Exercise 13 Solutions

Non null Types and Invariants

1.
   a. False. A field declared to be possibly-null can always refer to null, even when the reference to the object whose field it is has a non-raw type.
   b. False. If the reference has a raw (non-null) type then the object might still have null values in its non-null fields (e.g., the start of a constructor call).
   c. True. A non-raw reference to an object can only exist after its constructor has executed (see previous sheet), by which time it is guaranteed to have non-null values in non-null fields by the definite assignment checks imposed on constructors.
   d. False. For example, at the very end of a constructor body the newly-constructed object will be locally initialised, but the `this` reference is still raw.
   e. False. For example, at the very end of a constructor body, the newly-constructed object will be locally initialised, but if it was passed a raw non-null argument then it might have assigned that to one of its fields.
   f. True. If `e` has a non-raw type then `e.f` has a non-raw type, and so must be locally initialised (see (c) above).
   g. True. No difference with the previous statement, since non-raw references always refer to locally initialised objects anyway (c).
   h. True. Only non-null values are allowed to be assigned to non-null fields.
   i. True. This is what the definite assignment checks guarantee.

2.
   •

```
public class DogOwner {
  Dog! dog;   // DogOwners must have a dog
  Bone! bone; // DogOwners must have a bone (for their dog)

  public DogOwner() {
    this.dog = new Dog(this);
    this.bone = dog.bone;
  }
}

public class Dog {
  DogOwner! owner; // Dogs must have an owner
  Bone! bone;      // Dogs must have a bone

  public Dog(raw DogOwner! owner) {
    this.owner = owner;
    this.bone = new Bone(owner, dog);
  }
}

public class Bone {
  DogOwner! owner; // Bones must have an owner
```

```
  Dog! dog;         // Bones must belong to a dog..

  public Bone(raw DogOwner! owner, raw Dog! dog) {
    this.owner = owner;
    this.dog = dog;
  }
}
```

- The line `this.bone = dog.bone;` will not type-check because `dog` is a raw reference (it is returned from a constructor which was passed a raw argument), and reading a field from a raw reference always yields a possibly-null type (which is not allowed to be assigned to a field declared with a non-null type).

- We can change the line to `this.bone = (!) dog.bone;`

- We can add the following post-condition to the `Dog` constructor:
`// ensures: this.bone != null`

3.

- Immediate from the definition of reachability.

- This follows by essentially "appending" the field accesses used – if we have o_2 == $o_1.f_1.f_2...f_m$ and o_3 == $o_2.g_1.g_2...g_n$ then by substituting the former in the latter, we also have o_3 == $o_1.f_1.f_2...f_m.g_1.g_2...g_n$

- Suppose o_1 is an object with one field `f`, which currently references a further object o_2 which has no fields. Then o_2 is reachable from o_1 but o_1 is not reachable from o_2.

  NOTE: At the start of a constructor or method call, consider the set of objects reachable via all the references in scope. This is basically all the objects reachable via the object referenced by `this` and the objects referenced by the parameters which were passed to the constructor/method. During the execution of the constructor/method (including any constructors/methods it calls, etc.), any field updates that take place must be assigning to and from objects which were already reachable. Therefore, field updates can't increase the set of objects which are reachable. The only extra objects which can become reachable are those which get newly allocated during the execution of the constructor/method.

-

  Recall that for any object to be reachable from `e` , there must be a chain of field accesses starting at `e` which refers to the object. We can show by induction on `n` that for any such reference involving `n` field accesses, i.e., an expression $e.f_1.f_2...f_n$, the reference will be non-raw, and any object referred to by the reference will be locally initialised. The argument goes as follows:
  For the base case, we need that the object which `e` itself refers to is locally initialised. This, we know by question 1(c). Note that the current question already stated that `e` is a non-raw reference.
  For the inductive case, assume $e.f_1.f_2...f_k$ is a non-raw reference, and that if it refers to an object, the object is locally initialised. Then we need to show that,

for any valid field f_{k+1} we can deduce that $e.f_1.f_2...f_k.f_{k+1}$ is a non-raw reference, and that if it refers to an object, the object is locally initialised. We know the reference is non-raw, by the type system rules – looking up a field on a non-raw reference always yields another non-raw reference. We conclude that any object referred to by the reference must be locally initialised, by question 1(g).

4.

- Suppose that expression `e` refers to an object and the object is not fully initialised. If it were possible that `e` had a non-raw type, then by question 3, we would deduce that all objects reachable from `e` are locally initialised, i.e., that `e` is fully initialised. This contradicts the assumption that `e` is not fully initialised, so it cannot be possible for `e` to have a non-raw type.

- Unfortunately, the question isn't quite correct as written (sorry!). The statement "Once an object is fully initialised, it will always remain fully initialised." is not actually always true. To see why, consider the following example code:

```
public class C {
  C! f;
  public C(raw C! x) {
    this.f = this; // this is fully initialised
    this.f = x; // this might not be fully initialised
  }
}
```

Once the constructor of `this` has finished executing, then we are no longer allowed to assign raw references to its fields (because of the restriction at the start of the sheet, as discussed last week) and so it's tempting to think that the statement would be true if we added the extra proviso that the constructor of the object had already run. But the following example shows that this is not enough:

```
public class C {
  D! f;
  public C(raw D! x) {
    this.f = x;
  }
}
```

```
public class D {
  D! f;
  C! g;
  public D(raw D! y) {
    this.f = this;
    this.g = new C(this);
    this.f = y;
  }
}
```

Consider an execution of `D`'s constructor. After the first two lines have been executed, the newly-created object of class `D` will be fully initialised, as will the newly-created object of class `C` which was created. However, when the third line of the constructor is executed, *both* of these objects are potentially no-longer fully initialised (even though their constructors have both finished executing).

In fact, the statement in the question should be weakened to the following:
"Once an object is fully initialised and the constructors of all objects reachable from it have been run, it will always remain fully initialised."

This weaker statement can be shown to be true as follows. Firstly, by the restriction on field updates, once the constructors of all reachable objects have been run, it will never be possible to assign raw references to those objects' fields. When we assign non-raw references to those object's fields, it might be that new objects are reachable via the references. But this is ok – by question 3, we know that all of these objects will be locally initialised as well. Therefore, we will always maintain the property that all reachable objects are locally initialised, i.e., that the original object is fully initialised.

•

Point during execution:	Reachable "raw" objects...	...which are locally initialised
Beginning of executing `DogOwner` constructor	O	none
Beginning of executing `Dog` constructor	O, D	none
Beginning of executing `Bone` constructor	O, D, B	none
End of executing `Bone` constructor	O, D, B	B
End of executing `Dog` constructor	O, D, B	D, B
End of executing `DogOwner` constructor	O, D, B	O, D, B

a) This follows from the definite assignment check for constructors.

b) Any objects which were created will have had their constructors run, so this follows from the previous part (and question 1(h)).

c) Since it is possible to pass a raw reference to an object whose constructor has not finished executing yet, there is no guarantee that it (or any other objects reachable from it) will be locally initialised.

d) By the previous part, there can be references in scope which refer to objects which are not yet locally initialised. Since these references might have been assigned to the fields of the newly-constructed object, we deduce that the object might not be "fully initialised".

e) If we could give a non-raw type to such an object, then, given the previous part, we would contradict the statement at the end of question 3 (i.e., we wouldn't be living up to the intended invariants of the type system).

f) Since the only objects reachable will be newly-created objects, this follows by parts (a) and (b).

g) This follows from the previous part, and the definition of "fully initialised".

h) We could potentially have non-raw arguments to such a constructor. But, by question 3, any objects reachable from these arguments are guaranteed to be locally initialised (and will remain so, by 1(h)). During the execution of the constructor, only these objects, plus any newly-allocated objects, will be reachable (see NOTE in question 3). Therefore, by the previous part of this question, we know that all reachable objects will be locally initialised.

i) The previous part tells us that the constructed object will be fully initialised. But, since reachability is transitive (question 3), for any object o reachable from

the constructed object, we know that all object reachable from o must also be locally initialised (since they are reachable from the constructed object too). This means that all such objects o must be fully initialised.

j) As explained in the previous questions, the expectations which the type system has for a non-raw reference type are that all objects reachable via the references are fully initialised. By the previous part of the question, we know that these expectations are met when a constructor which was passed no raw references terminates. Therefore, it is safe to give the returned reference a non-raw type. Furthermore, note that in this situation we also know that all objects reachable from the newly-constructed object have had their constructors run. Therefore, the *amended* quoted statement in question 3 guarantees that the expectations of a non-raw reference will also continue to be satisfied as the program executes further.

5. a) The invariant can be written as follows:

```
∀i.0≤i<theTree.length/2⇒
        theTree[i] = theTree[2*i+1]+theTree[2*i+2]
```

Note that the condition `0≤i<theTree.length/2` says that node `i` is not a leaf (proof by induction on the height). Note also that "height" means the maximum distance of the root to the leaves (so a single node is a 0-height tree)

Of course, there should also be an invariant saying that the tree is complete:

$\exists h{:}\texttt{int}.\ h \geq 0 \wedge \texttt{theTree.length} = 2^{h+1} - 1$

b)

(a) The method clearly does not preserve the invariant. For example, imagine a three-node tree [10,5,5] and a call to `addToNode (0, 100)`

(b) When `addToLeaf` is called on a leaf, a sequence of recursive calls to `addToNode` begins. The first call adds a number `s` to the leaf, which temporarily breaks the invariant, because the parent of that leaf no longer holds the correct sum. Each subsequent call of `addToNode` corrects the sum of its current node, similarly making the sum of its parent (if there is one) outdated. The calls to `addToNode` happen recursively all the way up from the leaf to the root, at which point the invariant is fixed.

So, either the method `addToNode` is called on a leaf or the invariant must be broken exactly at the node we call `addToNode`. Furthermore, the sum of the children of that node must be exactly `s` less than what it is supposed to be.

(c) Precondition for `addToNode` that expresses this requirement:

```
    theTree.length/2≤i<theTree.length
∨   (( ∀j.0≤j<theTree.length/2 ∧ j≠i ⇒
        theTree[j] = theTree[2*j+1]+theTree[2*j+2])
    ∧ theTree[i] = theTree[2*i+1]+theTree[2*i+2]-s)
```

(d) **Note:** *the original exercise sheet provided a wrong precondition for* `addToLeaf`. *The new exercise sheet corrects the mistake. Sorry about that.*

The method addToNode is private and therefore can be called only from two places: The first place is addToLeaf, which, by its precondition, satisfies the first disjunct of the precondition of addToNode.

The second place is recursively from `addToNode` itself, if `i>0`. Assuming that the precondition a call of `addToNode` holds, we need to show that the precondition also holds when we make a recursive call to `addToNode`.

Let `o` be the value of the old tree and `t` be the value of the new tree. Let `L` be the length of both trees. Then assumption becomes:

```
    L/2 ≤ i <L
∨   ((∀j.0≤j<L/2 ∧ j≠i ⇒o[j]=o[2*j+1]+o[2*j+2])
    ∧ o[i]=o[2*i+1]+o[2*i+2]-s)
```

and the two trees are connected by the relation

```
(∀j. 0≤j<L ∧ j≠i  ⇒ o[j]=t[j]) ∧ t[i]=o[i]+s
```

We need to show (for `i>0`) that:

```
    L/2 ≤ i/2 < L
∨   (( ∀j.0≤j<L/2∧j≠(i-1)/2 ⇒ t[j]=t[2*j+1]+t[2*j+2])
    ∧ t[(i-1)/2] = t[2*((i-1)/2)+1]+t[2*((i-1)/2)+2]-s)
```

We can get the first disjunct out of the way, since it is false anyway. It suffices to prove that

```
(( ∀j.0≤j<L/2∧j≠(i-1)/2⇒
                t[j] = t[2*j+1]+t[2*j+2])
∧ t[(i-1)/2] = t[2*((i-1)/2)+1]+t[2*((i-1)/2)+2]-s)
```

Consider the last conjunct. Exactly one of the two indices `2*((i-1)/2)+1` and `2*((i-1)/2)+2` is equal to `i`. By the relationship between `o,t` the last conjunct becomes:

```
o[(i-1)/2] = o[2*((i-1)/2)+1]+o[2*((i-1)/2)+2]+s-s
```

which becomes

```
o[(i-1)/2] = o[2*((i-1)/2)+1]+o[2*((i-1)/2)+2]
```

From the universal quantification, we break the case `j=i`. The whole formula becomes:

```
(  ∀j.0≤j<L/2 ∧ j≠(i-1)/2 ∧ j≠i ⇒
                  t[j]=t[2*j+1]+t[2*j+2])
∧ t[i]=t[2*i+1]+t[2*i+2]
∧ o[(i-1)/2] = o[2*((i-1)/2)+1]+o[2*((i-1)/2)+2]
```

By the relationship between the trees (note that none of `2*j+1` and `2*j+2` can be equal to `i`, if `j≠i/2`):

```
   ( ∀j.0≤j<L/2 ∧ j≠(i-1)/2 ∧ j≠i ⇒
                o[j]=o[2*j+1]+o[2*j+2])
 ∧ o[i]+s=o[2*i+1]+o[2*i+2]
 ∧ o[(i-1)/2] = o[2*((i-1)/2)+1]+o[2*((i-1)/2)+2]
```

Finally, we combine the first and the third conjunct, and we get exactly the precondition (of the original call) that we assumed holds.

(e) To show now that, given the precondition of addToNode holds in the beginning, then the invariant holds in the end, notice that the method does not make further calls to itself if and only if i=0. In that case, given that the precondition holds in the beginning of the call:

```
          theTree.length/2≤0<theTree.length
 ∨   (( ∀j.0<j<theTree.length/2 ⇒
           theTree[j] = theTree[2*j+1]+theTree[2*j+2])
     ∧ theTree[0] = theTree[1]+theTree[2]-s)
```

After the call, `theTree[0]` is incremented by `s`, and no other change happens. So we have:

```
     (∀j.0<j<theTree.length/2 ⇒
           theTree[j] = theTree[2*j+1]+theTree[2*j+2])
     ∧ theTree[0] = theTree[1]+theTree[2]
```

which is equivalent to the invariant.