Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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 ∧ (¬∃ 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) **CORRECT:**
```
class D {
      /// requires true
      /// ensures result = x*(x+1)
      int foo(final int x) {...}    }
```

solution

Choice `(a)` is not valid since 2 is a valid input to `X::foo()`, but breaks the postcondition if `result = x + 1`.

Choice `(b)` is not valid as it has a weaker postcondition, namely the result is not guaranteed to be larger than 0.

Choice `(c)` is not valid as it does not have a weaker precondition. Note that `X::foo()` accepts 1 and all prime numbers. However, this includes 2, which is even and thus not allowed by the precondition of `C::foo()`.

Choice `(d)` obviously has a weaker precondition. Moreover, on strictly positive inputs, it guarantees strictly positive even outputs. Therefore it has a stronger postcondition.

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 overriden 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 overriden 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).

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

solution

Note that running the bytecode inference algorithm once from instruction 0 to instruction 2 results in retrieving the object in the second register and storing it in the third register. This object is of type `A` thus resulting in `([], [A, A, A])` as input to instruction 0 after the jump of instruction 2. One then needs to compute the smallest common supertype of `A` and `B`, which is `A` since `B` is a subclass of `A`. Therefore the resulting input to the next iteration of the algorithm is `([], [A, A, A])`. This is then propagated to the jump instruction, reaching the fixed point. (The inference algorithm therefore runs twice through instructions 0 and 1, and once through instruction 2, before reaching the fixed point.)

Task 4

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) **CORRECT:** `Number`, `Integer(9)`, `Object`

(e) None of the above

solution

Choices `(a)` and `(b)` are not valid as generic types are invariant in Java. Therefore, assigning a `Box<Integer>` to a `Box<Number>` is illegal.

Choice `(c)` is not valid since `b.get()` would return a `Number`, hence dissallowing the assignment `Integer c = b.get()`.

Choice `(d)` is valid. In the first gap, `Number` is clearly a valid option. In the second, by the substitution principle, we can pass an `Integer` as it is a subtype of `Number`. Finally, the assignment `Object c = b.get()` simply adds an implicit upcast from `Number` to `Object`, which is valid as `Number` is a subtype of `Object`.

Choice `(e)` is not valid as choice `(d)` is valid.

Task 5

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.

Task 6

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 7

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.

Task 8 Information Hiding

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`.

solution

The method `max_size()` in class `Chain` should have a `default` access modifier, so that method `max_size()` in class `Chain2x` does not override it but only hides it. In this way, even if method `max_size()` in class `Chain2x` has a weaker postcondition than method `max_size()` in class `Chain`, we still vacuously have behavioral subtyping.

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**.

solution

Yes, the invariants satisfy the requirements of behavioral subtyping because the invariants in class `Block2x` are stronger than the invariants in class `Block`.

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**.

solution

Yes, classes `Block` and `Block2x` are correct with respect to their invariants because their constructors establish the invariants of the newly created objects (and there are no methods in the two classes).

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.**

solution

It is possible to break the invariant by adding the following method to class `Block`:

```
public void reset() { num = 1; }
```

The client code that breaks the invariant is the following:

```
class Client {
    public static void main(String[] args) {
        Block2x b2x = new Block2x(1, null);
        b2x.reset();
    }
}
```

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.**

solution

It is possible to prevent the above problem by overriding the newly added method `reset` in class `Block2x`:

```
public void reset() { num = 2; }
```

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.**

solution

It is possible to break the invariant by adding the following method to class `Block`:

```
public void incr() { num = num + 1; }
```

The client code that breaks the invariant is the following:

```
class Client {
    public static void main(String[] args) {
        Block b = new Block(2);
        Block2x c = new Block2x(2, b);
        b.incr();
        b.incr();
    }
}
```

Task 9

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

solution

Choice `(a)` and `(b)` are not true since since we can have aliasing (`a` and `b` point to the same object) and `foo()` has no restriction on modifying its receiver, therefore it might modify the value of `b.f` via the alias `a`.

Choice `(c)` is not true since read-only types are transitive, meaning that `other.f.g` is read-only since `other` is read-only. Therefore the assignment is not allowed.

Task 10

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.

solution

The following code breaks the acyclicity requirement for the topology:

```
class C
{
  rep C down;
```

```
  void foo()
  {
    down.down = this;
  }
}
```

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

solution

None. The assignment is always 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)?

solution

We cannot upcast a reference that is being assigned to. This is illegal according to the subtyping rules.

Task 11

Consider the following Java code:

```
public class A {
    public static int a1;
    public static int a2;

    static {
        a1 = D.d * 3;
        new B<String>();
        new B<Integer>();
        a2 = 17 + D.d + a1;
    }
}

public class B<T> {
    public static int b;
    static {
        b = A.a2;
        A.a1 += 10;
    }
}

public class C {
    static { A.a2 += 100; }
}

public class D extends C { public static int d = A.a2; }

public class Main {
    public static void main(String[] args){
        if (1 == 2){
            System.out.println(D.d);
        }else{
            System.out.println(A.a2);
        }
    }
}
```

What is the result of compiling the code and running the `Main.main` method?

(a) The code does not compile

(b) The program does not terminate or aborts with a stack overflow

(c) The program terminates normally and prints `127`

(d) The program terminates normally and prints `417`

(e) The program terminates normally and prints `437`

(f) **CORRECT:** None of the above

solution

The initialisation is performed lazily. When executing the `main()` function, we skip the then branch and execute the `else` branch. There, `A` get initialized when the field read `A.a2`

is encountered. This launches the static initializer for `A`. When executing `a1 = D.d * 3`, the initializer for `D` is called. However, `D` extends `C`, therefore `C`'s initializer is executed first. Since `A`'s initializer has not completed execution and not yet assigned to `A.a2`, the field is still at the default 0 value. Therefore `C`'s initializer sets `A.a2` to be equal to 100. Then `D`'s initializer sets `D.d` to `A.a2` which is still 100. This allows to conclude the `a1 = D.d * 3` assignment, setting `A.a1` to 300. Then, as a `B` is constructed, it sets `B.b` to 100, and increments `A.a1` to 310. The second constructor does not affect the lazy initialization, since it is only computed once per class, and type erasure tell's us `B` is a single class.

At the current standing, we have that `A.a1` is 310, `A.a2` is 100, `B.b` is 100, `D.d` is 100. This makes the final assigment to `A.a2` trivial. Therefore the output of this program is 427.

Task 12

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 13

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).

Task 14

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

solution

See page 4 of the slides on reflection.

Task 15

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)

solution

This code snippet aims to create a `LinkedList` of `String`, add three elements to it, recover class and method information via reflection, remove an element from the list and add it to the list again.

The issue with this code is that the return type of `Method.invoke(...)` is `Object`. Therefore, the compiler complains that there is no suitable method `xs.add()` that takes an `Object` parameter returned from `remove.invoke(xs)`.