

Construction and Verification of Software

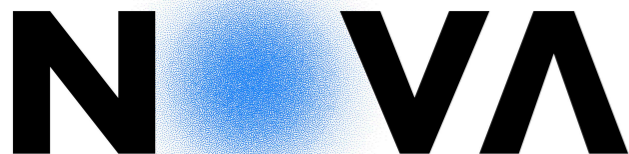
2021 - 2022

MIEI - Integrated Master in Computer Science and Informatics
Consolidation block

Lecture 8 - Arrays in Separation Logic

Bernardo Toninho (htoninho@fct.unl.pt)

based on previous editions by **João Seco** and **Luís Caires**



Outline

- Recap on Separation Logic
- Arrays in Separation Logic
- Basic Algorithms
- The Bag ADT
- Properties of elements in arrays (Bank)

Part I

Separation Logic (recap)

Separation Logic (recap)

Separation logic assertions used in our CVS course are described by the following grammar:

$A ::=$		Separation Logic Assertions
	$L \mapsto V$	Memory Access
	$ A * A$	Separating Conjunction
	$ \text{emp}$	Empty heap
	$ B$	Boolean condition (pure, not spatial)
	$ 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

Separation Logic (recap)

- The assignment rule in separation logic is

$$\{x \mapsto V\} x := E \{x \mapsto E\}$$

- Follows the small footprint principle, the precondition refers exactly to the part of the memory used by the fragment!

Separation Logic (recap)

- The assignment rule in separation logic is

$$\{x.l \mapsto V\} x.l := E \{x.l \mapsto E\}$$

- Follows the small footprint principle, the precondition refers exactly to the part of the memory used by the fragment!
- The memory locations here are fields of objects. We also have to model dynamically allocated arrays.

Example of Separation Logic in Verifast

```
/*@  
  predicate Node(Node t; Node n, int v) = t.nxt l-> n &*& t.val l-> v;
```

```
  predicate List(Node n;) = n == null ? emp : Node(n, ?nn, _) &*& List(nn);
```

```
  predicate StackInv(Stack t;) = t.head l-> ?h &*& List(h);
```

```
@*/
```

```
class Stack {  
  private Node head;  
  
  public Stack()  
  //@ requires true;  
  //@ ensures StackInv(this);  
  { head = null; }  
  
  public int pop()  
  //@ requires NonEmptyStackInv(this);  
  //@ ensures StackInv(this);  
  { int val = head.getValue(); head = head.getNext(); return val; }  
  
  public boolean isEmpty()  
  //@ requires StackInv(this);  
  //@ ensures result?StackInv(this):NonEmptyStackInv(this);  
  { return head == null; }  
}
```

...

```
public static void main(String args[])  
  //@ requires true;  
  //@ ensures true;  
  {  
    Stack s = new Stack();  
    s.push(0);  
    s.pop();  
    if( ! s.isEmpty() ) {  
      //@ open NonEmptyStackInv(_);  
      s.push(1);  
      s.pop();  
    }  
  }
```

Part II

Arrays in Separation Logic

Arrays in Separation Logic

- Arrays are dynamically allocated regions that need to be explicitly described by specially designed predicates.

```
public static int sum(int[] a)
  //@ requires ???
  //@ ensures ???
  {
    int total = 0; int i = 0;
    while(i < a.length)
      //@ invariant ???
      {
        int tmp = a[i]; total = total + tmp;
        i++;
      }
    return total;
  }
```

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**).

- or 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**).

Arrays in Separation Logic

- 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 the array (**a**) in the positions from (**i**) to (**j**).

- Signature:

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

- Where the predicate has the signature:

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

Arrays in Separation Logic

- 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 the array (**a**) in the positions from (**i**) to (**j**).

- Where the predicate has the signature:

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

```
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);
  }
}
```

- With auxiliary functions and definitions that define properties over the values of arrays.

```
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

```
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);
      }
  }
}
```

And auxiliary lemmas that can be applied in the course of the proof.

```
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);
  }
}
```

Part III

Managing Arrays in Objects

Verifast Example

- Consider a Bag of integers ADT based on an array with limited capacity.

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

•

Verifast Example - Bag

- Fields must be considered in separate heap chunks, pure conditions can be added to assertions and predicates.
- 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];
}
```

Verifast Example - Bag

- 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)  
        ;  
    @*/  
    ...  
}
```

Verifast Example - Bag

- So...

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

Verifast Example - Bag

- For all methods and fields...

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

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

Verifast Example - Bag

- For all methods and fields...

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

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

Verifast Example - Bag

- For all methods and fields...

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

boolean add(int v)
  //@ requires BagInv(?n);
  //@ ensures BagInv(n+1); // Does not hold, why?
{
  if(nelems < store.length) {
    store[nelems] = v;
    nelems = nelems+1;
    return true;
  } else {
    return false;
  }
}
```

Verifast Example - Bag

- For

```
public
//@
//@
{
store
nele
}
boolean
//@
//@
{
if(n
sto
nel
ret
} el
ret
}
}
```

The screenshot displays the Verifast IDE interface. At the top, a red error banner reads "Cannot prove dummy == (dummy + 1)". The main editor shows the following Java code:

```
predicate BagInv(int n) =
  store |-> ?s
  && nelems |-> n
  && s != null
  && 0 <= n && n <= s.length

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

int get(int i)
```

The local variable table on the right shows the following state:

Local	Value
n	(dummy + 1)
s	s
this	this

The debugger window at the bottom shows the following state:

- Steps: Executing statement, Executing second branch, Executing statement, Executing statement, Consuming assertion, Consuming assertion.
- Assumptions: !(this = 0), !(this = 0), !(s = 0), 0 <= dummy, dummy <= arraylength(s), !(s = 0).
- Heap chunks: Bag_nelems(this, dummy), java.lang.array_slice<int32>(s, 0, dum), java.lang.array_slice<int32>(s, dummy).

Verifast Example - Bag

- The parameters for the representation invariant predicate are specification level and define an abstract state.

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


Verifast Example - Bag

- The access to an 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 values of the array are accessed through specification level list values and related operations

```
drop( n, vs )
```

```
take( n, vs )
```

```
append( vs, vs' )
```

Part IV
An example of an ADT
(Bank)

Managing arrays in Separation Logic

- Properties of values stored in arrays can also be captured by the array predicates.
- Examples:
 - Bag of positive integers
 - Array of ADT objects (with representation invariants)

Managing arrays in Separation Logic

```
/*@  
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;  
    }  
    ...  
}
```

Managing arrays in Separation Logic

- The 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];  
    }  
    ...  
}
```

Managing arrays in Separation Logic

- And implements 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;  
    }  
    ...  
}
```

Managing arrays in SL

- And implements a couple of operations...

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

Managing arrays in SL

```
/*@
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;
@*/

}
```


array slice assertions

The predicate declared in file `java.lang.javaspec` by

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

represents the footprint of the array fragment

`array[start .. end-1]`

- `elements` is the list of array “values” v_i such that $a[i] \mapsto v_i$
- `elements` is an immutable pure value (like an OCaml list)

- `array_slice(array, start, end, elements)`

is equivalent to the assertion

- $v = [v_{start}, \dots, v_{end-1}]$

- $a[start] \mapsto v_{start}$

- $\&*\& a[start+1] \mapsto v_{start+1} \&*\& \dots \&*\& a[end-1] \mapsto v_{end-1}$

array slice assertions

The predicate declared in file `java.lang.javaspec` by

```
predicate array_slice_deep<T, A, V>(
    T[] array,
    int start,
    int end,
    predicate(A, T; V) p,
    A info;
    list<T> elements,
    list<V> values);
```

is as in the (simple) `array_slice` where `elements` is the list of array values v_i such that $a[i] \mapsto v_i$, and the predicate $p(a, v_i; o_i)$ holds for each v_i and `values` is the list of all values o_i

Managing arrays in SL

```
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];  
    }  
    ...  
}
```

Managing arrays in SL

```
public class Bank {  
  
    Account store[];  
    int nelems;  
    int capacity;  
  
    Account retrieveLastAccount()  
    //@ requires BankInv(?n,?m) &*& n>0;  
    //@ ensures  BankInv(n-1,m) &*& AccountInv(result,_);  
    {  
        Account c = store[nelems-1];  
        nelems = nelems-1;  
        return c;  
        // This does not verify correctly, Why?  
    }  
}
```

Managing arrays in SL

```
public class Bank {  
  
    Account store[];  
    int nelems;  
    int capacity;  
  
    Account retrieveLastAccount()  
    //@ 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;  
    }  
}
```

Managing arrays in SL

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

array slice “lemmas”

```
lemma void array_slice_deep_close<T, A, V>(
    T[] array, int start, predicate(A, T; V) p, A a);
    requires array_slice<T>(array, start, start+1, ?elems) &*& p(a, head(elems), ?v);
    ensures array_slice_deep<T,A,V>(array, start, start+1, p, a, elems, cons(v,nil));
```

- transforms the spec of an array element in a (singleton) **array_slice** spec into a (singleton) **array_slice_deep**
- there are other lemmas, that join together slices
- Verifast is usually able to apply lemmas automatically, but not always, in that case the programmer needs to “help”, by calling the needed lemmas.

array slice “lemmas”

```
lemma void array_slice_split<T>(T[] array, int start, int start1);
```

requires

```
array_slice<T>(array, start, ?end, ?elems) &*&
```

```
start <= start1 &*& start1 <= end;
```

ensures

```
array_slice<T>(array, start, start1, take(start1 - start, elems)) &*&
```

```
array_slice<T>(array, start1, end, drop(start1 - start, elems)) &*&
```

```
elems == append(take(start1 - start, elems), drop(start1 - start, elems))
```

- this “lemma” splits one array slice assertion into two (sub) array slice assertions.

array slice “lemmas”

```
lemma void array_slice_join<T>(T[] array, int start);
```

```
  requires
```

```
    array_slice<T>(array, start, ?start1, ?elems1) &*&
```

```
    array_slice<T>(array, start1, ?end, ?elems2);
```

```
  ensures
```

```
    array_slice<T>(array, start, end, append(elems1, elems2));
```

- this “lemma” joins two array slice assertions into a single array slice assertion.

array slice “lemmas”

← → ↻ 🏠 Secure | <https://people.cs.kuleuven.be/~bart.jacobs/verifast/examples/rt/Object.javaspec.html>

```
package java.lang;

import java.util.*;

/*@

inductive unit = unit;

inductive pair<a, b> = pair(a, b);

fixpoint a fst<a, b>(pair<a, b> p) {
  switch (p) {
    case pair(x, y): return x;
  }
}

fixpoint b snd<a, b>(pair<a, b> p) {
  switch (p) {
    case pair(x, y): return y;
  }
}

fixpoint t default_value<t>();

inductive boxed_int = boxed_int(int);
fixpoint int unboxed_int(boxed_int i) { switch (i) { case boxed_int(value): return value; } }

inductive boxed_bool = boxed_bool(boolean);
fixpoint boolean unboxed_bool(boxed_bool b) { switch (b) { case boxed_bool(value): return value; } }

predicate array_element<T>(T[] array, int index; T value);
predicate array_slice<T>(T[] array, int start, int end; list<T> elements);
predicate array_slice_deep<T, A, V>(T[] array, int start, int end, predicate(A, T; V) p, A info; list<T> elements

lemma_auto void array_element_inv<T>();
  requires [?f]array_element<T>(?array, ?index, ?value);
  ensures [f]array_element<T>(array, index, value) &*& array != null &*& 0 <= index &*& index < array.length;
```