ETH zürich

Alexander J. Summers

Program Verification

Exercise Sheet 11: Permissions and Concurrent Programs

Assignment 1 (Encoding Non-Determinism)

We've already seen one way to encode a `havoc x` statement in Viper: by calling an abstract method which returns the appropriate type (slide 257). It might be tempting to simulate a `havoc x` statement by adding additional *parameters* to the enclosing method (without any constraints in the precondition, these parameters will have unknown values, which could then be assigned to e.g. the `x` at the point of the intended `havoc`). This has the obvious disadvantage that a caller of the current method will have to provide values for these parameters. Ignoring this problem, the approach is also insufficient for reflecting the correct behaviour of `havoc x` statements, in general.

1. How many extra parameters would be needed, to eliminate the `havoc` statements from a given method body?
2. Why does this mean that *some* method bodies could not be handled by this approach?
3. Give a different approach for encoding a `havoc x` statement, using `inhale` and `exhale` operations.
4. Does your approach suffer from the same problems?
5. Show how to encode a non-deterministic choice statement $s_1 \,[]\, s_2$, using your ideas.

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. 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)?