ETH zürich

Alexander J. Summers

Program Verification

Exercise Solutions 11: Unbounded Heap Data Structures

Assignment 1 (List Segments)

1. A cyclic list starting and ending at reference `x` can be represented by a predicate instance `lseg(x, x)`.

2.
```
predicate mylseg(start: Ref, end: Ref) {
  start != end ==>
    acc(start.val) && acc(start.next) && mylseg(start.next,end)
}
```

3. An instance of this predicate cannot represent cyclic lists. The problem is that an assertion `mylseg(x,x)` is guaranteed not to contain any permissions, regardless of the value of e.g. `x.next`; the recursive definition terminates too early.

4. The following implementations work. Note the `assert` statements, the purpose of which is to allow the verifier to learn the contents of the predicates. You can achieve the same effect by unfolding the predicates and immediatelly folding them as was shown in the class.

```
method addAtEnd(l1: Ref, l2: Ref)
  requires lseg(l1, l2)
  requires acc(l2.val) && acc(l2.next) && list(l2.next)
  ensures lseg(l1, old(l2.next))
  ensures lsegelems(l1, old(l2.next))
            == old(lsegelems(l1, l2)) ++ Seq(old(l2.val))
  ensures list(old(l2.next))
  ensures elems(old(l2.next)) == old(elems(l2.next))
{
  unfold lseg(l1,l2)
  var tmp : Ref := l2.next
  if(l1.next == l2) {
    assert unfolding list(l2.next) in l1.next != l2.next
    fold lseg(l2, tmp)
    fold lseg(l1, tmp)
  } else {
```

```
    assert unfolding lseg(l1.next, l2) in
      unfolding list(l2.next) in l1.next != l2.next
    addAtEnd(l1.next, l2)
    fold lseg(l1, tmp)
  }
}

method prependLseg(l1: Ref, l2: Ref)
  requires lseg(l1, l2) && list(l2)
  ensures list(l1)
  ensures elems(l1) == old(lsegelems(l1, l2) ++ elems(l2))
  {
    unfold lseg(l1, l2)
    if(l1.next != l2) {
      prependLseg(l1.next, l2)
      assert unfolding list(l1.next) in l1.next != null
      fold list(l1)
    } else {
      assert unfolding list(l2) in l2 != null
      fold list(l1)
    }
  }
```

Assignment 2 (Heap-based Matrices)

1. ```
   domain HeapMatrix {
     function cell(m: HeapMatrix, i: Int, j:Int): Ref
     function dim(m: HeapMatrix): Int
     // for expressing injectivity:
     function first(r: Ref): HeapMatrix
     function second(r: Ref): Int
     function third(r: Ref): Int

     // injectivity:
     axiom all_diff {
       forall m: HeapMatrix, i: Int, j: Int :: {cell(m, i, j)}
         first(cell(m, i, j)) == m && second(cell(m, i, j)) == i
           && third(cell(m, i, j)) == j
     }

     axiom dim_nonneg {
       forall m: HeapMatrix :: {dim(m)} dim(m) >= 0
     }
   }
   field val : Int
   ```

2. For a given matrix `m`, the assertion would be
   ```

```
forall i:Int, j:Int ::
  0 <= i && i < dim(m) && 0 <= j && j < dim(m)
    ==> acc(cell(m,i,j).val)
```

3. We could represent square matrices of size `N` as arrays of size `N*N`, e.g. representing the (i, j)-th cell with the location `loc(a,`i`*N+`j`)`. In this representation, permission to the whole matrix would be represented by the assertion

   ```
   forall i:Int :: 0 <= i && i < size(a) ==> acc(loc(a,i).val)
   ```

   which is supported by the current tools. However, assertions denoting permission to single rows and columns of the matrix, or functional properties of these (e.g. loop invariants describing an operation which has so-far been performed on only a part of the matrix) will need to employ i`*N+`j expressions to describe the appropriate matrix regions. The terms in which these expressions occur can then typically not be used in triggers for the corresponding quantifiers, due to the usage of interpreted arithmetic operators. Furthermore, this encoding employs non-linear arithmetic, and support for this (undecidable) theory in the SMT solver is typically unreliable.

Assignment 3 (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 4 (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.