ETH zürich

Alexander J. Summers

Program Verification

Exercise Sheet 11: Unbounded Heap Data Structures

Assignment 1 (List Segments)

Consider the `lseg` predicate from slide 239.

1. Write an assertion using this predicate to describe a cyclic list.
2. The `lseg` predicate does not allow for "empty" segments; permission to the `start` reference's fields is always contained inside. Write a definition for an alternative `mylseg` predicate, which allows for possibly-empty list segments.
3. Can you use `mylseg` to describe cyclic lists?
4. Consider the `addAtEnd` and `prependLseg` methods (slide 247), which were needed in order to verify the iterative version of list append, in the lectures. These methods were not implemented in the lecture; in the code listed below the method declarations are included, but no bodies. Implement these methods, such that your resulting code verifies.

```
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))

method prependLseg(l1: Ref, l2: Ref)
  requires lseg(l1, l2) && list(l2)
  ensures list(l1)
  ensures elems(l1) == old(lsegelems(l1, l2) ++ elems(l2))
```

Assignment 2 (Heap-based Matrices)

On slide 253, a Viper encoding of arrays is shown, using a custom domain and quantified permission assertions. Suppose that we want to implement a similar encoding for heap-based *square matrices*: a special case of two-dimensional arrays.

1. Write a corresponding `Matrix` domain definition (you might want to borrow ideas from the `Array` domain).
2. What assertion would you use to describe full permission to all elements of a particular matrix? (Hint: it should involve two quantifiers.)
3. The current version of Carbon (one of the Viper verification back-ends) does not support quantified permissions under multiple (nested) quantifiers; only single quantifiers are supported. Describe an alternative representation of matrices which requires only a single quantifier, using the original `Array` domain. Can you think of any practical disadvantages of this encoding using single quantifiers, compared to the more-direct two-dimensional quantification? (Hint: there may be more than one problem.)

Assignment 3 (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 277). 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 $s_1 \texttt{[]} s_2$, using your ideas.

Assignment 4 (Graph Marking)

You can find below the encoding of a recursive graph marking algorithm. The encoding is incomplete: all quantifiers in the encoding are missing triggers. Please complete the example by specifying the missing triggers and answer the following questions:

1. Why do we need to explicitly ensure that the nodes are not modified? Can you think of a different way of ensuring this property, other than the one used in the example?
2. Under which conditions is the method `trav_rec` guaranteed to terminate? How would you informally justify that? Does the method precondition already require the necessary conditions, or would you need to explicitly write them?

```
field left: Ref
field right: Ref
field is_marked: Bool

define INV(nodes)
    !(null in nodes)
    && (forall n: Ref :: n in nodes ==> acc(n.left))
    && (forall n: Ref :: n in nodes ==> acc(n.right))
    && (forall n: Ref :: n in nodes ==> acc(n.is_marked))
    && (forall n: Ref ::
            n in nodes && n.left  != null ==> n.left  in nodes)
    && (forall n: Ref ::
            n in nodes && n.right != null ==> n.right in nodes)

method trav_rec(nodes: Set[Ref], node: Ref)
    requires node in nodes && INV(nodes)
    requires !node.is_marked

    ensures node in nodes && INV(nodes)

    // We do not unmark nodes. This allows us to prove that the
    // current node will be marked.
    ensures forall n: Ref ::  n in nodes
                ==> (old(n.is_marked) ==> n.is_marked)
    ensures node.is_marked

    /* The nodes are not being modified. */
    ensures forall n: Ref ::  n in nodes
                ==> (n.left == old(n.left))
    ensures forall n: Ref ::  n in nodes
                ==> (n.right == old(n.right))

    /* Propagation of the marker. */
    ensures forall n: Ref ::
                n in nodes ==> (
                    old(!n.is_marked)
                    && n.is_marked ==> (
                        n.left == null || n.left.is_marked
                    )
                )
    ensures forall n: Ref ::
                n in nodes ==> (
```

```
                    old(!n.is_marked)
                    && n.is_marked ==> (
                        n.right == null || n.right.is_marked
                    )
                )
{
    node.is_marked := true

    if (node.left != null && !node.left.is_marked) {
        trav_rec(nodes, node.left)
    }
    if (node.right != null && !node.right.is_marked) {
        trav_rec(nodes, node.right)
    }
}

method client_success() {
    var a: Ref; a := new(*); a.is_marked := false
    var b: Ref; b := new(*); b.is_marked := false

    a.left := b;    a.right := null
    b.left := null; b.right := a

    var nodes: Set[Ref] := Set(a, b)
    assert forall n: Ref :: n in nodes ==> !n.is_marked

    trav_rec(nodes, a)
    assert forall n: Ref :: n in nodes ==> n.is_marked
}

method client_failure() {
    var a: Ref; a := new(*); a.is_marked := false
    var b: Ref; b := new(*); b.is_marked := false

    a.left := a; a.right := a;
    b.left := a; b.right := a;

    var nodes: Set[Ref] := Set(a, b)
    assert forall n: Ref :: n in nodes ==> !n.is_marked

    trav_rec(nodes, a)

    // The assertion is expected to fail because b is in nodes,
    // but b is not reachable from a
    assert forall n: Ref :: n in nodes ==> n.is_marked
}
```