ETH zürich

Alexander J. Summers

# Program Verification

## Exercise Solutions 12:

## 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 (Graph Marking)

The complete example can be found on the Viper examples page at `http://viper.ethz.ch/examples/graph-marking.html`.

1. The method `trav_rec` takes full permission to all fields of each node. As a result, the caller has to havoc all its knowledge about what values these fields have. Therefore, if the method `trav_rec` did not explicitly ensure that the nodes are not modified, the caller would not know if the marked graph is the same one as the original one.

   An alternative way of ensuring that the graph is not changed would be to pass only read permissions to fields `left` and `right`. This can be done by introducing a ghost parameter `p` of type `Perm` that is required to be strictly between zero and full permission. This ghost parameter then could be used as a permission amount for fields `left` and `right`. Please note that in the recursive call, a strictly smaller value than `p` must be passed (for example, `p/2`), otherwise the caller will still have to havoc its knowledge about the graph.

2. Each call of the method `trav_rec` marks exactly one node that is not marked from the set `nodes`. In other words, the number of not marked nodes strictly decreases with each call. Since the number of nodes is non-negative, the method is guaranteed to terminate if the set `nodes` is not infinite. In Viper there is a cardinality function that maps from sets to integers, therefore, Viper sets can only be finite. As a result, there is no need to explicitly require that the set `nodes` is finite.