

Exercise 13

Self-Study Exercise Sheet

NOTE: This exercise sheet will not be discussed in an exercise session. We publish it now together with the solution to allow you to better prepare for the final exam. If you have any questions regarding this sheet, please consult your assistant.

Subtyping and Behavioral Subtyping

Task 1

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

```
class X {  
    /// requires  $x > 0 \wedge (\neg \exists i, j: \text{int} \mid 2 \leq i, j \leq x \wedge i * j = x)$   
    /// ensures  $\text{result} > 0 \wedge \text{result} \% 2 = 0$   
    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  $x \geq 0$   
    /// ensures  $\text{result} = x + 1$   
    int foo(final int x) {...} }
```

(b)

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

(c)

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

(d) **CORRECT:**

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

Inheritance, Dynamic Method Binding, Multiple Inheritance, and Linearization

Task 2

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.

— solution —

The code will print B1 C1 B2 C1 C2:

a1 is of static type IA and dynamic type B, a2 is of static type IA: a1.g(a2) maps to IA.g(IA), which is overridden in IB as IB.g(IA) and then in B as B.g(IA).

a2 is of static type IA and dynamic type C, b is of static type B: a2.g(b) maps to IA.g(IA), which is overridden in C as C.g(IA).

b is of static type B and dynamic type B: b.g(b) maps to B.g(IB) (more specific than B.g(IA) - overload resolution).

c is of static type C and dynamic type C, a2 is of static type IA: c.g(a2) maps to C.g(IA).

c is of static type C and dynamic type C, b is of static type B: c.g(b) maps to C.g(IB) (more specific - overload resolution).

Bytecode Verification

Task 3

Assume two Java classes A and B and assume that B is a subclass of A. Consider the following byte code:

```
0: aload_1
1: astore_2
2: goto 0
```

and assume that the input to the initial node of this code is $([], [A, A, B])$, where the first list indicates the contents of the stack and the second list indicates the contents of the registers.

After running the bytecode type inference algorithm, what is the inferred input to the initial node?

- (a) **CORRECT:** $([], [A, A, A])$
- (b) $([], [A, A, B])$
- (c) $([], [A, B, B])$
- (d) Nothing is inferred – the type inference does not terminate
- (e) Nothing is inferred – the type inference rejects the program

Parametric Polymorphism

Task 4

This is an 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.

— solution —

```
class Cage {
    ...
    Animal<?> getAnimal() {
        if (animal!=null) return animal;
        else {
```

```

        animal = new Sheep();
        Wolf wolf = new Wolf();
        wolf.lunchBag=wolf;
        return wolf;
    }
}
class Zoo{
    ...
    void manage() {
        feedAnimal(new Cage(null));
    }
}

```

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.

— solution —

```

class Fox extends Animal<Meat>{
    Fox() {}
    void eat(Meat m) {}
    Wolf getLunchBox() { cage.animal=new Sheep();return new Wolf(); }
    Cage cage;
}
class Zoo{
    ...
    void manage() {
        Fox fox = new Fox();
        Cage cage = new Cage(fox);
        fox.cage=cage;
        feedAnimal(cage);
    }
}

```

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.

— solution —

Here we cannot make a sheep eat a wolf.

The reason is that `cage.animal` evaluates to the same value in both expressions `cage.animal` and `cage.animal.getLunchBox()` and so type safety is not broken and the Sheep can only be fed with Grass, which the Wolf is not.

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.

— solution —

This is safe as no methods are called during the evaluation of arguments, so `cage.animal` cannot change.

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 in the sequential case.

— solution —

The version of `feedAnimal` in section D is unsafe as another thread might modify `Cage.animal` between the evaluation of the two expressions.

The version in section C is safe.

F) The current Java rule for evaluating an expression (including a method call) with wildcard typed arguments is to capture each wildcard in the arguments separately. Propose a more lenient wildcard capture rule than current Java, that is typesafe and accept all the above cases that you deem safe.

Hint: define "stable" paths that cannot be modified by calls.

— solution —

We could allow wildcard capture to happen only once per access path in the same statement (rather than once per occurrence in the statement), if either

1. It includes no method calls and there are no method calls evaluated between any two instances of the path (only in the sequential case).
2. It begins with a local variable and follows only final fields (no method calls) - works also in the concurrent case.

Information Hiding and Encapsulation

Task 5

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();}
```

— solution —

The subtyping relations are as follows: $C <: B <: A$

Using structural subtyping we require that the methods and fields of subclasses are more accessible than those of superclasses. When dealing with access modifiers, this means that methods with more permissive modifiers may override methods with less permissive modifiers.

Task 6

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`?

— solution —

The external interface is composed only of the method `public set(int)` since this is the only public element of class `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.

— solution —

The invariant can be broken easily by extending class `Hour`, and accessing the field `h` directly. For instance:

```
public WrongHour extends Hour {  
    public WrongHour() {super.h=-1;}  
}
```

This can be prevented by making the field `h` private.

Aliasing, Readonly Types, and Ownership Types

Task 7

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) **CORRECT:** None of the above is correct

Task 8

Consider this code, which uses the topological ownership type system and the owners-as-modifiers discipline. Assume that there is no concurrency and reflection is not used.

```
public class Cell {
    public int value;
    public Cell(int value) { this.value = value; }
}

public final class Some {
    public final any Cell a;
    public final peer Cell b;
    public final rep Cell c;

    public Some(any Cell a, peer Cell b, int cval) {
        this.a = a;
        this.b = b;
        this.c = new rep Cell(cval);
    }

    public void foo(peer Other other)
    {
        int av = a.value;
        int bv = b.value;
        int cv = c.value;
        other.bar(this)

        assert av == a.value // assertion A
        assert bv == b.value // assertion B
        assert cv == c.value // assertion C
    }
}
```



```

public class Other {
    public void bar(...) { ... }
}

public class Client {
    public void client() { ... }
}

```

You will have to choose an implementation for the methods `Other.bar` and `Client.client` later in this task.

A) After the call `other.bar(this)`, the `Some.foo` method asserts that the values of the three cells have not been changed by the method call. Without knowing anything about the `Other.bar` method (except that it type-checks), say, for each of the assertions, if it is guaranteed to hold or if it might fail. You do not have to justify your answers.

— solution —

- A) `assert av == a.value` - might not hold, can be changed (a may actually be peer)
- B) `assert bv == b.value` - might not hold, can be changed (directly)
- C) `assert cv == c.value` - holds, cannot be changed

Provide an implementation for the methods `Other.bar` and `Client.client`, such that, when the `client` method is executed, all assertions which are not guaranteed to hold will fail.

— solution —

```

class Client {
    void client() {
        peer Cell pc = new peer Cell(5);
        peer Some ps = new peer Some(pc, pc, 6);
        peer Other po = new peer Other();
        ps.foo(po);
    }
}

class Other {
    public void bar(peer Some some) {
        some.b.value = 7;
    }
}

```

Alternatively, all variables in `client` could also be of type `rep`.

B) Does any of your answers in part **A** change if you are also allowed to add arbitrary methods to class `Some`? Briefly explain your answer.

— solution —

Yes, `c.value` can now be changed, because `other` could call a method on `this` that modifies the value.

C) In class `Other`, write a method `cloneSome` which takes a `Some` instance and returns a copy of it. As a first priority, the method must maximize the amount of sharing between the original and the cloned object. As a second priority, its parameter type should allow as many clients to call it as possible. You may add additional constructors (but **not methods**) to class `Some`.

— solution —

```
public class Other {  
    // parameter must be peer, otherwise the method can only be called  
    // from Other or subclasses  
    public peer Some cloneSome(peer Some s) {  
        // to maximize sharing, we share Cells a and b, c cannot be shared  
        return new peer Some(s.a, s.b, s.c.value);  
    }  
}
```

Non-null Types and Initialization

Task 9

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?

— solution —

If `c` were `null`, the field dereferences `c.x` and `c.y` would generate exceptions. Furthermore, if `c.x` were `null` then method call `c.x.doubleValue()` would generate an exception. Similarly, if `c.y` were `null`.

There is no reasonable answer for the method to return if it encounters null values - any attempt to deal with these cases would have to return some arbitrary value, since the question the method is meant to answer is undefined in these cases.

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

— solution —

```
requires: c ≠ null ∧ c.x ≠ null ∧ c.y ≠ null
```

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 pre-condition?

— solution —

```
public double vectorLength(Vector! c)
```

would make the following pre-condition sufficient:

```
requires: c.x ≠ null ∧ c.y ≠ null
```

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?

— solution —

By changing the types of the fields `x` and `y` to be `Number!` we could guarantee that no pre-condition would be required. This seems a reasonable change, since a null `Vector` doesn't seem to be meaningful anyway.

Task 10

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.

— solution —

Here are the annotations for the first version of the code:

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

    public Person() { }
}

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

    public Dog(unc Person ! owner, unc 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(unc Dog ! toOwn) {
        this.dog = toOwn;
    }
}

```

Note that we choose the parameter to the construction of Bone to be unclassified - since it is public then it probably should be callable with a committed parameter from client code, but it is also called inside the body of the constructor of Dog, with a free parameter. Note that the returned reference from these two kinds of call will be different - committed in the former case, and free in the latter. For the Dog constructor, we can also choose to make the parameters unclassified. Although in this case we do not directly need to permit “free” arguments being passed to the constructor, we may as well be as permissive as possible. In general, if it is possible to type a constructor body using “unclassified” argument types then this should be the preferred choice of signature as it is the most permissive. Note that the same does not apply for method signatures, since any overriding method definitions are then also be forced to cope with unclassified arguments, which may be much less convenient than using committed ones.

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

— solution —

It isn’t reasonable to have constructors for Dog and Bone without parameters, since we need some way of initialising their non-null fields. Although it would be possible to do this by calling e.g., the Person constructor from the Dog constructor, this doesn’t seem very intuitive (nor would it be easy to establish the intuitive invariants of the code - that a Dog’s owner refers back to the same Dog, etc.). In particular, if all of the constructors need to take no parameters, they would need to call each other infinitely. This is because, we can’t set up a cyclic object structure without some kind of mutual initialization (in this case we can only build an infinite object structure to satisfy the non-null requirements of all the objects).

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 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 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 go still 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 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.

— solution —

Here is the fully annotated code for the cloning case:

```

public class Person {
    Dog? dog;    // A person might have a dog

    public Person() { }

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

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

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

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

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

```

```

    public Dog! clone(Person! toOwn) {
        return new Dog(this, toOwn);
    }
}
public class Bone {
    Dog! dog;           // A bone must belong to a dog..

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

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

```

Note that all parameters to the new constructors and methods need to have non-null type annotations, since they are each either dereferenced, used to initialize non-null-declared fields or passed on as further parameters to calls that require non-null parameters.

The `toClone` parameter of the new constructor of `Person` needs to be a committed parameter, otherwise when we dereference `toClone.dog` we will obtain an unclassified value, which will not be suitable to use as a parameter for the new `Dog` constructor.

The `toClone` parameter of the new constructor of `Dog` needs to be a committed parameter, since when a field is read from it, we need to obtain a result with a non-null type. However, the `newOwner` parameter of the new constructor of `Dog` needs to be an unclassified parameter. This is because this parameter is sometimes supplied from a free reference (in the new constructor of `Person`), and sometimes from a committed reference (in the clone method of `Dog`).

For similar reasons, the `toOwn` parameter of the constructor of `Bone` needs to be an unclassified parameter (as was suggested for the previous part of the question). This is because this parameter is sometimes supplied from a free reference (in the new constructor of `Dog`), and sometimes from a committed reference (in the clone method of `Bone`).

This is an important usage of the unclassified types in the construction types system - they are useful for constructors which get called sometimes with free and sometimes with committed parameters. Recall that the type of a new expression is determined from the static types of the actual parameters at a particular call, and not from the formal parameters in the constructor signature. For example, in the `clone` method of the `Bone` class, the new expression `new Bone(toOwn)` is given a committed type because the actual parameter `toOwn` has a static type which is committed, despite the fact that the constructor argument type is declared as unclassified in its signature. This means that the same constructor can produce committed/free results depending on the particular arguments provided in each call (new expression). In particular, the return type of the `clone` method can be a committed reference, as required in the question (the same applies to all of the clone methods in the code, since they each call constructors with only committed arguments).

Reflection

Task 11

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

- (a) It allows for much simpler code
- (b) It enables more flexibility
- (c) **CORRECT:** 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 12

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 run time)
- (c) `remove.invoke(xs)` is rejected by the compiler
- (d) `remove.invoke(xs)` raises an exception (at run time)
- (e) **CORRECT:** `xs.add(...)` is rejected by the compiler
- (f) `xs.add(...)` raises an exception (at run time)