ETH zürich

Alexander J. Summers

Program Verification

Exercise Solutions 11:

Permissions and Concurrent Programs

Assignment 1 (Encoding Non-Determinism)

1. To avoid unjustified assumptions about several `havoc` statements yielding the same value, we would need one extra parameter per `havoc` statement potentially executed in the method body.

2. Methods containing `havoc` statements inside (unbounded) loops would need an statically-unbounded number of extra parameters.

3. We could use an additional `Ref` value, and a *field location* of this `Ref` per type, to generate fresh values of that type. We could use an extra parameter for this `Ref`; alternatively, we could add a function `extraRef() : Ref)` to the program. Then, to simulate e.g. `havoc x` statements for *integer-typed variables* `x`, we add a field `intField : Int` to the program (of course, we should avoid clashes with any existing fields in the program, or else reuse one of those fields).

   We now encode a `havoc x` statement by temporarily adding permission to the extra field location, reading its (arbitrary, unconstrained) value, and then removing the permission; i.e. we would generate the following code to simulate a `havoc x` statement:

   ```
   inhale acc(extraRef().intField)
   x := extraRef().intField // read some value
   exhale acc(extraRef().intField)
   ```

4. This approach can use the above code for each `havoc` statement; there is no restriction on the number of such statements, since each time this code is executed, a newly-unconstrained value will be generated (we keep no permission to the field(s) in between).

5. A non-deterministic choice `s1[]s2` can be encoded as an if-condition on a `havoc`-ed boolean value. Assuming we introduce an extra field `boolField : Bool` to the program, then such a non-deterministic choice could be handled via:

```
var b: Bool // should be a fresh variable name for the program
inhale acc(extraRef().boolField)
b := extraRef().boolField
exhale acc(extraRef().boolField)
if(b) {
  s1
} else {
  s2
}
```

Assignment 2 (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.