Exercise 12

Construction Types and Initialisation

16th December 2011

1. With non-null types, any class type T can be annotated to explicitly declare non-nullity (`T!`) and possible-nullity (`T?`). In the Construction Types system, further variants of these types are introduced, for "`free`", "committed" (the default), and "unclassified" (`unc`) types. These types are all treated differently by the type system taught in the lectures.

   - Explain at least one difference between the treatments of a reference of type `T!` and a reference of type `T?` , giving an illustrative example.
   - Explain at least one difference between the treatments of a reference of type `free T!` and a reference of type `unc T!` , giving an illustrative example.
   - Explain at least two differences between the treatments of a reference of type `T!` (a committed reference) and a reference of type `unc T!` , giving illustrative examples.
   - Explain at least three differences between the treatments of a reference of type `T!` and a reference of type `free T!` , giving illustrative examples.

2. In the Construction Types 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.

3. In the Construction Types system, when we read from the field of an expression of committed type, we obtain a reference of committed type, i.e., if e_1 has a committed type then e_1`.f` has a committed type. Similarly, if e_1 has an unclassified type then e_1`.f` 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.

4. 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;
  }
}
```

• 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.
• Could we provide constructors for classes Dog and Bone with *no* parameters?

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.

5. (question from a previous exam)

Consider the following Java class, partially annotated with non-null and Construction Types. The spaces marked with ■■■ are places where extra non-null and Construction annotations might be required.

The class represents a binary tree, in which every node stores a numerical value, and has references to its `parent` (if any), `left` and `right` children (if any), and the `root` node of the tree to which it belongs. The root node of a tree also refers to itself via its own `root` field.

The class provides three constructors. `Constructor 1` is public, and creates a single, disconnected Node storing the specified value (i.e., it is a "tree" with just one node). `Constructor 3` is also public, and is used for cloning an existing tree structure. Its implementation simply passes on the constructor call to `Constructor 2` with suitably-specified extra arguments. `Constructor 2` is not public, and is used to implement the copying of trees. It creates a deep copy of the tree structure rooted at `toCopy`, but with a

newly-specified `parent` and `root` for the new copy of the tree. This allows copies of one tree to be added as children to another existing tree, as in the `copyToLeft` method.

```
public class Node {
  Node? parent;
  Node? left, right;
  Number! value;
  Node! root;

  // Constructor 1
  public Node(Number! value) {
    this.value = value;
    this.root = this;
  }

  // Constructor 2
  Node(■■■■Node■■ toCopy, ■■■■Node■■ parent, ■■■■Node■■ root) {
    Node? l = toCopy.left;
    Node? r = toCopy.right;
    if(l != null) {
      ■■■■Node■■ leftCopy = new Node(l, this, root);
      this.left = leftCopy;
    }
    if(r != null) { ... // analogous to left code..
    }
    this.parent = parent;
    this.root = root;
    this.value = toCopy.value;
  }

  // Constructor 3
  public Node(Node! toCopy) {
    this(toCopy, null, this); // invoke Constructor 2
  }

  public void copyToLeft(Node! toCopy) {
    ■■■■Node■■ copy = new Node(toCopy, this, this.root);
    this.left = copy;
  }
}
```

In all of the following questions, when we refer to a "type", we mean a static type, including any appropriate non-null and Construction Types annotations. Recall that if no Construction Type is annotated, the default meaning is "committed".

In the Construction Types system, a `new` expression is always given a non-null type. In addition, the expression is typed as "committed" if all of the arguments to the `new` expression have "committed" types, and the expression is typed as "free" otherwise.

Consider the method `copyToLeft()`, whose code is repeated here:

```
public void copyToLeft(Node! toCopy) {
  ■■■■Node■■ copy = new Node(toCopy, this, this.root);
  this.left = copy;
}
```

- What is the type of the expression `this.root` inside the body of the method `copyToLeft()`? Briefly explain your answer.

- Choose appropriate non-null and Construction Type annotations for the local variable `copy`, by adding annotations if necessary in the spaces marked below:

```
_____Node___ copy = new Node(toCopy, this, this.root);
```

- Given your choice of type annotations, justify that this assignment statement is permitted by the rules of the Construction Types system.

- Given your choice of type annotations, justify that the subsequent field assignment

  ```
  this.left = copy;
  ```

  is also permitted by the rules of the Construction Types system.

Now consider the second constructor (labelled `Constructor 2`). `Constructor 2` is called in three different places in the provided code: in its own body, in the body of `Constructor 3` (using the explicit constructor call `this(...)`), and in the body of the `addToLeft` method.

- Considering these three calls of the constructor, and the actual code inside the body of the constructor (copied below), choose appropriate non-null and Construction Types annotations for the parameters of the constructor, by adding annotations if necessary in the spaces marked below. Justify why each of your choices is necessary.

```
// Constructor 2

Node(____Node__ toCopy, ____Node__ parent, ____Node__
root) {
    Node? l = toCopy.left;
    Node? r = toCopy.right;
    if(l != null) {
      ■■■■Node■■ leftCopy = new Node(l, this, root);
      this.left = leftCopy;
    }
    if(r != null) { ... // similar
    }
    this.parent = parent;
    this.root = root;
    this.value = toCopy.value;
}
```

- Choose appropriate non-null and Construction Type annotations for the local variable `leftCopy`, by adding annotations if necessary in the spaces marked below. Briefly explain your choices.

```
_____Node___ leftCopy = new Node(toCopy.left, this, root);
```

- Justify clearly that each of the following three field assignments from the end of the body of `Constructor 2` are permitted by the Construction Types system, given your previous answers.

```
this.parent = parent;

this.root = root;

this.value = toCopy.value;
```

6. The Java approach to static (class) initialisation is to permit `static` blocks, defining code to be executed when the class is initialised. A class begins its initialisation immediately after it is loaded, which can be triggered by various criteria (see slide 62 of lecture 8.3). The C# approach is similar. Because the `static` block can contain unrestricted Java code, it is possible that executing a `static` block triggers the loading

of other classes. In this case, execution of the current `static` block will be postponed, and the `static` block for the new class executed first. The exception to this rule is that if initialisation for the new class has already been started, the "trigger" is ignored (to avoid cycles), and the previous class continues with its initialisation.

Bearing in mind this semantics, consider the following questions:

- One criterion to trigger the loading of a class is an access to a static field or method of the class. Given this criterion, is it safe for the code in the body of a static method to assume that that code of the class' `static` block has already been executed (i.e., class initialisation has already taken place)?
- Another criterion to trigger class initialisation is an attempt to create a new instance of a class. Given this criterion, is it safe for the code in the body of a constructor to assume that the class initialisation has already taken place? What about code in the body of instance methods of the class?
- A further triggering criterion is that initialisation of a superclass will be triggered by an attempt to initialise a subclass. Given this criterion, is it safe for code in the `static` block of the subclass to assume that the superclass initialisation has already taken place? What about code in the bodies of instance methods in the subclass?

7. Consider the following Java classes:

```
public class A {
  public static final int value = B.value + 1;
}

public class B {
  public static final int value = C.value + 1;
}

public class C {
  public static final int value = A.value + 1;
}
```

Will these classes compile? If not, how could we modify them so that they do?

What would the output of running the following program be?

```
public class Program {

  public static void main(String[] args) {
    System.out.println(A.value);
    System.out.println(B.value);
    System.out.println(C.value);
  }
}
```

In what ways can you change the output of the program by reordering the statements?