ETH zürich

Alexander J. Summers

Program Verification

Exercise Sheet 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. Give an example of such a method, using locks (using the high-level syntax for `acquire` and `release` as shown in the lecture).
2. Show how the method would be encoded into Viper, using the techniques from the lecture.
3. Conceptually, where do the permissions present in the method precondition but missing from the method postcondition go?
4. Why is it difficult to come up with an analogous example using (only) the structured parallelism features explained in the lecture?
5. Would this situation change if we could use *unstructured* parallelism (i.e. allow threads to be forked and joined in different methods/scopes)?

Assignment 2 (Deadlock Avoidance)

Let's consider the following method that tries to acquire the same lock twice:

```
def deadlockExample(l: Lock) {
    acquire l;
    acquire l;
    assert A;
}
```

In a programming language in which locks are not reentrant, this method would deadlock.

1. What could you prove in the state after the deadlock if you used the encoding of locks given on slides 292-295? In other words, what assertion `A` you expect to be able to verify?
2. Why is your expectation justified with respect to the runtime behaviour of the program?
3. Suppose we model `holds` not as a field, but as a predicate. That is, we encode `acquire l` as:

   ```
   inhale holds(l) && inv[l/this]
   ```

   Would this change the answer to the first question?