Construction and Verification of Software

2018 - 2019

MIEI - Integrated Master in Computer Science and Informatics
Consolidation block

Lecture 4 - Abstract State vs Representation State

João Costa Seco (joao.seco@fct.unl.pt)
based on previous editions by **Luís Caires** (lcaires@fct.unl.pt)

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Abstract Data Types

Set ADT

```
class ASet {
// an abstract Set of numbers

    method new(sz:int) {}
    // initializes aset ( e.g., Java constructor )

    method add(v:int) {}
    // adds v to aset if space available )

    function size() : int
    // returns number of elems in aset

    function contains(v:int) : bool
    // check if v belongs to set

    function maxsize() : int
    // returns max number of elems allowed in aset

}
```

Set ADT

- Abstract State
  - a set of positive integers aset

Set ADT

- Representation type
  - an array of integers **store** with sufficient large size
  - an integer nelems counting the elements in **store**

Set ADT

- Representation type
  - an array of distinct integers **store**
  - an integer nelems counting the elements in **store**
- Representation invariant

```
(store != null) &&

(0 <= nelems <= store.length) &&

forall k :: (0<=k<nelements) ==> forall j::(k<j<nelements)  ==> b[k] != b[j]
```

Set ADT

- Representation type
  - an array of **distinct** integers **store**
  - an integer nelems counting the elements in **store**
- Representation invariant

```
(store != null) &&

(0 <= nelems <= store.length) &&

forall k :: (0<=k<nelements) ==> forall j::(k<j<nelements)  ==> b[k] != b[j]
```

Set ADT

- Representation type
  - an array of distinct integers **store**
  - an integer nelems counting the elements in **store**
- Representation invariant

```
(store != null) &&

(0 <= nelems <= store.length) &&

forall k :: (0<=k<nelements) ==> forall j::(k<j<nelements)  ==> b[k] != b[j]
```

- Abstraction mapping
  - $\langle nelems=n, store=[v_0,v_1,\ldots v_{store.Length-1}]\rangle \rightarrow \{v_0,\ldots,v_{n-1}\}$
  - more later ....

Set ADT

```
class ASet {

  var a:array<int>;
  var size:int;

  constructor(SIZE:int)
    requires SIZE > 0;
    ensures RepInv()
  {
    a := new int[SIZE];
    size := 0;
  }

…
```

Set ADT

```
class ASet {

  var a:array<int>;
  var size:int;

  constructor(SIZE:int)
    requires SIZE > 0;
    ensures RepInv()
  {
    a := new int[SIZE];
    size := 0;
  }

  function RepInv():bool
    reads this,a;
  {
   …
  }
…
```

Set ADT

```
class ASet {

  var a:array<int>;
  var size:int;

…

  function RepInv():bool
    reads this,a;
  {
    a!=null &&
    0 < a.Length &&
    0 <= size <= a.Length &&
    unique(a,0, size)
  }
…
```

Set ADT

```
class ASet {

  var a:array<int>;
  var size:int;

…
  function unique(b:array<int>, l:int, h:int):bool
  reads b;
  requires b != null && 0<=l <= h <= b.Length ;
  {
    forall k::(l<=k<h) ==> forall j::(k<j<h)  ==> b[k] != b[j]
  }
…
```

Set ADT

```
class ASet {

  var a:array<int>;
  var size:int;

  function count():int
  reads this,a;
  requires RepInv();
  { size }

  function maxsize():int
  reads this,a;
  requires RepInv();
  { a.Length }

  method add(x:int)
  modifies this,a;
  requires RepInv() && x >= 0 && count() < maxsize();
  ensures RepInv()
  {
    var f:int := find(x);
    if (f < 0) {
      a[size] := x;
      size := size + 1;
    }
  }
…
```

Set ADT

```
class ASet {

  var a:array<int>;
  var size:int;
…
  method find(x:int) returns (r:int)
  requires RepInv();
  ensures -1 <= r < size;
  ensures r < 0 ==> forall j::(0<=j<size) ==> x != a[j];
  ensures r >=0 ==> a[r] == x;
  {
    var i:int := 0;
    while (i<size)
    decreases size-i
    invariant 0<=i<=size;
    invariant forall j::(0<=j<i) ==> x != a[j];
    {
      if (a[i]==x) { return i; }
      i := i + 1;
    }
    return -1;
  }
```

Set ADT

```
class ASet {

  var a:array<int>;
  var size:int;
…
  method contains(v:int) returns (f:bool)
  requires RepInv();
  ensures  f <==> exists j::(0<=j<size) && v == a[j];
  ensures RepInv();
  {
    var p:int := find(v);
    f := (p >= 0);
  }
}
```

Soundness and Abstraction Map

- We have learned how to express the representation invariant and make sure that no unsound states are ever reached
- We have informally argued that the representation state in every case represents the right abstract state, but how to make sure?
- We next see how the correspondence between the representation state and the abstract state can be explicitly expressed in Dafny using ghost variables, specification operations, and abstraction map soundness check.

Soundness between Abstract State & Representation state

Technical ingredients in ADT design

RECAP

- The ***abstract state***
  - defines how client code sees the object
- The ***representation type***
  - chosen by the programmer to implement the ADT internals. The programmer is free to chose the implementation strategy (data-structures, algorithms). This is done at construction time.
- The ***concrete state***
  - in general, not all representation states are legal concrete states
  - a concrete state is a representation state that really represents some well-defined abstract state

RECAP

Technical ingredients in ADT design

- The ***representation invariant***
  - the representation invariant is a condition that restricts the representation type to the set of (safe) concrete states
  - if the ADT representation falls outside the rep invariant, something is wrong (inconsistent representation state).
- The ***abstraction function***
  - maps every concrete state into some abstract state
- The ***operation pre- post- conditions***
  - expressed for the representation type
  - also expressed for the abstract type (for client code)

Soundness and Abstraction Map

- A so-called ghost variable is only used in the spec and does not actually use memory space
- Usages of ghost variables only occur in spec operations (are never executed at runtime)

```
class ASet {
    // Abstract state
    ghost var s:set<int>;

    // Representation state
    var a:array<int>;
    var size:int;
```

- We therefore represent the abstract state with a ***ghost variable***.

Sound

- A so-ca … d does not act…
- Usages … ns (are ne…

```
class ASe
    // Ab
    ghost

    // Re
    var a
    var s
```

- We the … ***t variable***.

Dafny - Sets

Research

other tutorials close

Sets

1. tutorials

Sets of various types form one of the core tools of verification for Dafny. Sets represent an orderless collection of elements, without repetition. Like sequences, sets are immutable value types. This allows them to be used easily in annotations, without involving the heap, as a set cannot be modified once it has been created. A set has the type:

```
set<int>
```

for a set of integers, for example. In general, sets can be of almost any type, including objects. Concrete sets can be specified by using display notation:

load in editor

```
var s1 := {}; // the empty set
var s2 := {1, 2, 3}; // set contains exactly 1, 2, and 3
assert s2 == {1,1,2,3,3,3,3}; // same as before
var s3, s4 := {1,2}, {1,4};
```

The set formed by the display is the expected set, containing just the elements

Soundness and Abstraction Map

- We next define a boolean function `Sound()` that specifies the precise relationship the abstract and concrete state:

```
// The mapping function between abstract and representation state
function Sound():bool
    reads this,a
    requires RepInv();
{
    forall x::(x in s) <==> exists p::(0<=p<size) && (a[p] == x)
}
```

- We then express in all operations how the abstract state changes, and how it is kept well related with a proper representation state
- As a benefit, we may then also express pre and post conditions in terms of the abstract state !

Set ADT (with abstract state)

```
class ASet {
   // Abstract state
   ghost var s:set<int>;
   // Representation state
   var a:array<int>;
   var size:int;

   // The mapping function between abstract and representation state
   function Sound():bool
       reads this,a
       requires RepInv();
   { forall x::(x in s) <==> exists p::(0<=p<size) && (a[p] == x) }

   function RepInv():bool
       reads this,a
   { 0 < a.Length && 0 <= size <= a.Length && unique(a,0,size) }

   function AbsInv():bool
       reads this,a
   { RepInv() && Sound() }

   // Spec functions
   function unique(b:array<int>, l:int, h:int):bool
       reads b;
       requires 0 <= l <= h <= b.Length ;
   { forall k::(l<=k<h) ==> forall j::(k<j<h)  ==> b[k] != b[j] }
```

Set ADT (with abstract state)

```
 class ASet {
    // Abstract state
    ghost var s:set<int>;

    // Representation state
    var a:array<int>;
    var size:int;
…
    // Implementation: Constructor and Methods

    constructor(SIZE:int)
        requires SIZE > 0;
        ensures AbsInv() && s == {};
    {
        // Init of Representation state
        a := new int[SIZE];
        size := 0;
        // Init of Abstract state
        s := {};
    }
…
```

Set ADT (with abstract state)

```
 class ASet {
    // Abstract state
    ghost var s:set<int>;

    // Representation state
    var a:array<int>;
    var size:int;
…
    method find(x:int) returns (r:int)
        requires AbsInv()
        ensures  AbsInv()
        ensures -1 <= r < size;
        ensures r < 0 ==> forall j::(0<=j<size) ==> x != a[j];
        ensures r >=0 ==> a[r] == x;
    {
        var i:int := 0;
        while (i<size)
            decreases size-i
            invariant 0 <= i <= size;
            invariant forall j::(0<=j<i) ==> x != a[j];
        {
            if (a[i]==x) { return i; }
            i := i + 1;
        }
        return -1;
    }
…
```

Set ADT (with abstract state)

```
class ASet {
   // Abstract state
   ghost var s:set<int>;

   // Representation state
   var a:array<int>;
   var size:int;
…
   method add(x:int)
       modifies a, this
       requires AbsInv()
       requires count() < maxsize()
       ensures  AbsInv() && s == old(s) + {x}
   {
       var i := find(x);
       if (i < 0) {
           a[size] := x;
           s := s + { x };
           size := size + 1;
           assert a[size-1] == x;
           assert forall i :: (0<=i<size-1) ==> (a[i] == old(a[i]));
           assert forall x::(x in s) <==> exists p::(0<=p<size) && (a[p] == x);
       }
   }
…
```

Loop Invariants Recap & Sorting

ADT Specifications

- Method contracts, expressed as assertions

```
method P(... parameters ...)
  requires pre-condition-assertion
  ensures post-condition-assertion
  modifies global-state-changed
  {
     ... method code
  }
```

- Abstract State Invariants (visible by the ADT clients)
- Representation State Invariants (the implementation type)
- Abstract Mapping between the two (soundness of ADT)

ADT Specifications

- Method contracts, expressed as assertions

```
class PSet {
 …
   method add(x:int)
      modifies this,a;
      requires RepInv() && count() < maxsize();
      ensures RepInv()
   { … }
 …
```

- Representation State Invariants (the implementation type)
- Abstract State Invariants (visible by the ADT clients)
- Abstract Mapping between the two (soundness of ADT)

ADT Specifications

- Method contracts, expressed as assertions
- Representation State Invariants (the implementation type)

```
var a:array<int>;

 var size:int;

function RepInv():bool
    reads this,a
{
    0 < a.Length &&
    0 <= size <= a.Length &&
    unique(a,0,size) &&
    forall p :: (0 <= p < size) ==> 0 <= a[p]
}
```

- Abstract State Invariants (visible by the ADT clients)
- Abstract Mapping between the two (soundness of ADT)

ADT Specifications

- Method contracts, expressed as assertions
- Representation State Invariants (the implementation type)
- Abstract State Invariants (visible by the ADT clients)

```
var s:set<int>;

function AbsInv():bool
    reads this,a
{ forall x :: (x in s ) ==> 0 <= x }

method add(x:int)
    modifies a, this
    requires AbsInv() && count() < maxsize() && 0 <= x
    ensures  AbsInv() && s == old(s) + {x}
{ …
```

- Abstract Mapping between the two state representations (soundness of ADT specification)

ADT Specifications

- Method contracts, expressed as assertions
- Representation State Invariants (the implementation type)
- Abstract State Invariants (visible by the ADT clients)
- Abstract Mapping between the two (soundness of ADT)

```
function Sound():bool
    reads this,a
    requires RepInv();
{
    forall x::(x in s) <==> exists p::(0<=p<size) && (a[p] == x)
}

function AbsInv():bool
    reads this,a
{
    forall x :: (x in s ) ==> 0 <= x
    && RepInv() && Sound()
}
```

Loop Invariants

- Loop invariant approximate state assertions before the loop, between iterations, and in the end of the loop.

```
function maxArray(a:array<int>,n:int,m:int):bool
    requires 0 < n <= a.Length
    reads a
{    forall k : int :: 0 <= k < n ==> a[k] <= m }

method Max(a:array<int>) returns (m:int)
    requires 0 < a.Length
    ensures maxArray(a,a.Length,m)
{
    m := a[0];
    var i := 1;
    while i < a.Length
        invariant 1 <= i <= a.Length
        invariant maxArray(a,i,m)
    {
        if m < a[i]
        { m := a[i]; }
        i := i + 1;
    }
}
```

Example: BinarySearch

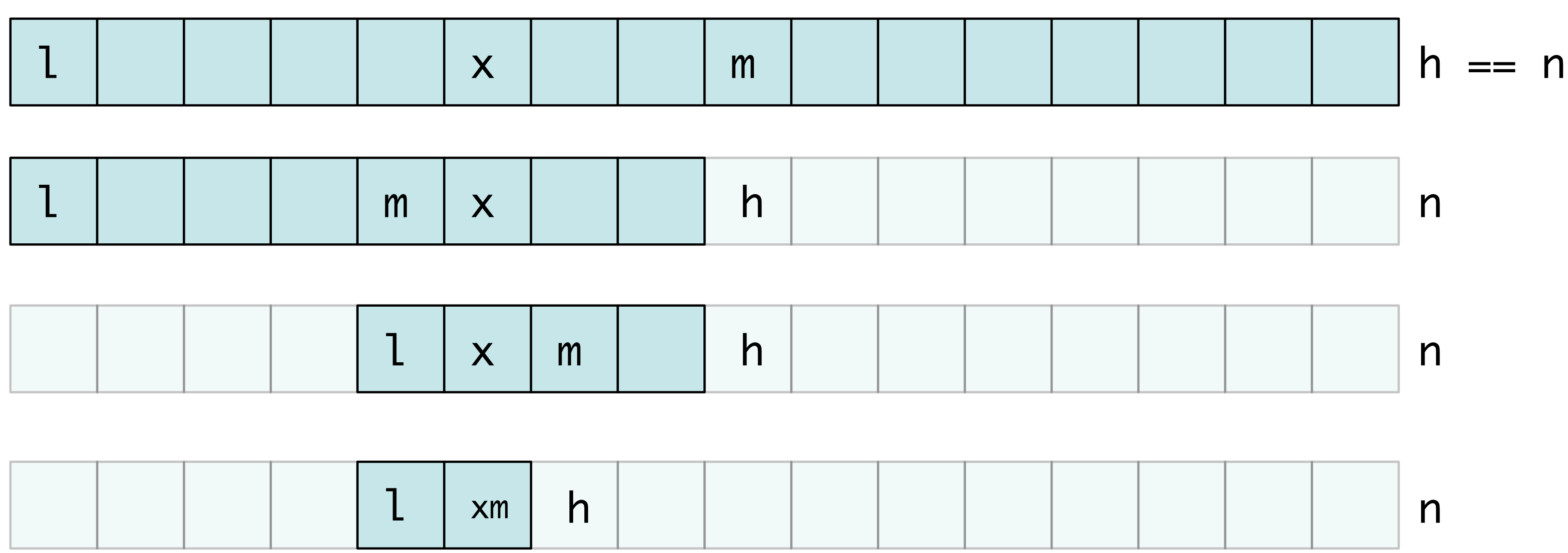
```
function sorted(a:array<char>, n:int):bool
    requires 0 <= n <= a.Length
    reads a
{   forall i, j:: (0 <= i < j < n) ==> a[i] <= a[j]  }
```

Example: BinarySearch

```
function sorted(a:array<char>, n:int):bool
    requires 0 <= n <= a.Length
    reads a
{   forall i, j:: (0 <= i < j < n) ==> a[i] <= a[j]  }

method BSearch(a:array<char>, n:int, value:char) returns (pos:int)
    requires 0 <= n <= a.Length && sorted(a, n)
    ensures …
    ensures …
{










}
```

Example: BinarySearch

```
function sorted(a:array<char>, n:int):bool
    requires 0 <= n <= a.Length
    reads a
{   forall i, j:: (0 <= i < j < n) ==> a[i] <= a[j]  }

method BSearch(a:array<char>, n:int, value:char) returns (pos:int)
    requires 0 <= n <= a.Length && sorted(a, n)
    ensures 0 <= pos ==> pos < n && a[pos] == value
    ensures pos < 0  ==> forall i :: (0<= i < n) ==> a[i] != value
{
```




```
}
```

Example: BinarySearch

```
function sorted(a:array<char>, n:int):bool
    requires 0 <= n <= a.Length
    reads a
{ forall i, j:: (0 <= i < j < n) ==> a[i] <= a[j] }

method BSearch(a:array<char>, n:int, value:char) returns (pos:int)
    requires 0 <= n <= a.Length && sorted(a, n)
    ensures 0 <= pos ==> pos < n && a[pos] == value
    ensures pos < 0  ==> forall i :: (0<= i < n) ==> a[i] != value
{
    var low, high := 0, n;
    while low < high
        decreases high - low
        invariant ???
        invariant ???
        invariant ???
    {
        var mid := (low + high) / 2;
        if a[mid] < value          { low := mid + 1; }
        else if value < a[mid]     { high := mid; }
        else /* value == a[mid] */ { return mid; }
    }
    return -1;
}
```

Example: BinarySearch

```
function sorted(a:array<char>, n:int):bool
    requires 0 <= n <= a.Length
    reads a
{ forall i, j:: (0 <= i < j < n) ==> a[i] <= a[j] }

method BSearch(a:array<char>, n:int, value:char) returns (pos:int)
    requires 0 <= n <= a.Length && sorted(a, n)
    ensures 0 <= pos ==> pos < n && a[pos] == value
    ensures pos < 0  ==> forall i :: (0<= i < n) ==> a[i] != value
{
    var low, high := 0, n;
    while low < high
        decreases high - low
        invariant 0 <= low <= high <= n
        invariant forall i :: 0 <= i < n && i < low ==> a[i] != value
        invariant forall i :: 0 <= i < n && high <= i ==> a[i] != value
    {
        var mid := (low + high) / 2;
        if a[mid] < value          { low := mid + 1; }
        else if value < a[mid]     { high := mid; }
        else /* value == a[mid] */ { return mid; }
    }
    return -1;
}
```