserves the invariants of the class in which it is declared but makes it possible to violate the *second invariant* of class `Block2x` (i.e., `pred != null ==> pred.num < num`) from client code in package `B2X` *in a way that cannot be prevented by further extending the code in part **B***. **Specify in which class you want to declare the method, write the method, and write the client code that violates the invariant.**

Aliasing, Readonly Types, and Ownership Types

Task 13

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.

Task 14

Consider the following class definitions in the context of the read-only type system taught in the course:

```
class C {
   public D f;
   void foo(readonly C other) {...}
}
```

```
class D { E g; }

class E {}
```

Let `a` and `b` be non-null references of type `C`. Which of the following statements is true:

(a) The call `a.foo(b)` is guaranteed not to change the value of `b.f`, but may change the value of `b.f.g`

(b) The call `a.foo(b)` is guaranteed not to change the value of `b.f` and neither the value of `b.f.g`

(c) The assignment `other.f.g = new E();` may appear in the code of `foo`

(d) None of the above is correct

Task 15

Annotate the following program with appropriate ownership type modifiers (according to the topological ownership system) in order to maximize the buffer, the producer, and the consumer *encapsulation*. This means that the modifiers you choose should increase the depth of nested ownership context and reduce the number of (non-rep) edges/pointers between different contexts.

```
class Producer {
 int[] buf;
 int n;
 Consumer con;

 Producer() {
  buf = new int[10];
 }

 void produce(int x) {
  buf[n] = x;
  n = (n+1)
   % buf.length;
 }
}
```

```
class Consumer {
 int[] buf;
 int n;
 Producer pro;

 Consumer(Producer p) {
  buf = p.buf;
  pro = p;
  p.con = this;
 }

 int consume() {
  n = (n+1)
   % buf.length;
  return buf[n];
 }
}
```

```
class Context {
 Producer p;
 Consumer c;

 Context() {
  p = new Producer();
  c = new Consumer(p);
 }

 public void run() {
  for(int i=-5; i<=5;
      ++i) {
   p.produce(i);
   if(i%2 == 0)
    c.consume();
  }
 }
}
```

Task 16

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.

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

Task 17 (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;`

Task 18

Assume the topological ownership framework taught in the course. Suppose that we want to create a class `SortedLinkedList`, with the internal invariant that the values stored in the nodes are sorted in ascending order.

```
package SortedLinkedList;
public class SortedLinkedList {
  private rep Node head;

  /// invariant head != null ==> head.sorted()
  ...
}
private class Node {
  protected peer Node next;
  protected int value;

  /// pure
  boolean sorted() {
    return next != null ==> value < next.value && next.sorted()
  }
}
```

Suppose that all the methods in `SortedLinkedList` are guaranteed to preserve the invariant of the class. Furthermore, suppose that we want to create iterators for such lists (defined in the same package):

```
public class LinkedListIterator { private any Node current_item; ... }
```

A) Why is the field `current_item` annotated as `any`? What drawbacks would the other possible annotations have?

B) We would like to have the following features:

(i) the invariant of a `SortedLinkedList` object is guaranteed to hold in any program, except when one of its methods executes

(ii) `LinkedListIterator` is a *modifying* iterator, i.e., it may change the value of the object it is pointing to

We can't have both features. Depending on whether or not we impose the "owners as modifiers" discipline, we can have either (i) or (ii). Explain why this is the case.

C) The fact that (i) and (ii) cannot both hold together is not surprising. A modifying iterator can break the invariant of the list it is iterating over. However, the "owners as modifiers" discipline may disallow harmless designs. Write a benign class (perhaps a restricted modifying iterator), which would not break the invariant of any object of `SortedLinkedList`, but still does not compile under "owners-as modifiers".

Reflection

Task 19

Which of the following is the *defining* characteristic of reflection?

(a) It allows for much simpler code

(b) It enables more flexibility

(c) It allows a program to observe and modify its own structure and behavior

(d) It is not statically safe

(e) It may hurt performance

(f) None of the above

Task 20

Consider the following Java code:

```
void foo() throws java.lang.Exception {
  LinkedList<String> xs = new LinkedList<String>();
  xs.add("A"); xs.add("B"); xs.add("C");

  Class<?> c = xs.getClass();
  Method remove = c.getMethod("remove");
  xs.add(remove.invoke(xs));
}
```

which uses the following methods of class `LinkedList<E>`

```
  public E remove()
  public boolean add(E e)
```

Which of the following statements is true? The invocation of ...

(a) `c.getMethod("remove")` is rejected by the compiler

(b) `c.getMethod("remove")` raises an exception (at runtime)

(c) `remove.invoke(xs)` is rejected by the compiler

(d) `remove.invoke(xs)` raises an exception (at runtime)

(e) `xs.add(...)` is rejected by the compiler

(f) `xs.add(...)` raises an exception (at runtime)