Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 12

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

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 2 (from a previous exam)

Consider the following Java classes:

```
class A {
   public void foo (Object o) { System.out.println("A"); }
}

class B {
   public void foo (String o) { System.out.println("B"); }
}

class C extends A {
   public void foo (String s) { System.out.println("C"); }
}

class D extends B {
   public void foo (Object o) { System.out.println("D"); }
}

class Main {
   public static void main(String[] args) {
      A a = new C(); a.foo("Java");
      C c = new C(); c.foo("Java");
      B b = new D(); b.foo("Java");
      D d = new D(); d.foo("Java");
   }
}
```

What is the output of the execution of the method `main` in class `Main`?

(a) The code will print A C B D

(b) The code will print A C B B

(c) The code will print C C B B

(d) The code will print C C B D

(e) None of the above

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

Consider the following C++ code:

```
class Person {
   Person *spouse;
   string name;

public:
   Person (string n) { name = n; spouse = nullptr; }

   bool marry (Person *p) {
      if (p == this) return false;
      spouse = p;
      if (p) p->spouse = this;
      return true;
   }

   Person *getSpouse () { return spouse; }
   string getName () { return name; }
};
```

The method `marry` is supposed to ensure that a person cannot marry him/herself. Without changing the code above, create a new object that belongs to a subclass of `Person` and marry it with itself.

Hint: use multiple inheritance. Explain what happens.

Task 6

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++?

Task 7

Java 8 allows interface methods to have a default implementation directly in the interface.

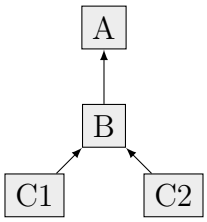
A) What are some advantages of this feature?

- What are some advantages of this feature?
- Given the restrictions above, are there any problems left with such an implementation of multiple inheritance? If so what are they? Propose a solution for each problem you have identified.

Bytecode Verification

Task 8

Consider the following type hierarchy:



Suppose that the method `f` of the 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 9

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 this cannot be done.

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

Parametric Polymorphism

Task 10

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 11 (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 12

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 13 (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 the 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 the 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 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

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".

Non-null Types and Initialization

Task 17

Consider a Java class `Vector`, representing a 2 dimensional vector:

```
public class Vector {
   public Number x; // Remark: Number is a super-interface for
   public Number y; //   Integer, Double, etc.

   public Vector (Number x, Number y) {
      this.x = x;
      this.y = y;
   }
}
```

Suppose that in some other class we write the following method to calculate the length of the vector represented by a `Vector` object:

```
public double vectorLength(Vector c) {
   double x = c.x.doubleValue();
   double y = c.y.doubleValue();
   return Math.sqrt(x * x + y * y);
}
```

A) This implementation is unsafe - when executed it may throw exceptions. Why? Is this a reasonable behavior?

B) Add a pre-condition for the method, specifying what is required to be safe.

C) Suppose you are allowed to modify the signature of the method to include non-null type annotations. To what extent can you weaken the necessary precondition?

D) Suppose that you are also allowed to upgrade the class `Vector` to include reasonable non-null type annotations. How does this affect your previous answer? Do these changes to the class seem reasonable?

Task 18 (from a previous exam)

This question is about extending the non-null type system to handle arrays (ignoring initialization). Array types can have two type modifiers, declaring independently the nullity expectations for the array itself and for the array elements. For any array type `T[]` the corresponding variants are `T?[]?`, `T?[]!`, `T![]?`, `T![]!` (the first modifier applies to the type of objects stored in the array, while the second modifier concerns the reference to the array object itself).

Assuming that we want to guarantee a statically sound approach to subtyping (that is, we want to enforce safety at compile time, without using runtime checks), explain whether or not the following subtype relations are safe. For each relation you consider unsafe, provide a code snippet illustrating that allowing such a subtype relationship would break the safety guarantees of the type system (e.g. call a method on a `null` receiver). For these unsafe cases, explain also what runtime checks could be made to restore safety.

- `T?[]! <: T?[]?`
- `T![]! <: T![]?`
- `T![]? <: T?[]?`
- `T![]! <: T?[]!`

NOTE: Object Initialization was discussed only briefly during the lectures and exercise sessions. The following exercises are left here in case you are interested, but this topic will not be examined.

Task 19

In the construction type system, when we read from the field of an expression of a committed type, we obtain a reference of a committed type, i.e., if e_1 has a committed type then e_1`.f` has a committed type as well. Similarly, if e_1 has an unclassified type then e_1`.f` also has an unclassified type. However, if e_1 has a free type then e_1`.f` does not have a free type, but instead has an unclassified type. Explain why the alternative choice would be unsound (given the existing rules of the system), giving an example of what would go wrong.

Task 20

In the construction type system, a field assignment e_1`.f =` e_2 is permitted if the usual subtyping holds, and if, in addition either e_1 has a free type, or e_2 has a committed type.

In particular (in terms of construction types), it is ok for an expression with committed type to be assigned to the field of an expression with committed type and it is also ok for an expression of free type to be assigned to the field of an expression of free type. However, it is not permitted for an expression of unclassified type to be assigned to the field of an expression of unclassified type. Explain why not, giving an example of what would go wrong if we were to allow this.

Task 21 (from a previous exam)

Consider the following two different implementations of a cyclic list that use the construction type system taught in the course. The type system rejects both of them. The constructors are used to clone an existing list. In both cases we establish a link between a node and its clone.

A) Are there lines of code where we are trying to incorrectly assign to a field of a committed object? If so, in which implementation (*left* or *right*) and on which lines?

B) If we allowed these implementations to run, is it possible that a committed object would become not locally initialized, while a constructor is executing? If so, in which implementation (*left* or *right*) and on which line is there an assignment where this happens?

C) If we allowed these implementations to run, is it possible that a committed object would become not transitively initialized, while a constructor is executing? If so, in which implementation (*left* or *right*) and on which line is there an assignment where this happens?

D) Without changing the constructor signatures in any way, which two lines of the implementation on the *right* can you change and how, so that it typechecks in the construction type system and achieves the expected result? Write the line numbers and the new content of the lines.

```
1 class Node {
2  Node! next; // cyclic
3  Node? copy;
4  int value;
5
6  Node(int x)
7  {
8   next = this;
9   value = x;
10 }
11
12 Node( Node! other )
13 {
14  value = other.value;
15  other.copy = this;
16
17  if(other.next == other)
18   next = this;
19  else
20   next = new Node(other, other.next);
21 }
22
23 Node( Node! first, Node! other )
24 {
25  value = other.value;
26  other.copy = this;
27
28  if(other.next == first)
29   next = other.next.copy;
30  else
31   next = new Node(first, other.next);
32 }
33 }
```

```
1 class Node {
2  Node! next; // cyclic
3  Node? original;
4  int value;
5
6  Node(int x)
7  {
8   next = this;
9   value = x;
10 }
11
12 Node( Node! other )
13 {
14  value = other.value;
15  original = other;
16
17  if(other.next == other)
18   next = this;
19  else
20   new Node(this, this, other.next);
21 }
22
23 Node( free Node! first,
24       free Node! prev, Node! other )
25 {
26  value = other.value;
27  original = other;
28  prev.next = this;
29
30  if(other.next == first.original)
31   next = first;
32  else
33   new Node(first, this, other.next);
34 }
35 }
```

Task 22

Consider the following three classes (declared in the same package):

```
public class Person {
   Dog? dog; // people might have a dog

   public Person() { }
}

public class Dog {
   Person! owner; // Dogs must have an owner
   Bone! bone;    // Dogs must have a bone
   String! breed; // Dogs must have a breed

   public Dog(Person owner, String breed) {
      this.owner = owner;
      this.bone = new Bone(this);
      this.breed = breed;
   }
}
```

```
public class Bone {
   Dog! dog;         // Bones must belong to a dog

   public Bone(Dog toOwn) {
      this.dog = toOwn;
   }
}
```

A) Annotate the code with non-null and construction type annotations where they are necessary. Explain why the code now type-checks according to construction types.

B) Could we provide constructors for the classes `Dog` and `Bone` with no parameters?

C) Now, suppose a (possibly mad) scientist wants to extend the implementations of these classes with some genetic engineering. Firstly, we want to be able to "clone" a bone. We can add the following method to the class `Bone` to make a copy of an existing bone and assign it to another `Dog`:

```
public Bone clone(Dog toOwn) {
   return new Bone(toOwn);
}
```

However, our scientist would like to go further and be able to clone dogs. A cloned `Dog` should also have its bone cloned along with it, but may be assigned to a new owner: we add the following extra constructor and method to the class `Dog`:

```
Dog(Dog toClone, Person newOwner) {
   this.owner = newOwner;
   this.breed = toClone.breed;
   this.bone = new Bone(this);
}

public Dog clone(Person toOwn) {
   return new Dog(this, toOwn);
}
```

However, our scientist would like to still go further and be able to clone people. A cloned `Person` should also have its dog (if any) cloned along with it: we add the following extra constructor and method to the class `Person`:

```
Person(Person toClone) {
   Dog? d = toClone.dog;
   if(d!=null) {
      this.dog = new Dog(d, this);
   }
}

public Person clone() {
   return new Person(this);
}
```

Annotate this extra code with appropriate non-null and construction types annotations. You should guarantee that each of the `clone` methods (which belong to the public interface) return a committed reference. You should ensure that your answers guarantee that all of the code type-checks. Explain your choices.

Hint: think carefully about how constructor calls are typed and what happens if the constructors are called in more than one situation.