Alexander J. Summers

Program Verification

Exercise Solutions 13:

Permissions and Concurrent Programs

Assignment 1 (Postcondition Permissions)

In Viper, function postconditions are not required to specify the permissions "returned" when the function is invoked; this is because such functions cannot have side-effects, including on the permissions held; they can be seen as evaluated in a fixed program state.

Viper methods, on the other hand, are required to explicitly specify permissions returned in their postconditions; there is no assumption that the permissions in the precondition will necessarily be the same as those in the postcondition. In the lectures, we briefly discussed two reasons for this. Firstly, the permissions might be organised differently into different predicates (e.g. in the `prependLSeg` method, from the `list-examples.vpr` file). Secondly, a method might allocate new objects (and gain corresponding new permissions, e.g. via `inhale` statements); we want to be able to return these extra permissions to the method caller.

Consider the encodings of concurrency features (structured parallelism and locks) presented in slide deck 12. For such concurrent programs, it is sometimes *necessary* that a method postcondition describes *fewer* permissions than were present in the method's precondition.

1. Using the class from slide 276 as an example:

```
method relLock(x:LockableCounter)
  requires acc(x.count) && x.count > 0 // implicitly: ensures true
{
  release x
}
```

2.

```
method relLock(x:Ref)
  requires acc(x.count) && x.count > 0 // implicitly: ensures true
{
  exhale acc(x.count) && x.count >= 0 // exhale the lock invariant
}
```

3. The permission to `x.count` is returned to the lock invariant (ready for the next thread to lock the object).

4. A newly-forked thread would also entail an `exhale` of permissions. However, using only structured parallelism, any such thread will also be joined in the same method body, and if the method this thread executes has no way to "remove" permissions, then we will not be able to construct an analogous example in which some permissions conceptually *must* be transferred elsewhere by the end of a method execution.

5. With unstructured parallelism, it would be possible to fork a thread but not join it in the same method scope. In this case, any permissions required by the newly-forked thread's precondition would be removed and not returned to the forking method's scope, as for the `relLock` example above.

Assignment 2 (Deadlock Avoidance)

1. We should be able to assert false because at that point we would have more than a full permission to the field `holds`.

2. Being able to assert false at some program point means that there is no trace that could reach that point. This is exactly the case in the given example because the thread executing the method will deadlock when it tries to acquire `l` the second time.

3. Yes. If we modelled `holds` as a predicate, we would not be able to assert false because in Viper it is allowed to have more than one full permission to a predicate.