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

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

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) **CORRECT:** 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.

solution

The code will print `B1 C1 B2 C1 C2`:

`a1` has static type `IA`, `a2` has static type `IA`: the statically selected method for the first call is `IA.g(IA)`. This method is overriden in `B` (the dynamic type of `a1`) by `B.g(IA)`.

`a2` has static type `IA`, `b` has static type `B`: the statically selected method for the second call is `IA.g(IA)`. This method is overriden in `C` (the dynamic type of `a2`) by `C.g(IA)`.

`b` has static and dynamic type `B`: the statically selected method for the third call is `B.g(IB)` (the most specific method according to the overloading resolution). This method will be executed at runtime.

`c` has static and dynamic type `C`, `a2` has static type `IA`: the statically selected method for the fourth call is `C.g(IA)`. This method will be executed at runtime.

`c` has static and dynamic type `C`, `b` has static type `B`: the statically selected method for the last call is `C.g(IB)` (the most specific method according to the overloading resolution). This method will be executed at runtime.

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.

solution

The following C++ code breaks the invariant:

```
class Jeweler : virtual public Person {
   public: Jeweler () : Person (true) {}
           // diamonds are a jeweler's best friend
};

class JewelerProgrammer : public Jeweler, public Programmer {
   public: JewelerProgrammer () :
           Person (true), Jeweler (), Programmer () {}
};

void break () {
   JewelerProgrammer* programmer = new JewelerProgrammer();
}
```

The call of the constructor `Person(true)` in class `JewelerProgrammer` bypasses the corresponding call `Person(false)` in class `Programmer`, breaking the invariant.

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.

solution

The following C++ code breaks the invariant:

```
class B : public Person
{ public: B (string n) : Person (n) {} };
class C : public Person
{ public: C (string n) : Person (n) {} };
class D : public B, public C
{ public: D (string n) : B(n), C(n) {} };

void marryMyself () {
   D me = D("Me");
   B *b = &me;
   C *c = &me;
   b->marry(c);
   if (b->getSpouse()) cout << b->getSpouse()->getName();
}
```

The object `me` contains an object of class `B` and an object of class `C`. The addresses of these objects are different and they are obtained using the assignments to `b` and `c`, respectively. During the call `b->marry(c)`, the condition `p == this` compares these two addresses and finds them not equal.

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++?

solution

Here are the three requested classes:

```
class Queue {
   int[] contents;
   int size;

   public: Queue() { contents = new int[100]; size = 0; }
           void enqueue(int x) {...}
           int dequeue() {...}
           int getSize() { return size; }
};

class SumQueue : virtual public Queue {
   int sum;
```

```
   public: SumQueue() : Queue() { sum = 0; }
           void enqueue(int x) {
             sum += x;
             Queue::enqueue(x);
           }

           int dequeue() {
             int r = Queue::dequeue();
             sum -= r;
             return r;
           }

           int getSum() { return sum; }
};

class ProductQueue : virtual public Queue { ... };

class SuperQueue : public ProductQueue, SumQueue {
   public: SuperQueue() : Queue(), ProductQueue(), SumQueue() {}
           void enqueue(int x) {
             ProductQueue::enqueue(x);
             SumQueue::enqueue(x);
           }

           int dequeue() {
             int r = ProductQueue::dequeue();
             SumQueue::dequeue();
             return r;
           }
};
```

One problem is that the `enqueue` and `dequeue` methods of the superclass are called twice. An item is enqueued and dequeued twice. Interestingly, this behaves exactly like a queue, but the capacity is half of the capacity of the original and the `getSize` method reports the correct size multiplied by 2.

We can use traits and linearization to ensure that the `enqueue`/`dequeue` methods are called only once. Here is the relevant Scala code:

```
class Queue {
   ...
   def enqueue(x: int) = {...}
   def dequeue(): int = {...}
}

trait Sum extends Queue {
   var sum: int = 0
   override def enqueue(x:int) = { sum += x; super.enqueue(x); }
   override def dequeue(): int = {
        var x = super.dequeue();
        sum = sum - x;
        return x;
   }
}

trait Prod extends Queue {
   var count: int = 1
   override def enqueue(x: int) = { prod *= x; super.enqueue(x); }
   override def dequeue(): int = {
        var x = super.dequeue();
        prod = prod / x;  // this assumes no zeros in the queue
```

```
            return x;
        }
    }
```

Now, an object of type `Queue with Sum with Prod` has both functionalities, but calls each underlying `enqueue`/`dequeue` method only once. The problems of the multiple inheritance solution do not appear here.

Task 7

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

A) What are some advantages of this feature?

solution

An advantage is that default implementations can be reused in multiple classes. Another advantage (and the main reason this feature was added to Java) is that default method implementations allow interface evolution. Without a default implementation, adding new methods to an interface would break all existing classes that implement that interface, since they do not contain an implementation for the new methods. The new features removes this problem.

B) What could be some problems with this feature? How can they be resolved?

solution

A problem could be inheriting two default implementations of the same method from unrelated interfaces. In that case we will have to either choose which implementation we prefer or write a new implementation that overrides both.

Another issue is that interfaces can now suffer from the fragile base class problem. Compared to the usual issue with normal Java classes, this is even more dangerous for interfaces with default methods, since these methods will mostly call other methods of the interface which are overriden in implementing classes. A very restrictive solution here could be to prohibit calls to other methods of the interface, within the implementation of default methods. Alternatively we can "deal" with the problem just like Java deals with the issue in classes - do nothing and rely on the programmer to be careful.

C) What problems of C++ multiple inheritance are avoided by this new design for Java interfaces?

solution

We still avoid problems with correct initialization of fields of super types, since only one super type (the extended class) can have fields, and we can directly call its constructor. Furthermore there are no problems with field duplication as in non-virtual C++ inheritance.

D) Now suppose that, in addition to method implementations, Java also allowed interfaces to define fields. Interfaces would not have constructors and interface fields would always be initialized with a default value.

solution

This makes multiple inheritance in Java very similar to C++.

- What are some advantages of this feature?

solution

An advantage is that we can also reuse fields. This will enable more methods with default implementations in interfaces which could increase code reuse and reduce the effort required to create new classes.

- 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.

solution

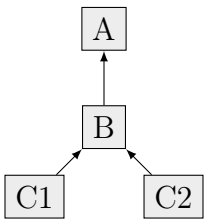
These restrictions are somewhat similar to Scala traits, which also do not have specialized constructors (only a default constructor). In this way we manage to avoid problems with initialization order. However a problem that still remains is: how many copies of a field exist? In particular:

- A class might implement the same interface multiple times (for example by implementing two different interfaces that are a subtype of the same interface). A solution here might be to only have a single copy of the field (as in C++ virtual inheritance).
- A class might implement two different interfaces that both declare the same field. Here we could either restrict interfaces to defining only private fields (which are invisible to the implementor), or we could require some disambiguation syntax when accessing fields, similar to C++ or the proposed syntax for disambiguating conflicting default methods in Java 8.

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.

solution

Here the initial state is `([], [E,b,b,C1,C2,A])`. We denote the type `boolean` as `b` for convenience (in reality the Java bytecode verifier views it as an integer). We show the solution following the convention from the lecture. To each command we dedicate an input and an output column. A command may have multiple inputs and outputs, corresponding to the different iterations of the algorithm. You may also want to see an animated solution of this task, published separately.

		IN	OUT
0	`iload 1`	([], [E,b,b,C1,C2,A]) ([], [E,b,b,B,A,A]) ([], [E,b,b,A,A,A])	([b], [E,b,b,C1,C2,A]) ([b], [E,b,b,B,A,A]) ([b], [E,b,b,A,A,A])
1	`ifeq 22`	([b], [E,b,b,C1,C2,A]) ([b], [E,b,b,B,A,A]) ([b], [E,b,b,A,A,A])	([], [E,b,b,C1,C2,A]) ([], [E,b,b,B,A,A]) ([], [E,b,b,A,A,A])
4	`iload 2`	([], [E,b,b,C1,C2,A]) ([], [E,b,b,B,A,A]) ([], [E,b,b,A,A,A])	([b], [E,b,b,C1,C2,A]) ([b], [E,b,b,B,A,A]) ([b], [E,b,b,A,A,A])
5	`ifeq 12`	([b], [E,b,b,C1,C2,A]) ([b], [E,b,b,B,A,A]) ([b], [E,b,b,A,A,A])	([], [E,b,b,C1,C2,A]) ([], [E,b,b,B,A,A]) ([], [E,b,b,A,A,A])
8	`aload 3`	([], [E,b,b,C1,C2,A]) ([], [E,b,b,B,A,A]) ([], [E,b,b,A,A,A])	([C1], [E,b,b,C1,C2,A]) ([B], [E,b,b,B,A,A]) ([A], [E,b,b,A,A,A])
9	`goto 14`	([C1], [E,b,b,C1,C2,A]) ([B], [E,b,b,B,A,A]) ([A], [E,b,b,A,A,A])	([C1], [E,b,b,C1,C2,A]) ([B], [E,b,b,B,A,A]) ([A], [E,b,b,A,A,A])
12	`aload 4`	([], [E,b,b,C1,C2,A]) ([], [E,b,b,B,A,A]) ([], [E,b,b,A,A,A])	([C2], [E,b,b,C1,C2,A]) ([A], [E,b,b,B,A,A]) ([A], [E,b,b,A,A,A])
14	`astore 3`	([C1], [E,b,b,C1,C2,A]) ([B], [E,b,b,C1,C2,A]) ([B], [E,b,b,B,A,A]) ([A], [E,b,b,B,A,A]) ([A], [E,b,b,A,A,A])	- ([], [E,b,b,B,C2,A]) - ([], [E,b,b,A,A,A]) ([], [E,b,b,A,A,A])
15	`aload 5`	([], [E,b,b,B,C2,A]) ([], [E,b,b,A,A,A])	([A], [E,b,b,B,C2,A]) ([A], [E,b,b,A,A,A])
17	`astore 4`	([A], [E,b,b,B,C2,A]) ([A], [E,b,b,A,A,A])	([], [E,b,b,B,A,A]) ([], [E,b,b,A,A,A])
19	`goto 0`	([], [E,b,b,B,A,A]) ([], [E,b,b,A,A,A])	([], [E,b,b,B,A,A]) ([], [E,b,b,A,A,A])
22	`aload 3`	([], [E,b,b,C1,C2,A]) ([], [E,b,b,B,A,A]) ([], [E,b,b,A,A,A])	- - ([A], [E,b,b,A,A,A])
23	`areturn`	- - ([A], [E,b,b,A,A,A])	- - ([], [E,b,b,A,A,A])

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.

solution

In the following code, we show the types that are given by the user and those inferred by the type checker.

```
  // given: ([],[E,b,b,A,A,A])
0: iload 1
  // ([b], E,b,b,A,A,A])
1: ifeq 22
```

```
  // ([], [E,b,b,A,A,A])
4: iload 2
  // [b], [E,b,b,A,A,A]
5: ifeq 12
  // ([], [E,b,b,A,A,A])
8: aload 3
  // ([A], [E,b,b,A,A,A])
9: goto 14


  // ([], [E,b,b,A,A,A])
12: aload 4
  // given: ([A], [E,b,b,A,A,A])
14: astore 3
  // ([], [E,b,b,A,A,A])
15: aload 5
  // ([A], [E,b,b,A,A,A])
17: astore 4
  // ([], [E,b,b,A,A,A])
19: goto 0
  // ([], [E,b,b,A,A,A])
22: aload 3
  // ([A], [E,b,b,A,A,A])
23: areturn
  // ([], [E,b,b,A,A,A])
```

The requirement to have type information at all basic blocks is a simplification that makes it easier to determine where the compiler should output the information. Note that some basic blocks have only a single preceding instruction, but determining this statically could be hard. Such basic blocks, in theory, do not need type information. Only basic blocks that are also join points definitely need type information. In our example, the instructions 4, 8, 12, and 22 are indeed the beginnings of basic blocks, but there is exactly one path to enter these blocks and therefore type information is not really needed since this information will be identical to the *out*-state of the single preceding instruction.

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.

solution

```
0: aload 0
1: iconst 1
2: ifne 4
3: aload 0
4: astore 1
```

Note: `ifne` jumps to the given index if the integer value at the top of the stack is not equal to zero. It pops the value at the top of the stack.

There are two possibilities for the stack size after executing this program. In any of the two cases, the height of the stack at point 4 is at least 1, and there will be surely a reference value at the top of the stack.

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.

solution

We distinguish between two different cases:

1. If the stack sizes are statically known we can construct such an algorithm. The update is as follows: when joining stacks of different sizes, pick the smallest one, but carry as extra information the size of the largest one to be used when checking for overflow. Note that if we just picked the smaller one and used that, we would not prevent stack overflows at runtime. If we just picked the largest one and made the "extra" values into dummy values by giving them the "top" type, we might not prevent underflows when using instructions such as `pop()`.
2. In general it is not possible to implement an algorithm that can deal with stack sizes which could vary at runtime. For example, if we push elements on top of the stack in a loop, then the verifier will have no way of deciding what an upper bound for the size is. Conversely for loops which pop elements from the stack, the verifier will not be able to deduce a lower bound for the stack size. These situations can easily result in over/underflows and should be rejected.

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

solution

This limitation is not essential. If there are two states `{[head1, x], [head2]}` where `head1` and `head2` are stacks of the same size, then any following code cannot access `x` and it would have been possible to remove `x` already during bytecode generation. This is indeed what the Java compiler does. Consider the following Java code:

```
public int bar() { return 42; }

public int foo(int x) {
  if (x == 0) { bar(); }
  return x;
}
```

If `bar` is called then it will put 42 on the stack, but this value is not actually needed for the final `return` instruction. The Java compiler would emit as many `pop` instructions as necessary to remove unneeded stack elements and make sure that all the paths that reach `return` have the same stack length. Here is the bytecode that corresponds to the `foo` method:

```
 0: iload 1
 1: ifne 9
 4: aload 0
 5: invokevirtual bar // Call to bar(), puts an int on the stack
 8: pop               // Pop the stack to remove the unnecssary int
 9: iload 1           // Here we get equal stack sizes from both paths
10: ireturn
```

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) **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 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.

solution

Note that in order to have a `Sheep` eat a `Wolf`, `Cage.getAnimal().lunchbag` needs to return a `Wolf`, and the second call to `Cage.getAnimal()` to return a `Sheep`. This is not possible with the current implementation of `Cage.getAnimal()`. Therefore the solution is to change the implementation of said function to return different objects for different calls. This can be done in several ways.

The following code uses an implicit flag (`null` field), but using an explicit flag is also possible. Another possible solution is to count the number of times the function was called and alternate the objects that are returned.

```
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

The solution here is to realize that the only code that can be modified that will run between the two accesses of `cage.animal` is in the call to `cage.animal.getLunchbag()`. Somehow the code will need to change the animal contained in the cage so that the access of `cage.animal` returns a `Sheep`. Clearly the animal cannot be a `Sheep` all along, as this would disallow returning a `Wolf` from `getLunchbag()`. The idea is to have an animal capable of eating a `Wolf` (a subtype of `Animal<Meat>` that contains a reference to its cage, in order to change the contents of its own cage to a `Sheep` during the call to `getLunchbag()`:

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

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 the 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 happen 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.

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`?

solution

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

solution

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

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();
    }
}
```

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

solution

Choices `(a)` and `(b)` are not true 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 readonly types are transitive, meaning that `other.f.g` is readonly since `other` is readonly. Therefore the assignment is not allowed.

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();
  }
 }
}
```

solution

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

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

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

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

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

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

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

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

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

We do not have to add ownership modifiers to primitive types. We could have annotated `con` in `Producer` and `pro` in `Consumer` as `any` — in general, this would even allow one modification less (in the topological system): of an `any` receiver, only an `any` field can be modified, whereas of a `peer` receiver, both a `peer` and an `any` field can be modified. However, our goal is to maximize encapsulation and therefore `peer` is the best choice here.

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?

solution

If `current_item` were annotated as `rep`, then the owner of the node it refers to is the iterator itself. In this case, the iterator cannot iterate over a `SortedLinkedList` object l, because l also owns its nodes. The ownership topology allows at most one owner per object.

If `current_item` were annotated as `peer`, then, assuming that `current_item` has a list owner l, the owner of the iterator must also be l. This may be OK in topological ownership. However, if we add "owners as modifiers", the iterator's methods that traverse l cannot be called directly from an object outside l, which defeats the purpose of iterators.

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.

solution

If we don't have "owners as modifiers", an object may get/hold an `any` reference to a node of the list, modify its `value` field, and break the invariant: (i) is not achieved.

If we do have "owners as modifiers", then the iterator may not modify the value of the node it is pointing at, because it holds an `any` reference to it: (ii) is not achieved.

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".

solution

We could have an iterator that performs the requested modification iff this does not violate the invariant:

```
public class LinkedListIterator {
  private any Node f;

  ... // some non-modifying methods

  public void modifyCarefully(int x) {
    if (f.value <= x && (f.next == null || x < f.next.value))
      f.value = x;
    // benign but does not type check 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?

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` $\neq$ **null** $\wedge$ `c.x` $\neq$ **null** $\wedge$ `c.y` $\neq$ **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 precondition?

solution

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

would make the following precondition sufficient:

requires: `c.x` $\neq$ **null** $\wedge$ `c.y` $\neq$ **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 precondition would be required. This seems a reasonable change, since a `null Vector` doesn't seem to be meaningful anyway.

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?[]!`

solution

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

```
Object![]? x = new Object![1]? {
  new Object() // initialization of the first element in the array
};
Object?[]? y = x;
if (y != null) y[0] = null;
if (x != null) x[0].toString();
```

Note: in the last two lines we check the non-nullness of `x` and `y` because we assume that the dataflow analysis does not infer that `x` is non-null after the first statement.

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

```
Object![]! x = new Object![1]! {
  new Object() // initialization of the first element in the array
};
Object?[]! y = x;
y[0] = null;
x[0].toString();
```

In both the last two cases, we need to check at runtime if a value stored in an array with dynamic non-null type for the elements stored in the array is not the `null` value. Alternatively, we can check at runtime if a value read from an array with dynamic non-null type is not the `null` value.

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.

solution

Because anything (in terms of construction type annotations) can be stored in the fields of a free reference, when we read something back from such a field we do not have any guarantees. In particular, it is possible to store a committed reference in the field of a free reference and if we could then read it back as free, this would be unsound. For example, the following code would type-check:

```
public class C {
   C! f, g;

   public C(C! x) {                  // x is committed, this is free
      this.f = x;                    // assigning committed to free - ok
      free C? y = this.f;            // this.f is free - ok
      if (y != null) {
          y.f = this;                // y.f is free, this is free - ok
          this.g = x.f.g;            // what happens here?
      }
   }

   public C() { f = this; g = this; }
}

void client() {
   C! c = new C(new C());
   c = c.g.g; // NullPointerException
}
```

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.

solution

Because unclassified references are supertypes of the corresponding free and committed references, then if we were to allow this, we might "disguise" the assignment of a free reference to the fields of a committed reference. For example, the following code would then type-check, which is unsound:

```
public class C {
   C! f, g;
   public C(C! x) {     // x is committed, this is free
      unc C! y = x;     // cast committed to unclassified - ok
```

```
        unc C! z = this; // cast free to unclassified - ok
        y.f = z;         // assign unc to field of unc (?)
        this.g = x.f.g;  // what happens here?
        this.f = 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?

solution

left: 15, 26

Note that furthermore line 29 does not type-check according to the non-null type system (not required as an answer to this question).

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?

solution

It is not possible for a committed object to become not locally initialized.

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?

solution

left: 15, 26

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.

solution

```
20: next = new Node(this, this, other.next);
33: next = new Node(first, this, other.next);
```

```
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 (from a previous exam)

Consider the following code in a Java-like language enriched with the non-null type system of the course:

```
class Node {
   int depth;
   public Node! parent;
   public Node! left;
   public Node! right;

   Node(int d) { ... }
   ...
}
```

The constructor shown above, when invoked with a positive integer, as in

```
new Node(d)
```

must create a complete binary tree (type `Node!`) of depth `d` containing exactly $2^{d+1} - 1$ nodes. The root node has depth 0. The `depth` field of every node in the constructed tree must be initialized to the depth of that node in the tree. The `parent` field of the root node should point to the root node itself. Similarly the `left` and `right` fields of leaf nodes should point to the leaf nodes themselves.

A) Write the body of the constructor. You may introduce other constructors and methods. Make sure that you adhere to the rules of the non-null type system including construction types.

solution

Here is a possible implementation:

```
Node(int d) {
   depth = 0;
   parent = this;
   if(d == 0) {
      left = this;
      right = this;
   } else {
      left = new Node(d, 1, this);
      right = new Node(d, 1, this);
   }
}

Node(int goal, int d, free Node! p) { // can be unc Node!
   depth = d;
   parent = p;
   if(d == goal) {
      left = this;
      right = this;
   } else {
      left = new Node(goal, d + 1, this);
      right = new Node(goal, d + 1, this);
   }
}
```

The body of the first constructor could be replaced by a call to `this(d,0,this)`, assuming the definite assignment rule correctly determines that calling another constructor guarantees that all fields are assigned (in that other constructor). This question is, however, not addressed in the lecture, so the above solution is the "safe" solution.

B) Consider the following method:

```
void foo(unc Node! o) {
   unc Node! x = new Node(2);
   free Node! y = new Node(2);
   Node! z = new Node(2);
   o.right = new Node(2);
}
```

Which of these assignments would typecheck? Explain your answer.

solution

The type of `new Node(2)` is committed. This can be shown trivially, because no references are passed to the constructor.

From the assignments to local variables, the second one is not allowed because it violates the subtyping rules. The other two are allowed.

The fourth assignment is allowed. By the rules for assignments to fields, we know that a committed reference can be assigned to non-null fields of unclassified, free, and committed objects.

Task 23

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 constructor 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 the 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 initializing 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 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.

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).