ETH zürich

Alexander J. Summers

Program Verification

Exercise Solutions 10: Unbounded Heap Data Structures

Assignment 1 (List Segments)

Consider the `lseg` predicate from Slide 225.

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, which guarantee that the definitions of the relevant functions don't terminate "too early":

```
method addAtEnd(l1:Ref, l2:Ref)
  requires lseg(l1,l2) && acc(l2.val) && acc(l2.next) && list(l2.next)
  // last conjunct above was added (compared with original sheet)
  ensures lseg(l1,old(l2.next)) && lsegelems(l1,old(l2.next)) ==
    old(lsegelems(l1,l2)) ++ Seq(old(l2.val))
  // this next postcondition was added (compared with original sheet)
  ensures list(old(l2.next)) && 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) && 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.