

# Software Verification

## Master Programme in Computer Science

Mário Pereira      `mjp.pereira@fct.unl.pt`

Nova School of Science and Technology, Portugal

November 11, 2025

### Lecture 10

based on previous editions by João Seco, Luís Caires, and Bernardo Toninho  
also based on lectures by Jean-Marie Madiot and Arthur Charguéraud

The second handout is due for this Saturday, at 23:59.

Delivery method: send an email to

`mjp.pereira@fct.unl.pt`

with subject **[CVS-2025] Handout 2.**

You should attach to your email

- Your solution `dynamic.mlw` file.
- The content of the sub-directory `dynamic` generated by Why3. In particular, this directory **must contain** session files `why3session.xml` and `why3shapes.gz`.
- A `report.pdf` document, in which you report on your work.

The email should contain information about the team members

- numbers
- names

I will not be here **next Wednesday**.

If you can, please attend lab session at **Tuesday**.

- At 2 pm
- Room 119

Next week, I will open a poll to check if you need an extra lab session.

# Verification of Heap-Manipulating Programs (2/n)

---

1. Separation Logic – Recap
2. Lists and List Segments in Separation Logic
3. Trees in Separation Logic
4. Arrays in Separation Logic

## Separation Logic – Recap

---

# Basics of Separation Logic – Recap

We will use the following grammar for Separation Logic assertions:

$A ::=$	Separation Logic Assertions
$L \leadsto V$	Points-to
$  H_1 \star H_2$	Separating conjunction
$  \text{emp}$	Empty heap
$  B$	Boolean condition (pure predicate)
$  B ? A : A$	Conditional

$B ::= B \wedge B \mid B \vee B \mid V = V \mid V \neq V \mid \dots$

$V ::= \dots$  Pure expressions

$L ::= x.\ell$  Object field

# Rules of Separation Logic, Assignment – Recap

Recall the Hoare Logic assignment rule:

$$\frac{}{\text{Hoare} \{P[x \mapsto t]\} x := t \{P\}}$$

The assignment rule in Separation Logic is

$$\frac{}{\text{SL} \{x \rightsquigarrow V\} x := t \{x \rightsquigarrow t\}}$$

Note the **small footprint** principle, the precondition refers **exactly** to the part of the memory used by the fragment.

# Example of Separation Logic in Verifast

```
/*@ predicate Node(Node t; Node n, int v) =
    n.nxt |-> n &*& n.val |-> v;
    predicate List(Node n;) =
        n == null ? emp : Node(n,?h,_) &*& List(h);
    predicate StackInv(Stack s;) = s.head |-> ?h &*& List(h);
@*/

public class Stack {
    Node head;

    public Stack()
        //@ requires true;
        //@ ensures StackInv(this);
    { head = null; }

    public int pop()
        //@ requires NonEmptyStack(this);
        //@ ensures StackInv(this);
    { int v = head.getval(); head = head.getnext(); return v; }

    public boolean isEmpty()
        //@ requires StackInv(this);
        //@ ensures result ? StackInv(this) : NonEmptyStack(this);
    { return head == null; }
```

## Example of Separation Logic in Verifast

```
public static void main()  
  //@ requires true;  
  //@ ensures true;  
  {  
    Stack s = new Stack();  
    s.push(1);  
    if (! s.isEmpty()) {  
      //@ open(NonEmptyStack(s));  
      s.pop();  
    }  
    s.push(2);  
    s.push(3);  
    s.pop();  
  }
```

# Lists and List Segments in Separation Logic

---

Separation Logic allows us to reason about:

- pointer manipulating programs
- dynamically-allocated data structures

Lets look at some more serious examples of this.

We can define the predicate `isList L p`, denoting

- `p` is the pointer to a heap-allocated list
- `L` is the logical sequence of elements in the list

Recursive definition of `isList`:

$$\text{isList}(L, p) \triangleq \begin{cases} \text{emp} \wedge L = [] & \text{if } p = \text{null} \\ \exists x, L', p' \bullet p.\text{val} \rightsquigarrow x \star p.\text{next} \rightsquigarrow p' \\ \quad \star \text{isList}(L', p') \star L = x :: L' & \text{if } p \neq \text{null} \end{cases}$$

Consider the following program:

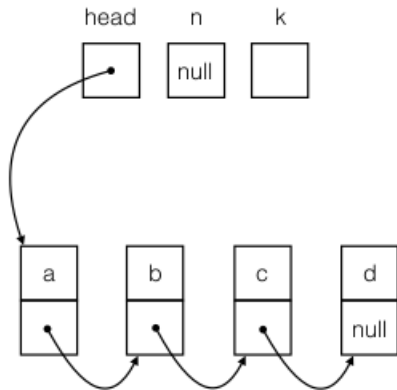
```
public class List {  
    Node head;  
    void mystery()  
    {  
        Node n = null;  
  
        while (head != null)  
        {  
            Node k = head.next;  
            head.next = n;  
            n = head;  
            head = k;  
        }  
        head = n;  
    }  
}
```

What does this method do?

# Lists in Separation Logic

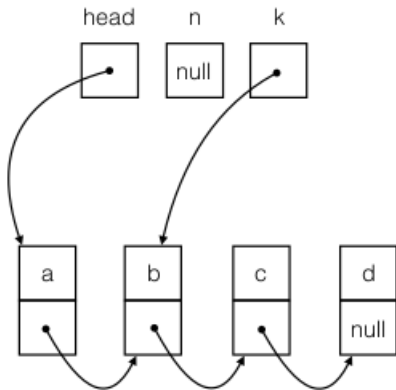
Consider the following program:

```
public class List {  
    Node head;  
    void mystery()  
    {  
        Node n = null;  
  
        while (head != null)  
        {  
            Node k = head.next;  
            head.next = n;  
            n = head;  
            head = k;  
        }  
        head = n;  
    }  
}
```



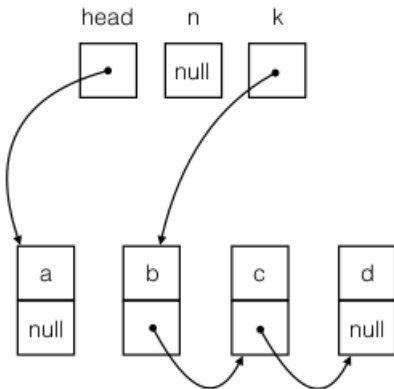
Consider the following program:

```
public class List {  
    Node head;  
    void mystery()  
    {  
        Node n = null;  
  
        while (head != null)  
        {  
            Node k = head.next;  
            head.next = n;  
            n = head;  
            head = k;  
        }  
        head = n;  
    }  
}
```



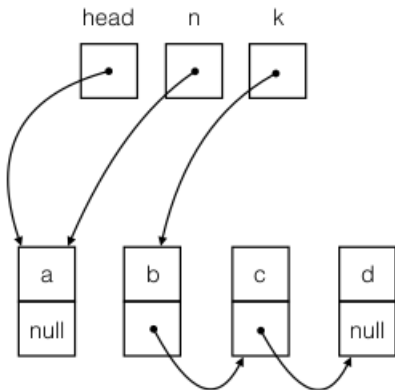
Consider the following program:

```
public class List {  
    Node head;  
    void mystery()  
    {  
        Node n = null;  
  
        while (head != null)  
        {  
            Node k = head.next;  
            head.next = n;  
            n = head;  
            head = k;  
        }  
        head = n;  
    }  
}
```



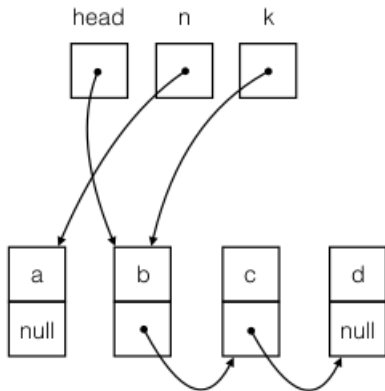
Consider the following program:

```
public class List {  
    Node head;  
    void mystery()  
    {  
        Node n = null;  
  
        while (head != null)  
        {  
            Node k = head.next;  
            head.next = n;  
            n = head;  
            head = k;  
        }  
        head = n;  
    }  
}
```



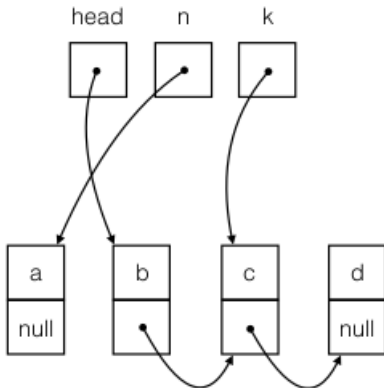
Consider the following program:

```
public class List {  
    Node head;  
    void mystery()  
    {  
        Node n = null;  
  
        while (head != null)  
        {  
            Node k = head.next;  
            head.next = n;  
            n = head;  
            head = k;  
        }  
        head = n;  
    }  
}
```



Consider the following program:

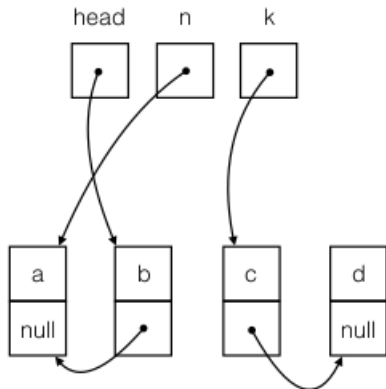
```
public class List {  
    Node head;  
    void mystery()  
    {  
        Node n = null;  
  
        while (head != null)  
        {  
            Node k = head.next;  
            head.next = n;  
            n = head;  
            head = k;  
        }  
        head = n;  
    }  
}
```



# Lists in Separation Logic

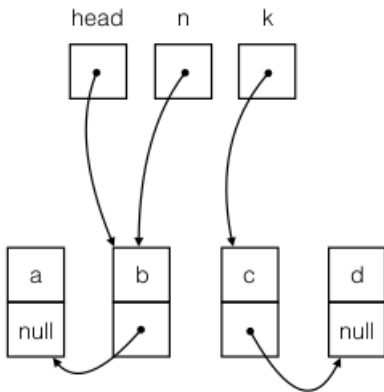
Consider the following program:

```
public class List {  
    Node head;  
    void mystery()  
    {  
        Node n = null;  
  
        while (head != null)  
        {  
            Node k = head.next;  
            head.next = n;  
            n = head;  
            head = k;  
        }  
        head = n;  
    }  
}
```



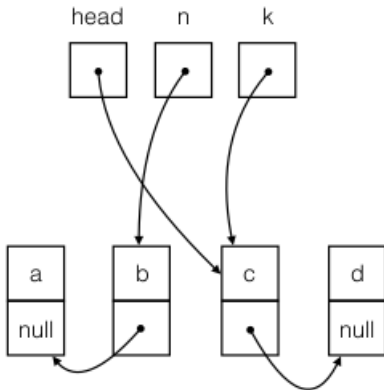
Consider the following program:

```
public class List {  
    Node head;  
    void mystery()  
    {  
        Node n = null;  
  
        while (head != null)  
        {  
            Node k = head.next;  
            head.next = n;  
            n = head;  
            head = k;  
        }  
        head = n;  
    }  
}
```



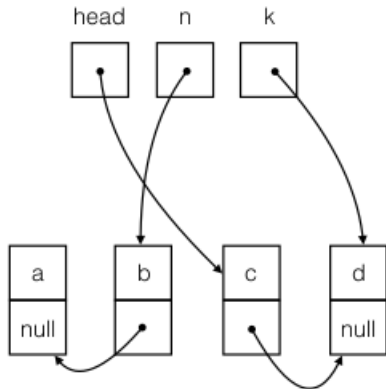
Consider the following program:

```
public class List {  
    Node head;  
    void mystery()  
    {  
        Node n = null;  
  
        while (head != null)  
        {  
            Node k = head.next;  
            head.next = n;  
            n = head;  
            head = k;  
        }  
        head = n;  
    }  
}
```



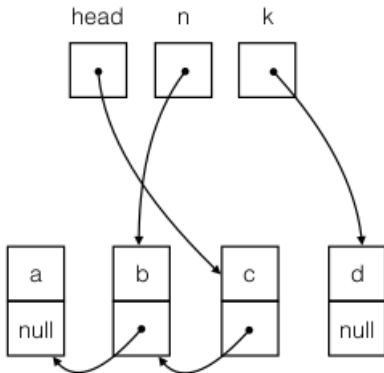
Consider the following program:

```
public class List {  
    Node head;  
    void mystery()  
    {  
        Node n = null;  
  
        while (head != null)  
        {  
            Node k = head.next;  
            head.next = n;  
            n = head;  
            head = k;  
        }  
        head = n;  
    }  
}
```



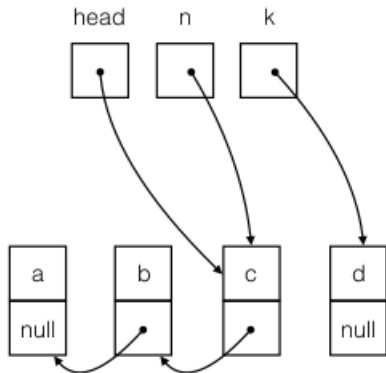
Consider the following program:

```
public class List {  
    Node head;  
    void mystery()  
    {  
        Node n = null;  
  
        while (head != null)  
        {  
            Node k = head.next;  
            head.next = n;  
            n = head;  
            head = k;  
        }  
        head = n;  
    }  
}
```



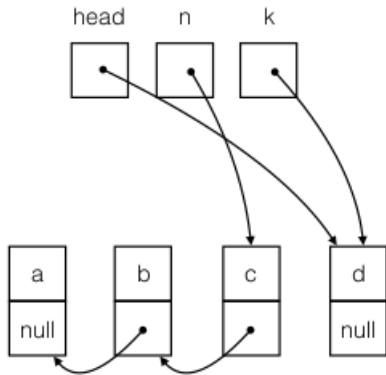
Consider the following program:

```
public class List {  
    Node head;  
    void mystery()  
    {  
        Node n = null;  
  
        while (head != null)  
        {  
            Node k = head.next;  
            head.next = n;  
            n = head;  
            head = k;  
        }  
        head = n;  
    }  
}
```



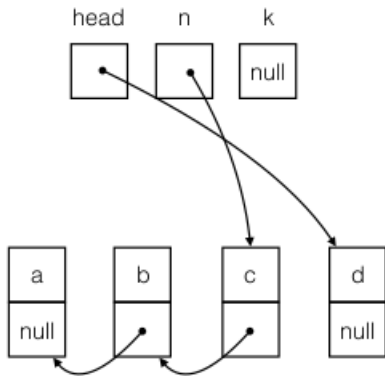
Consider the following program:

```
public class List {  
    Node head;  
    void mystery()  
    {  
        Node n = null;  
  
        while (head != null)  
        {  
            Node k = head.next;  
            head.next = n;  
            n = head;  
            head = k;  
        }  
        head = n;  
    }  
}
```



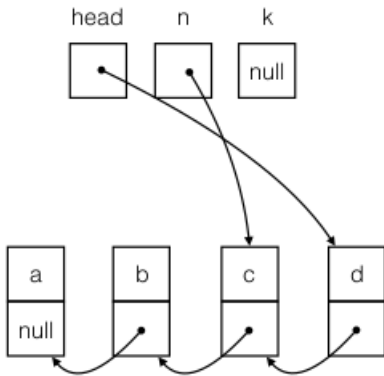
Consider the following program:

```
public class List {  
  Node head;  
  void mystery()  
  {  
    Node n = null;  
  
    while (head != null)  
    {  
      Node k = head.next;  
      head.next = n;  
      n = head;  
      head = k;  
    }  
    head = n;  
  }  
}
```



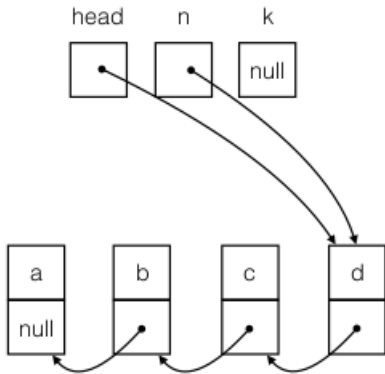
Consider the following program:

```
public class List {  
    Node head;  
    void mystery()  
    {  
        Node n = null;  
  
        while (head != null)  
        {  
            Node k = head.next;  
            head.next = n;  
            n = head;  
            head = k;  
        }  
        head = n;  
    }  
}
```



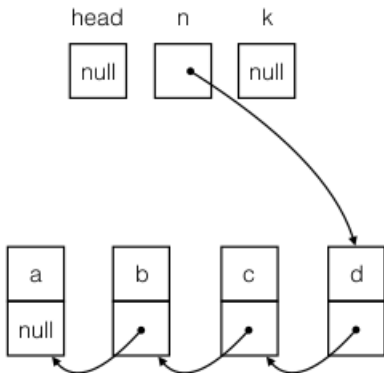
Consider the following program:

```
public class List {  
    Node head;  
    void mystery()  
    {  
        Node n = null;  
  
        while (head != null)  
        {  
            Node k = head.next;  
            head.next = n;  
            n = head;  
            head = k;  
        }  
        head = n;  
    }  
}
```



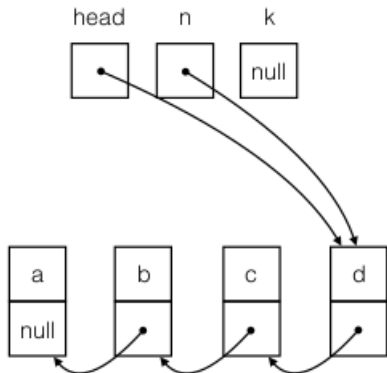
Consider the following program:

```
public class List {  
    Node head;  
    void mystery()  
    {  
        Node n = null;  
  
        while (head != null)  
        {  
            Node k = head.next;  
            head.next = n;  
            n = head;  
            head = k;  
        }  
        head = n;  
    }  
}
```



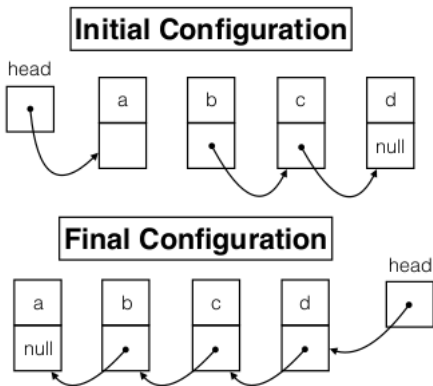
Consider the following program:

```
public class List {  
    Node head;  
    void mystery()  
    {  
        Node n = null;  
  
        while (head != null)  
        {  
            Node k = head.next;  
            head.next = n;  
            n = head;  
            head = k;  
        }  
        head = n;  
    }  
}
```



Consider the following program:

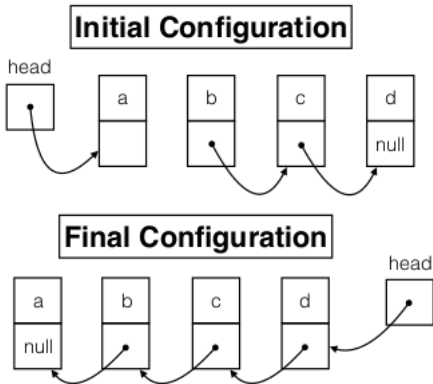
```
public class List {  
  Node head;  
  void mystery()  
  {  
    Node n = null;  
  
    while (head != null)  
    {  
      Node k = head.next;  
      head.next = n;  
      n = head;  
      head = k;  
    }  
    head = n;  
  }  
}
```



**In-place list reversal**

Consider the following program:

```
public class List {  
    Node head;  
    void mystery()  
    {  
        Node n = null;  
  
        while (head != null)  
        {  
            Node k = head.next;  
            head.next = n;  
            n = head;  
            head = k;  
        }  
        head = n;  
    }  
}
```



How to prove this program?

## Specification:

$$\{ \text{isList } \alpha \text{ head} \} \text{ reverse } () \{ \text{isList } \alpha^\dagger \text{ head} \}$$

where  $\alpha^\dagger$  denotes the reverse sequence of  $\alpha$ .

```

{ isList  $\alpha$  head }
{ isList  $\alpha$  head  $\star$  (emp  $\wedge$  null = null) }
Node n = null;
{ isList  $\alpha$  head  $\star$  (emp  $\wedge$  n = null) }
{ isList  $\alpha$  head  $\star$  isList [ ] n }
//loop invariant
{  $\exists \sigma, \tau :: \text{isList } \sigma \text{ head} \star \text{isList } \tau \text{ n} \star \alpha = \tau^\dagger \uplus \sigma$  }
while (head != null)
{
    Node k = head.next;
    head.next = n;
    n = head;
    head = k;
}
head = n;
}
    
```

# Lists in Separation Logic

```
Node n = null;
while (head != null)
{
    //loop invariant
    {  $\exists \sigma, \tau :: \text{isList } \sigma \text{ head} \star \text{isList } \tau \text{ n} \star \alpha = \tau^\dagger \dashv\vdash \sigma$  }
    {  $\exists \sigma, \tau :: \text{isList } (a :: \sigma) \text{ head} \star \text{isList } \tau \text{ n} \star \alpha = \tau^\dagger \dashv\vdash (a :: \sigma)$  }
    {  $\exists \sigma, \tau, k :: \text{head} \rightsquigarrow a, k \star \text{isList } \sigma \text{ k} \star \text{isList } \tau \text{ n} \star \alpha = \tau^\dagger \dashv\vdash (a :: \sigma)$  }
    Node k = head.next;
    {  $\exists \sigma, \tau :: \text{head} \rightsquigarrow a, k \star \text{isList } \sigma \text{ k} \star \text{isList } \tau \text{ n} \star \alpha = \tau^\dagger \dashv\vdash (a :: \sigma)$  }
    head.next = n;
    {  $\exists \sigma, \tau :: \text{head} \rightsquigarrow a, n \star \text{isList } \sigma \text{ k} \star \text{isList } \tau \text{ n} \star \alpha = \tau^\dagger \dashv\vdash (a :: \sigma)$  }
    {  $\exists \sigma, \tau :: \text{isList } (a :: \tau) \text{ head} \star \text{isList } \sigma \text{ k} \star \alpha = \tau^\dagger \dashv\vdash (a :: \sigma)$  }
    {  $\exists \sigma, \tau :: \text{isList } (a :: \tau) \text{ head} \star \text{isList } \sigma \text{ k} \star \alpha = (a :: \tau)^\dagger \dashv\vdash \sigma$  }
    {  $\exists \sigma, \tau :: \text{isList } \sigma \text{ k} \star \text{isList } \tau \text{ head} \star \star \alpha = \tau^\dagger \dashv\vdash \sigma$  }
    n = head;
    head = k;
    //loop invariant
    {  $\exists \sigma, \tau :: \text{isList } \sigma \text{ head} \star \text{isList } \tau \text{ n} \star \alpha = \tau^\dagger \dashv\vdash \sigma$  }
}
head = n;
```

```

{ isList  $\alpha$  head }
Node n = null;
while (head != null)
{
    Node k = head.next;
    head.next = n;
    n = head;
    head = k;
}
{  $\exists \tau :: \text{isList } [ ] \text{ head} \star \text{isList } \tau \text{ n} \star \alpha = \tau^\dagger ++ [ ] \}$ 
{  $\exists \tau :: \text{isList } [ ] \text{ head} \star \text{isList } \tau \text{ n} \star \alpha = \tau^\dagger \}$ 
{  $\exists \tau :: \text{isList } \tau \text{ n} \star \alpha = \tau^\dagger \}$ 
{  $\exists \tau :: \text{isList } \tau \text{ n} \star \alpha^\dagger = \tau^{\dagger\dagger} \}$ 
{  $\exists \tau :: \text{isList } \tau \text{ n} \star \alpha^\dagger = \tau \}$ 
{ isList  $\alpha^\dagger$  n }
head = n;
{ isList  $\alpha^\dagger$  head }
}

```

# List Reversal, in VeriFast

```
/*@
predicate Node(Node n; Node nn, int v) =
    n.next |-> nn &*& n.val |-> v;

predicate List(Node n; list<int> elems) =
    n == null? (emp &*& elems == nil) :
    Node(n,?nn,?v) &*& List(nn,?tail) &*& elems == cons(v,tail);

predicate ListInv(List l; list<int> elems) =
    l.head |-> ?h &*& List(h,elems);
@*/

class Node {
    Node next;
    int val;
    Node(int v, Node next)
    //@requires true;
    //@ensures Node(this,next,v);
    {
        this.next = next;
        val = v;
    }
}

class List {
    Node head;
    public List()
    //@ requires true;
    //@ ensures ListInv(this,nil);
    { head = null; }
    void add(int elem)
    //@ requires ListInv(this,?l);
    //@ ensures ListInv(this,cons(elem,l));
    {
        Node next = new Node(elem,head);
        head = next;
    }
}
```

# List Reversal, in VeriFast

```
/*@
predicate Node(Node n; Node nn, int v) = n.next |-> nn &*& n.val |-> v;

predicate List(Node n; list<int> elems) =
  n == null? (emp &*& elems == nil) :
  Node(n,?nn,?v) &*& List(nn,?tail) &*& elems == cons(v,tail);

predicate ListInv(List l; list<int> elems) = l.head |-> ?h &*& List(h,elems); @*/

void reverseList()
/*@ requires ListInv(this,?l);
   @ ensures ListInv(this,reverse(l));
{
  Node n = null;
  //@open ListInv(this,l);
  while (head != null)
    //@ invariant head |-> ?h &*& List(h,?l1) &*& List(n,?l2) &*& l == append(reverse
      (l2),l1);
  {
    Node k = head.next;
    head.next = n;
    n = head;
    head = k;
    //@assert l1 == cons(?v,?tail0) &*& l == append(reverse(l2),cons(v,tail0));
    //@reverse_reverse(cons(v,tail0));
    //@reverse_append( reverse(cons(v,tail0)) , l2 );
    //@append_assoc(reverse(tail0),cons(v,nil),l2);
    //@reverse_append(reverse(tail0),cons(v,l2));
    //@reverse_reverse(tail0);
  }
  //@open List(h,l1);
  head = n;
  //@append_nil(reverse(l2));
}
```

It **works** like a **charm**!

All **lemmas** used are part of the VeriFast **standard library**:

```
lemma_auto void append_nil<t>(list<t> xs);  
  requires true;  
  ensures append(xs, nil) == xs;
```

```
lemma void append_assoc<t>(list<t> xs, list<t> ys, list<t> zs);  
  requires true;  
  ensures append(append(xs, ys), zs) == append(xs, append(ys, zs));
```

```
lemma void reverse_append<t>(list<t> xs, list<t> ys);  
  requires true;  
  ensures reverse(append(xs, ys)) == append(reverse(ys), reverse(xs));
```

```
lemma_auto void reverse_reverse<t>(list<t> xs);  
  requires true;  
  ensures reverse(reverse(xs)) == xs;
```

Consider the following method:

```
int lengthImp (Node p)
{
    Node f = p;

    int t = 0;

    while (f != null)
    {
        f = f.next;
        t = t + 1;
    }

    return t;
}
```

Exercise:

1. Specify the state before the loop.
2. Specify the state after the loop.
3. State a loop invariant.

A picture can certainly help!

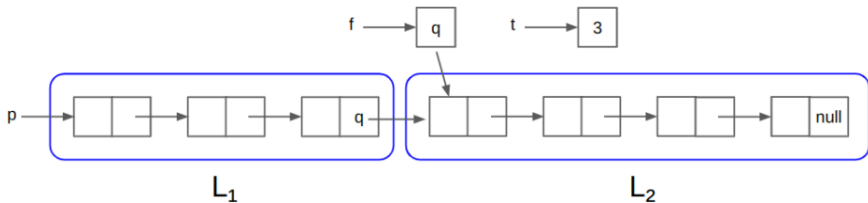
Before the loop:

$$\text{isList}(p, L) \star f \rightsquigarrow p \star t \rightsquigarrow 0$$

After the loop:

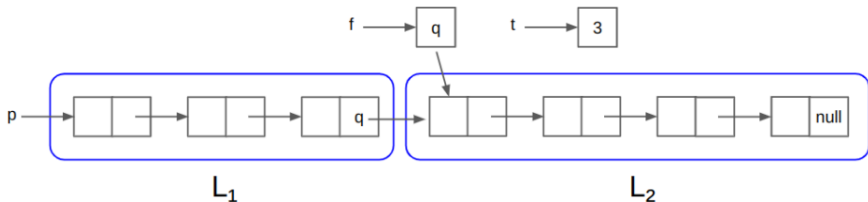
$$\text{isList}(p, L) \star f \rightsquigarrow \text{null} \star t \rightsquigarrow \text{length } L$$

## lengthImp – Loop Invariant



Loop invariant:

## lengthImp – Loop Invariant



Loop invariant:

$$\begin{aligned} \exists L_1 L_2 q \bullet & \quad L = L_1 \upharpoonright L_2 \star t \rightsquigarrow \text{length } L_1 \star f \rightsquigarrow q \\ & \star \text{LSeg}(p, q) \star \text{isList}(q, L_2) \end{aligned}$$

# Representation Predicates for List Segments

$$\text{isList}(\mathbf{L}, p) \triangleq \begin{cases} \text{emp} \wedge \mathbf{L} = [] & \text{if } p = \text{null} \\ \exists x, \mathbf{L}', p' \bullet p.\text{val} \rightsquigarrow x \star p.\text{next} \rightsquigarrow p' \\ \quad \star \text{isList}(\mathbf{L}', p') \star \mathbf{L} = x :: \mathbf{L}' & \text{if } p \neq \text{null} \end{cases}$$

**List segments:** generalize `isList` to define `LSeg(p, q, L)` where

- `L` denotes the list of items
- in the **list segment** from `p` (**inclusive**) to `q` (**exclusive**).

$$\text{LSeg}(p, q, \mathbf{L}) \triangleq \begin{cases} \text{emp} \wedge \mathbf{L} = [] & \text{if } p = q \\ \exists x, \mathbf{L}', p' \bullet p \neq \text{null} \\ \quad \star p.\text{val} \rightsquigarrow x \star p.\text{next} \rightsquigarrow p' \star \text{LSeg}(p', q, \mathbf{L}') \\ \quad \star \mathbf{L} = x :: \mathbf{L}' & \text{if } p \neq q \end{cases}$$

```
int lengthImp (Node p)
/*@ requires ListInv(this, ?l);
    ensures ListInv(this, l) &*& result == length(l);
{
    Node it = p;
    //@ Node p = it;

    int size = 0;

    while (it != null)
        /*@ invariant lseg(p, it, ?vs1) &*& lseg(it, null, ?vs2) &*&
            l == append(vs1, vs2) &*& size == length(vs1); @*/
    {
        it = it.next;

        size = size + 1;
    }

    return size;
}
```

## lengthImp – Proof (Auxiliary Steps)

```
int lengthImp (Node p)
/*@ requires ListInv(this, ?l);
  @ ensures ListInv(this, l) &*& result == length(l);
{
    Node it = p;
    @ Node p = it;

    int size = 0;

    while (it != null)
    /*@ invariant lseg(p, it, ?vs1) &*& lseg(it, null, ?vs2) &*&
      l == append(vs1, vs2) &*& size == length(vs1); @*/
    {
        @ Node old_it = it;
        it = it.next;
        @ open lseg (it, null, _);
        size = size + 1;
        @ lseg_merge(p, old_it, it);
        @ append_assoc (vs1 , cons(head(vs2), nil), tail(vs2));
    }

    @ open lseg ( it , it , _ );
    return size;
}
```

## LengthImp – Auxiliary Lemma

```
lemma void lseg_merge (Node x, Node y, Node w)
  requires lseg(x, y, ?vs1) &*& lseg(y, w, ?vs2)
         &*& lseg(w, null, ?vs3);
  ensures lseg(x, w, append(vs1, vs2))
         &*& lseg(w, null, vs3);
{
  open lseg(w, null, vs3);
  open lseg(x, y, vs1);
  if (x != y) {
    lseg_merge(x.next, y, w);
  }
}
```

Separation Logic is useful to **specify** and **verify** pointer-based data structures. For instance:

- Binary trees
- Doubly-linked lists
- Chunked sequences
- Queues

Pointer-based algorithms:

- Tree traversals
- Graph algorithms
- In-place DFS without additional memory: **Schorr-Waite**

# Trees in Separation Logic

---

# Binary Search Trees in Separation Logic

```
public class Tree {
    public int value;
    public Tree left, right;
    public void add(int x)
        /*@ requires tree(this,?b) &*& b!=tnil &*& false==t_contains(b,x) &*& inorder(b)
            ==true;
        /*@ ensures tree(this,tree_add(b,x)) &*& inorder(tree_add(b,x))==true;
    { /*@ open tree(this,b);
        int v=this.value;
        Tree l=this.left;
        /*@ open tree(l,?bl); close tree(l,bl);
        Tree r=this.right;
        /*@ open tree(r,?br); close tree(r,br);
        if(x < v) {
            if(l!=null){ l.add(x);
                /*@ tree_add_inorder(b,x);
                /*@ close tree(this,tcons(v,tree_add(bl,x),br)); }
            else {
                Tree temp=new Tree(x);
                this.left=temp;
                /*@ open tree(l,bl);
                /*@ close tree(this,tcons(v,tcons(x,tnil,tnil),br));
                /*@ tree_add_inorder(b,x);
            }
        } else {
            if(v < x) {
                if(r!=null) { r.add(x);
                    /*@ tree_add_inorder(b,x);
                    /*@ close tree(this,tcons(v,bl,tree_add(br,x))); }
                else {
                    Tree temp=new Tree(x);
                    this.right=temp;
                    /*@ open tree(r,br); close tree(this,tcons(v,bl,tcons(x,tnil,tnil)));
                }
            }
        }
    }
```

In-place DFS (Schorr-Waite graph marking):

- standard garbage collection algorithm
- mark reachable nodes using only one extra bit per node
- unmarked nodes can be collected and re-used

How to find all reachable nodes in a graph? DFS or BFS.

Depth-first Search:

- **Recursive:** requires space proportional to the size of the graph
- **Iterative:** requires an explicit stack.

# Schorr-Waite in Separation Logic

Schorr-Waite graph marking manipulates the pointers in the graph so that the **stack of nodes** is **encoded** in the **graph itself**.

```
/*@
  predicate Tree(Node t, boolean m) =
    t == null ? true : t.marked |-> m &*& t.child |-> ?c &*&
    t.left |-> ?l &*& t.right |-> ?r &*& Tree(l, m) &*& Tree(r
      ,m);

  predicate Stack(Node t) =
    t == null ? true : t.marked |-> true &*& t.child |-> ?c
      &*&
      t.left |-> ?l &*& t.right |-> ?r &*&
      (c == false ? Stack(l) &*& Tree(r,
        false) : Stack(r) &*& Tree(l,true))
      ;
  @*/

class Node {
  boolean marked;
  boolean child; //false: L , true: R
  Node left;
  Node right;
```

# Schorr-Waite in Separation Logic

```
void SchorrWaite()
/*@ requires Tree(this, false);
   @ ensures Tree(_, true);
{
    Node t = this;
    Node p = null;
    //@close Stack(p);
    //@open Tree(this,false);
    while (p != null || (t!=null && !(t.marked)))
    /*@ invariant (t == null ? true : t.marked |-> ?m &*& t.
        child |-> ?c &*& t.left |-> ?l &*&
t.right |-> ?r &*& Tree(r,m) &*& Tree(l,m)) &*& Stack(p);
    @*/
    { if (t == null || t.marked) {
        //@open Stack(p);
        if (p.child) { //pop
            Node q = t;
            t = p;
            p = p.right;
            t.right = q;
            //@close Tree(q,true);
        }
    }
}
```

# Schorr-Waite in Separation Logic

```
    else { //swing
        Node q = t;
        t = p.right;
        p.right = p.left;
        p.left = q;
        p.child = true;
        //@close Tree(q,true);
        //@close Stack(p);
        //@open Tree(t,false);
    }
}
else { //push
    Node q = p;
    p = t;
    t = t.left;
    p.left = q;
    p.marked = true;
    p.child = false;
    //@open Tree(t,false);
    //@close Stack(p);
}
}
//@open Stack(p);
//@close Tree(t,true);
}
```

# Arrays in Separation Logic

---

The access to an array in Verifast is disciplined by predicates that describe segments of one position:

```
array_element(a, index, v)
```

to denote that the value ( $v$ ) is stored in position ( $index$ ) of the array value ( $a$ ).

More than one position:

```
array_slice(a, 0, n, vs)
```

to denote the access to the positions of array ( $a$ ) from ( $0$ ) to ( $n$ ) with values given by the specification-level list ( $vs$ ).

Signature:

```
array_slice<T>(T[] array, int start, int end; list<T> elements)
```

Properties about the elements of the array:

```
array_slice_deep(a, i, j, P, unit, vs, unit);
```

to denote that predicate ( $P$ ) is valid for all values ( $vs$ ) stored in array ( $a$ ), from indices ( $i$ ) to ( $j$ ).

Signature:

```
array_slice_deep<T, A, V>(T[], int, int,  
    predicate(A, T; V), A; list<T>, list<V>)
```

Predicate has signature:

```
predicate P<A, T, V>(A a, T v; V n);
```

Example of using `array_slice_deep` and a `predicate`:

```
predicate Positive(unit a, int v; unit n) =  
    v >= 0 &* & n == unit;  
array_slice_deep(s,0,n,Positive,unit,elems,_)
```

# Arrays in Separation Logic

```
fixpoint int sum(list<int> vs) {  
  switch(vs) {  
    case nil: return 0;  
    case cons(h, t): return h + sum(t);  
  }  
}  
  
public static int sum(int[] a)  
  //@ requires array_slice(a, 0, a.length, ?vs);  
  //@ ensures array_slice(a, 0, a.length, vs) &*& result == sum(  
    vs);  
{  
  int total = 0; int i = 0;  
  while(i < a.length)  
    /*@ invariant 0 <= i &*& i <= a.length  
      &*& array_slice(a, 0, a.length, vs)  
      &*& total == sum(take(i, vs)); @*/  
  {  
    int tmp = a[i]; total = total + tmp;  
    //@ length_drop(i, vs);  
    //@ take_one_more(vs, i);  
    i++;  
  }  
  return total;  
}
```

# Arrays in Separation Logic – Auxiliary Lemmas

```
lemma void take_one_more<t>(list<t> vs, int i)
  requires 0 <= i && i < length(vs);
  ensures append(take(i, vs), cons(head(drop(i, vs)), nil)) ==
    take(i + 1, vs);
{
  switch(vs) {
    case nil:
    case cons(h, t):
      if(i == 0) { }
      else {
        take_one_more(t, i - 1);
      }
  }
}
```

```
lemma_auto(sum(append(xs, ys))) void sum_append(list<int> xs,
  list<int> ys)
  requires true;
  ensures sum(append(xs, ys)) == sum(xs) + sum(ys);
{
  switch(xs) {
    case nil:
    case cons(h, t): sum_append(t, ys);
  }
}
```

Bag ADT using an array with limited capacity:

```
public class Bag {  
    int store[];  
    int nelems;  
  
    int get(int i) { ... }  
  
    int size() { ... }  
  
    boolean add(int v) { ... }  
}
```

Array access is disciplined by the predicate `array_slice`

```
int get(int i)
  //@ requires this.store |-> ?s &* & array_slice(s,0,?n,_) &* &
           0 <= i &* & i < n;
  //@ ensures ... ;
{
  return store[i];
}
```

The representation invariant captures the legal states of the ADT, including the access to the array.

```
public class Bag {  
    int store[];  
    int nelems;  
    ...  
}  
  
/*@  
    predicate BagInv(int n) =  
        store |-> ?s  
        &*& nelems |-> n  
        &*& s != null  
        &*& 0<=n &*& n <= s.length  
        &*& array_slice(s,0,n,?elems)  
        &*& array_slice(s,n,s.length,?others);  
    @*/
```

Back to method get:

```
int get(int i)
    //@ requires BagInv(?n) &*& 0 <= i &*& i < n;
    //@ ensures BagInv(n);
{
    return store[i];
}
```

The same strategy for all methods:

```
int size()
    //@ requires BagInv(?n) &*& n >= 0;
    //@ ensures BagInv(n) &*& result >= 0;
{ return nelems; }

public Bag (int size)
    //@ requires size >= 0;
    //@ ensures BagInv(0);
{
    store = new int[size];
    nelems = 0;
}

boolean add (int v)
    //@ requires BagInv(_);
    //@ requires BagInv(_);
{
    if(nelems<store.length) {
        store[nelems] = v;
        nelems = nelems+1;
        return true;
    } else { return false; }
}
```

Lets focus on method add:

```
boolean add (int v)
    //@ requires BagInv(?n);
    //@ requires BagInv(n+1); // DOES NOT HOLD!!!!
{
    if(nelems<store.length) {
        store[nelems] = v;
        nelems = nelems+1;
        return true;
    } else { return false; }
}
```

Use the parameters of the representation predicate as a definition of **abstract states**.

```
boolean add (int v)
  //@ requires BagInv(?n);
  //@ requires result ? BagInv(n+1) : BagInv(n); // CHECKS! :D
{
  if(nelems<store.length) {
    store[nelems] = v;
    nelems = nelems+1;
    //@ close BagInv(n+1);
    return true;
  } else {
    //@ close BagInv(n);
    return false;
  }
}
```

## Arrays in Verifast – Recap

The **physical** access (*i.e.*, the pointer) to the array in Verifast is disciplined by predicates that describe **segments** of the array:

```
array_element(a, index, v);  
array_slice(a, 0, n, vs);  
array_slice_deep(a, i, j, P, unit, vs, unit);
```

The **abstract** access (*i.e.*, the values) to the array is disciplined by **specification level lists** and associated operations:

```
drop(n, vs);  
take(n, vs);  
append(vs, vs');
```

Properties of values stored in arrays are captured by the array predicates.

### Examples:

- Bag of positive integers
- Array of ADT objects

Next: a Bank using an array of Accounts.

```
/*@  
predicate AccountInv(Account a;int b) =  
    a.balance |-> b &*& b >= 0;  
@*/  
  
public class Account {  
    int balance;  
    public Account()  
        //@ requires true;  
        //@ ensures AccountInv(this,0);  
    {  
        balance = 0;  
    }  
    ...  
}
```

Hence, the balance of each Account is always positive.

A Bank holds an array of Accounts:

```
public class Bank {  
    Account store[];  
    int nelems;  
    int capacity;  
    Bank(int max)  
    {  
        nelems = 0;  
        capacity = max;  
        store = new Account[max];  
    }  
    ...  
}
```

A couple of operations:

```
public class Bank {  
    Account store[];  
    int nelems;  
    int capacity;  
  
    ...  
    Account retrieveAccount()  
    {  
        Account c = store[nelems-1];  
        store[nelems-1] = null;  
        nelems = nelems-1;  
        return c;  
    }  
}
```

A couple of operations:

```
public class Bank {  
    Account store[];  
    int nelems;  
    int capacity;  
  
    ...  
    Account addNewAccount()  
    {  
        Account c = new Account();  
        store[nelems] = c;  
        nelems = nelems + 1;  
    }  
}
```

# Bank in Separation Logic – Representation Predicate

```
/*@  
predicate AccountP(unit a, Account c; unit b) =  
    AccountInv(c, ?n) &* & b == unit;  
@*/  
  
public class Bank {  
  
    /*@  
    predicate BankInv(int n, int m) =  
        this.nelems |-> n  
        &* & this.capacity |-> m  
        &* & m > 0  
        &* & this.store |-> ?accounts  
        &* & accounts.length == m  
        &* & 0 <= n &* & n <= m  
        &* & array_slice_deep(accounts, 0, n, AccountP, unit, _  
            , _)  
        &* & array_slice(accounts, n, m, ?rest) &* & all_eq(rest,  
            null) == true;  
    @*/  
}
```

## Bank in Separation Logic – Specification

A Bank holds an array of Accounts:

```
public class Bank {  
    Account store[];  
    int nelems;  
    int capacity;  
  
    Bank(int max)  
        //@ requires max > 0;  
        //@ ensures BankInv(0, max);  
    {  
        nelems = 0;  
        capacity = max;  
        store = new Account[max];  
    }  
    ...  
}
```

## Bank in Separation Logic – Specification

```
public class Bank {  
    Account store[];  
    int nelems;  
    int capacity;  
  
    ...  
    Account retrieveAccount()  
    //@ requires BankInv(?n, ?m) &*& n > 0;  
    //@ ensures BankInv(n-1, m) &*& AccountInv(  
        result, _);  
    {  
        Account c = store[nelems-1];  
        store[nelems-1] = null;  
        nelems = nelems-1;  
        return c;  
    }  
}
```

## Bank in Separation Logic – Specification

```
public class Bank {  
    Account store[];  
    int nelems;  
    int capacity;  
  
    ...  
    Account addNewAccount()  
    //@ requires BankInv(?n, ?m) &* & n < m;  
    //@ ensures BankInv(n+1, m);  
    {  
        Account c = new Account();  
        store[nelems] = c;  
        nelems = nelems + 1;  
    }  
}
```