ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 12

Initialization

December 14, 2012

Task 1

Consider the following abstract class, representing a node of a singly-linked list:

```
public abstract class ListNode<X> {
   public abstract void setItem(X x);
   public abstract X getItem();
   public abstract ListNode<X> getNext();
}
```

Consider now the following implementation using a simple (acyclic) list:

```
public class AcyclicListNode<X> extends ListNode<X> {
   protected X item;
   protected AcyclicListNode<X> next;

   public AcyclicListNode<X> (X item) {
      this.item = item;
      this.next = null;
   }

   public void setItem(X x) { item = x; }
   public X getItem() { return item; }
   public AcyclicListNode<X> getNext() { return next; }
}
```

In this implementation, suppose that an empty list is represented simply by a `null` reference. Suppose that a further design intention of this implementation is that each node is guaranteed to store an `X` object in its item field.

- Annotate the class `AcyclicListNode<X>` with appropriate non-null type annotations to express these design intentions as far as possible. You do not need any annotations from the Construction Types system (`free` or `unc` annotations).

Now consider an alternative implementation using a cyclic list:

```
public class CyclicListNode<X> extends ListNode<X> {
   protected X item;
   protected CyclicListNode<X> next;

   public CyclicListNode<X> (X item) {
      this.item = item;
      this.next = this;
   }

   public void setItem(X x) { item = x; }
   public X getItem() { return item; }
   public CyclicListNode getNext() { return next; }
}
```

In this implementation, the design intention is that every node will always have a next object in the list (sometimes itself). In this design, we choose to represent an empty list by a single node whose `next` field points to itself, but whose `item` field is `null`. All non-empty lists will be represented using only nodes whose `item` fields are non-null.

- Annotate the class `CyclicListNode<X>` with appropriate non-null type annotations to express these design intentions as far as possible. You do not need any annotations from the Construction Types system (`free` or `unc` annotations).
- Now consider how to annotate the method signatures in `ListNode<X>` so that both implementations can be accommodated. Your solution should be compatible with the usual co/contra-variance rules for subclass method signatures.

Task 2

Some languages like C++ provide numerous features that enable programmers to implement advanced types with very customizable behavior. In this question we explore how to implement a universal non-null pointer template class. Your task is to write the body of the `NN` class such that the `main` function below works as expected. Incorrect code should not compile if uncommented.

After you write the code, discuss how implementing non-null types in this way is different from built-in support for non-null types.

Hint: To fully answer this question you'll need to get familiar with the following:

- Defining templates.
- Defining copy and conversion constructors.
- Defining and overloading operators (=, *, ->, type conversions).
- Defining friend classes.

```
class BadNonNullCast{}; // Throw when casting null to a non-null pointer

template <class T>
class NN {

...
};

// Some classes
class A{ public: int v{}; };
class B : public A{};
class C : public B{};
class X {};

#include <iostream>
using namespace std;
int main()
{
   try
   {
      // Standard C++ pointers
      A* a = new A(); a->v = 1;
      B* b = new B(); b->v = 2;

      // Non-null pointers can be initialized from standard pointers.
      // Non-nullnes is checked at run-time.
      NN<A> nna = a;
      NN<B> nnb = b;
```

```
        NN<C> nnc = new C(); nnc->v = 3;

        // Subtyping works as expected
        C* c = nnc;
        nnb = nnc;
        nnb = c;
        b = nnc;

        // NN<T> behaves just like a standard pointer
        cout << (*a).v << (*nna).v << endl;
        cout << c->v << nnc->v << endl;

        // Non-null checks are performed at run time.
        try{ nnb = nullptr; }
        catch (BadNonNullCast&) {cout << "Expected exception"<<endl;}

        C* nullC = nullptr;
        try{ nnb = nullC; }
        catch (BadNonNullCast&) {cout << "Expected exception"<<endl;}

        X* x = new X();
        // Each of the following will cause a compile time error
        //NN<A> uninitialized;
        //nna = x;
        //nnc = nna;
        //nnc = a;
    }
    catch (BadNonNullCast&)
    {
        cout << "Unexpected Exception" << endl;
    }

    return 0;
}
```

Task 3

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.

Task 4

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.

Task 5

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.

Task 6

(From a previous exam)

Consider the following code in a Java-like language enriched with the non-null types 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 types system including construction types.

(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);
}
```

There are four assignments in the body of this method. Which of them type-check? Which of them do not? Why?

Task 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?