Construction and Verification of Software

2017 - 2018

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

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()
    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

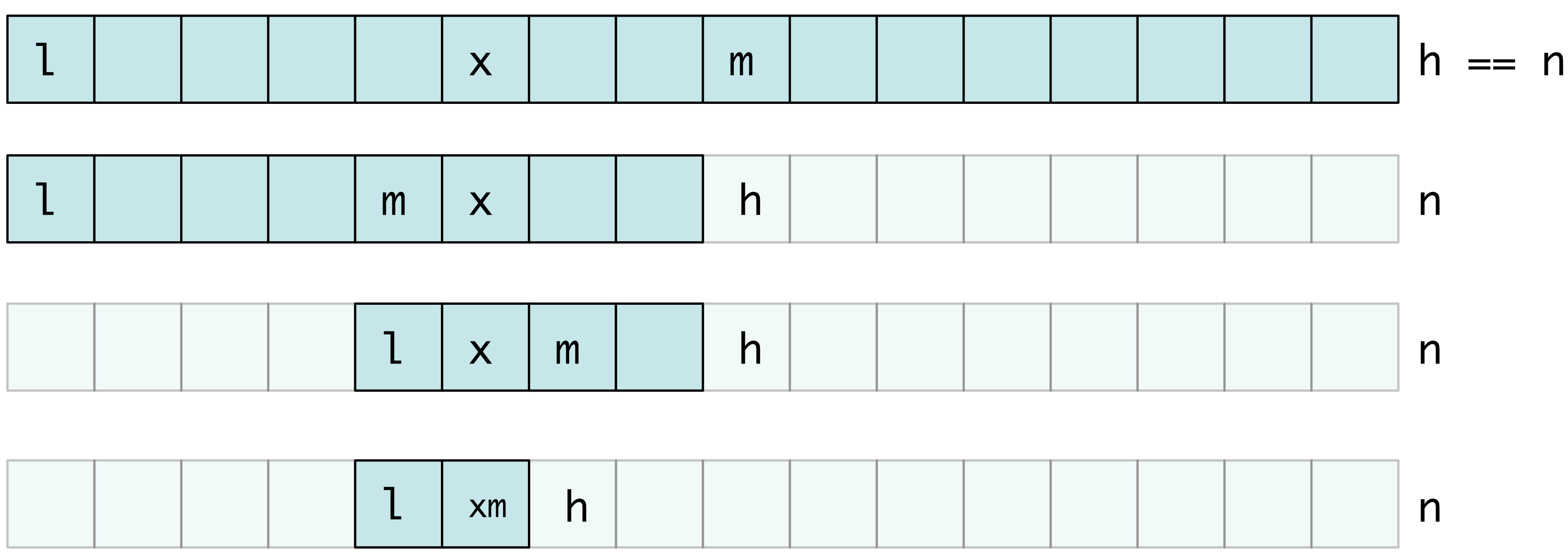
```
function sorted(a:array<char>, n:int):bool
    requires 0 <= n <= a.Length
    reads a
{   forall i, j:: (0 <= i < j < n) ==> a[i] <= a[j]  }
```

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;
}
```

Back to ADTs
Abstract State &
Representation state

RECAP

Technical ingredients in ADT design

- 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

Technical ingredients in ADT design

RECAP

- 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***.

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);
       }
   }
…
```

Changing state & Framing

Set ADT (growable)

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

   // Representation state
   var a:array<int>;
   var size:int;
…
   method Grow() returns (na:array<int>)
       requires RepInv()
       ensures size < na.Length
       ensures fresh(na)
       ensures forall k::(0<=k<size) ==> na[k] == a[k];
   {
       na := new int[a.Length*2];
       var i := 0;
       while (i<size)
           decreases size-i
           invariant 0 <= i <= size ;
           invariant forall k::(0<=k<i) ==> na[k] == a[k];
       {
           na[i] := a[i];
           i := i + 1;
       }
   }
```

Set ADT (growable)

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

    // Representation state
    var a:array<int>;
    var size:int;
…
    method add(x:int)
        modifies this,a;
        requires RepInv()
        ensures RepInv()
    {
        var i := find(x);
        if (i < 0) {
            if (size == a.Length)
            { a := Grow(); }
            a[size] := x;
            size := size + 1;
      }
    }
```

Set ADT (growable)

```
class ASet {
    method del(x:int)
        modifies this,a;
        requires RepInv()
        ensures RepInv()
    {
        var i:int := find(x);
        if (i >= 0) {
            assert a[i] == x && forall j::(0<=j<i) ==> a[j] == old(a[j]); // <<<<<<
            var pos:int := i;
            while (i < size-1)
                modifies a;
                decreases size - 1 - i
                invariant …
            {
                a[i] := a[i+1];
                i := i + 1;
            }
            size := size - 1;
        }
    }

}
```

Further hints on invariants

- We illustrate a famous issue related to using formal logic to reason about dynamical systems, the so-called “**frame-problem**”.
- There is no “purely logical” way of inferring what does not change after an action, we need in each case to specify for each action not only what changes, but also what has not (remains stable).
- E.g. this arises in reasoning about programs

{x.val() == a && y.val() == 0 }

x.inc() { x.val() == a+1 && y.val() == ?}

- How do we know changing x affects y or not?

Some hints on invariants

- Historically, the “frame problem” appeared while using logic to reason about robot actions.
- Consider the “action” axioms:

  Paint(t, x, c) ==> Colour(t+1, x, c)

  Move(x, p) ==> Pos(t+1, x, p)

- We would like the following implication to hold

  Colour(2, cube, red) && Pos(2,cube,6) && Paint(2, cube, blue)

  ==> Colour(3, cube, blue) && Pos(3,cube,6)

Some hints on invariants

- Unfortunately, there is no way to derive

  Colour(2, cube, red) && Pos(2,cube,6) && Paint(2, cube, blue)

  ==> Colour(3, cube, blue) **&& Pos(3,cube,6)**

- Painting has nothing to do with moving, and we would like to avoid having to explicitly say that.
- But “inertial” assumptions are always domain specific, and usually one needs to add “frame axioms” to assert what doesn’t change.
- While Hoare Logic does not need frame axioms, since there is no aliasing or invisible side effects.

Set ADT (growable)

```
class ASet {
    method del(x:int)
        modifies this,a;
        requires RepInv()
        ensures RepInv()
    {
      var i:int := find(x);
      if (i >= 0) {
          var pos:int := i;
          while (i < size-1)
            modifies a;
            decreases size - 1 - i
            invariant pos <= i <= size-1
            invariant unique(a,0,i) && unique(a,i+1,size)
            invariant forall j::(0 <= j < pos) ==> a[j] == old(a[j])
            invariant forall j::(pos <= j < i) ==> a[j] == old(a[j+1])
            invariant forall j::(i+1 <= j < size) ==> a[j] == old(a[j])
          {
              a[i] := a[i+1];
              i := i + 1;
          }
          size := size - 1;
      }
    }
}
```

Some hints on invariants

- In the previous code, we consider the invariants

```
invariant 0 <= pos <= i <= size-1;

invariant unique(a,0,i) && unique(a,i+1,size);

invariant forall j::(0<=j<pos) ==> a[j] == old(a[j]) ;

invariant forall j::(pos<=j<i) ==> a[j] == old(a[j+1]) ;

invariant forall j::(i+1<=j<size) ==> a[j] == old(a[j]);
```

- The method body assumes that all components of this and a can be modified as a side effect, so Dafny does not add any frame principles: we need to include the necessary ones in invariants.

Set ADT (growable)

```
class ASet {
…
    method Grow() returns (na:array<int>)
        requires RepInv()
        ensures size < na.Length
        ensures fresh(na)
        ensures forall k::(0<=k<size) ==> na[k] == a[k];
    {
        na := new int[a.Length*2];
        var i := 0;
        while (i<size)
            decreases size-i
            invariant 0 <= i <= size ;
            invariant forall k::(0<=k<i) ==> na[k] == a[k];
        {
            na[i] := a[i];
            i := i + 1;
        }
    }
}
```

Some hints on invariants

- In the previous code for grow, no frame conditions where added, since there is no **modifies** clause !
- In general, we need only add frame conditions for data that is declared to be subject to change by some **modifies** declaration.

Some hints on invariants

- Another way, much simpler, of expressing frame conditions, is to add **modifies** declarations directly in the loop, close to the invariants
- A **modifies** clause in a while loop overrides the method modifies clause, making it more precise.
- **Only** the instance variables mentioned in the loop modifies clause will be assumed to be changed by the loop.
- The other variables will be framed out, and Dafny will automatically know that they will not change.
- This will often save frame conditions.

Key Points

- The ADT operations pre / post conditions must always preserve the representation invariant
- Other operations (private helper methods) do not need to preserve the invariant, they are need to know about the ADT implementation details
- The ADT pre / post conditions should avoid referring to the concrete state, to preserve information hiding
- To do that, you may expose ghost variables
- Alternatively, use also some form of **typestate**, enough to express rich dynamic constraints (next lecture)

Construction and Verification of Software

2017 - 2018

MIEI - Integrated Master in Computer Science and Informatics
Consolidation block

Lab Assignment 3 - Loop Invariants (II) & ADTs

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

Aux Functions

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

Exercise 12

```
// the method inserts integer v in the sorted array a
// if a already contains v, the method does nothing

method Insert(a:array<int>, nelems:int, v:int) returns (newsize:int)
    modifies a;
    requires 0 <= nelems < a.Length-1 && sorted(a, nelems);
    ensures nelems <= newsize <= nelems+1 && sorted(a,newsize);
    ensures exists p:: 0<=p<newsize && a[p] == v;
{

}

// write the code and fully check it with dafny
// define the weakest preconditions you can think of
// define the strongest postconditions you can think of
```

Exercise 13

- sort

```
// the method sort returns in b a sorted array
// first consider the following post-conditions
// and write the code for sort (use the selection sort algorithm)

function Majors(c:array<int>,i:int,nelems:int):bool
    requires 0 <= nelems < c.Length
    reads c;
{
    // first i elems of c are <= than the elems from i to nelems-1
    forall k::0<=k<i ==> forall l::i<=l<nelems ==> (c[k] <= c[l])
}
```

Exercise 13

- sort

```
// the method sort returns in b a sorted array
// first consider the following post-conditions
// and write the code for sort (use the selection sort algorithm)

method Sort(a:array<int>, nelems:int, b:array<int>)
    modifies b
    requires 0 <= nelems < a.Length && 0 <= nelems < b.Length
    ensures sorted(b,nelems)
{

}

// to express the loop invariants, you may find it useful
// the function majors defined in the previous slide
```

Exercise 14

- `ADT PSet`

```
// Use the set implementation ASet of Lecture 3 and
// add to the representation invariants the property about
// all values being positive. Make the post-conditions
// stronger using that property

// Design some client methods and write assertions
// that are a consequence of the abstract invariant.
```

Exercise 15

- Extended Bank Account with movements

```
// Implement a bank account whose internal
// representation is an array of bank movements
// (debit, credit).

// Make the adequate abstract representation for
// the balance and define the soundness mapping.
```

Exercise (1st Handout 17/18)

- eliminate duplicates

```
// the method eliminates the duplicates in an array.

method Deduplicate(a:array<int>, n:int) returns (b:array<int>, m:int)
    requires 0 <= n <= a.Length
    requires sorted(a,n)
    ensures 0 <= m <= b.Length
    ensures sorted(b,m) && unique(b,m)
    ensures …???
{

…

}

// write the code and fully check it with dafny
// define the weakest preconditions you can think of
// define the strongest postconditions you can think of
```