Exercise 12 Solutions

16th December 2011

1\.
For all examples below, let us suppose that class T has the following field declarations:

```
T! f;
T? g;
```

- If `x` is a reference of type `T!` then `x.f` is a permitted field read (without any if-checks/dataflow analysis), but if `x` is a reference of type `T?` then it is not.
  Also, `x` can only be assigned to the `f` field of an object in the former case and not the latter (`T!` is a subtype of `T?` but not vice versa).
- Suppose `y` is a reference of type `free T!.` If `x` is also a reference of type `free T!` then `x.f = y;` is a permitted field update, but if `x` is a reference of type `unc T!` then it is not.
  Also, `free T!` is a subtype of `unc T!` but not vice versa.
- If `x` is a reference of type `T!` then `x.f.f` is a permitted field read, since `x.f` also has the type `T!`. But if `x` is a reference of type `unc T!` then it is not permitted, since `x.f` has the type `unc T?`.
  If `y` is a further reference of type `unc T!` then `y.f = x` is allowed when `x` has the type `T!` but not when `x` has the type `unc T!`.
  Also, `T!` is a subtype of `unc T!` but not vice versa.
  Furthermore, a constructor call `new C(x)` will be given a committed type if `x` is committed, but instead a free type if `x` is unclassified.
- If `x` is a reference of type `T!` then `x.f.f` is a permitted field read, since `x.f` also has the type `T!`. But if `x` is a reference of type `free T!` then it is not permitted, since `x.f` has the type `unc T?`.
  If `y` is a further reference of type `unc T!` then `y.f = x` is allowed when `x` has the type `T!` but not when `x` has the type `free T!`.
  Similarly, `x.f = y` is allowed when `x` has the type `free T!` but not when `x` has the type `T!`.
  Furthermore, a constructor call `new C(x)` will be given a committed type if `x` is committed, but instead a free type if `x` free.

2. 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 not sound:

```
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;
  }
}
```

3. Because anything (in terms of Construction Type annotation) can be stored in the fields of a free reference, when we read something back out of such a field we cannot make any guarantees about what is stored there. 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 free to committed - ok
    this.f.f = this; // this.f free(?), so this would be ok
    this.g = x.f.g;  // what happens here?
  }
}
```

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

```
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(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.

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 initialisation (in this case we can only build an infinite object structure to satisfy the non-null requirements of all the objects).

Here is the fully annotated code for the cloning case:

```
public class Person {
  Dog? dog;   // people 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; // Dogs must have an owner
  Bone! bone;      // Dogs must have a bone
  String! breed; // Dogs 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;        // Bones 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 initialise 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 a 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).

5.

- The expression is typed as `Node !`. We are reading from the field of a (non-null) committed reference and the declared type of the field has a non-null annotation.
- `Node ?/! copy = new Node(toCopy, this, this.root);`
  (either non-null or possibly-null are ok here).
- The `new` expression gets a committed type since every argument has a committed type.

- For the field update, in terms of Construction types, a committed reference can always be assigned to a field.

- `// Constructor 2`

```
Node(Node! toCopy, unc Node? parent, unc Node! root)
```

Making `toCopy` committed is necessary because we need `toCopy.value` to have a non-null type.
Making `toCopy` non-null is necessary because we dereference `toCopy` in the body of the constructor.
Making `parent` unclassified is necessary because we call the constructor both with free and committed second arguments.
Making `parent` possibly-null is necessary because we call the constructor with a null argument in one case.
Making `root` unclassified is necessary because we call the constructor with both free and committed third arguments.
Making `root` non-null is necessary because we assign root to a non-null declared field (root) of this.

- **free/unc** `Node` **?/!** `leftCopy = new Node(l, this, root);`
  The type of the new expression is free Node!, because not all of its arguments are committed (this is free). We can choose either free or unclassified, and either possibly-null or non-null here.

- `this.parent = parent;` is ok because “this” is free (and “left” is declared possibly null) and we can assign anything to the fields of free references

  `this.root = root;` is ok because “this” is free and “root” is declared non-null while the argument “root” is also non-null

  `this.value = toCopy.value;` is ok because “this” is free. Also, toCopy is committed non-null , and value is declared non-null and so toCopy.value is non-null, and so can be assigned to a non-null field.

6.

- No – here is an example (consider calling `B.bar()` when `A` hasn’t been loaded):

```
public class A {              public class B {
  public static B b;            public B() {
  public static int x;            A.foo();
                                }
  static {
    b = new B();                public static int bar() {
    x = 1;                        return A.x;
  }                             }
                              }
  public static void foo() {
    assert x > 0; // safe?
  }
}
```

- No – here is an example (consider calling `B.bar()` when `A` hasn’t been loaded):

```
public class A {              public class B {
  public static B b;            public B() {
  public static int x;            A temp = new A();
                                }
  static {
    b = new B();                public static int bar() {
    x = 1;                        return A.x;
  }                             }
                              }
  public A() {
    assert x > 0; // safe?
  }
}
```

- No – here is an example (consider calling `A.foo()` when neither class is loaded):

```
public class A {              public class B extends A {
  public static B b;            static {
  public static int x;            assert A.x > 0; //safe?
                                }
  static {
    b = new B();                public void bar() {
    b.bar();                      assert A.x > 0; //safe?
    x = 1;                      }
  }                           }

  public static void foo(){}
}
```

7. The classes will compile.

When the program is run, the output will be:

```
3
2
1
```

This is because, starting to initialise `A` causes `B` to start being initialised which causes `C` to start being initialised (at which point Java realises `A` has already started initialisation and just carries on initialising `C`). When `C.value` gets assigned, `A.value` still contains the default value `0`

The class we first mention will always get loaded first, and so complete initialisation last. By changing the order of the second two classes, we can vary the output between the one above, and:

```
3
1
2
```