Construction and Verification of Software

2018 - 2019

MIEI - Integrated Master in Computer Science and Informatics
Consolidation block

Lecture 3 - Specification and Verification (cont.)

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

Simple Programming Language

RECAP

E	$::=$	Expressions	
		num	Integer
	$\mid$	x	Variable
	$\mid$	$E + E \mid \ldots$	Integer operators
	$\mid$	$E < E \mid \ldots$	Relational operators
	$\mid$	E `and` $E \ldots$	Boolean operators

P	$::=$		Programs
		`skip`	No op
	$\mid$	$x := E$	Assignment
	$\mid$	$P; P$	Sequential Composition
	$\mid$	`if` E `then` P `else` P	Conditional
	$\mid$	`while` E `do` P	Iteration

Hoare Logic - Structural Rules

RECAP

$$\{A\}\ \texttt{skip}\ \{A\}$$

$$\{A[{}^{E}/_{x}]\}\ x := E\ \{A\}$$

$$\frac{\{A\}\ P\ \{B\} \qquad \{B\}\ Q\ \{C\}}{\{A\}\ P;Q\ \{C\}}$$

$$\frac{A' \implies A \qquad \{A\}\ P\ \{B\} \qquad B \implies B'}{\{A'\}\ P\ \{B'\}}$$

Rule for Assignment

$$\{A[{}^{E}/_{x}]\}\ x := E\ \{A\}$$

- A[E/x] means:
    - the result of replacing all free occurrences of variable x in assertion A by the expression E
- For this rule to be sound, we require E to be an expression without side effects (a pure expression)

Rule for Assignment

$$\{A[^{E}/_{x}]\}\ x := E\ \{A\}$$

- We can think of A as a condition where “x” appears in some places. A is a condition dependent on “x”.
- The assignment x := E changes the value of x to E, but leaves everything else unchanged
- So everything that could be said of E in the precondition, can be said of x in the postcondition, since the value of x after the assignment is E
- Example: { x + 1 > 0 } x := x + 1 { x > 0 }

Rule for Assignment

$$\{A[^E/_x]\}\ x := E\ \{A\}$$

- Example, let's check { x > -1 } x := x + 1 { x > 0 }

{ (x+1 > 0) } x := x+1 { x > 0 } by the := Rule

that is, { (x > 0)[x+1/x] } x := (x+1) { x > 0 }

{ x > -1 } x := x + 1 { x > 0 } by deduction

Rule for Assignment

$$\{A[^E/_x]\}\ x := E\ \{A\}$$

- Trick: if x does not appear in E or A.

  We can always write { A && E == E } x := E { x == E }
  So, if x does not occur in E, A the triple

  { A } x := E { A && x == E }

  is always valid

Rule for Assignment

$$\{A[^{E}/_{x}]\}\ x := E\ \{A\}$$

- Exercises. Derive:
  - { y > 0 } x := y { x > 0 && y == x }
  - { x == y } x := 2*x { y == x div 2 }
  - { P(y) && Q(z) } (here P and Q are any properties)

    x := y ; y := z; z:= x

    { P(z) && Q(y) }

Exercises

Prove using the assignment rule that:

```
assert y > 0;
x := y;
assert x > 0 && y == x;
```

```
assert y == x;
x := 2 * x;
assert y == x / 2;
```

Exercises

Prove using the assignment rule that:

```
function P(x:int):bool {
  …
}
function Q(x:int):bool {
  …
}

  var x := …;
  var y := …;
  var z;

  assert P(x) && Q(y);
  z := x;
  x := y;
  y := z;
  assert P(y) && Q(x);
```

Example

- Consider the program

  $P \triangleq$ if (x>y) then z := x else z := y

- We (mechanically) check the triple

  { true } P { z == max(x,y) }

Example

- Consider the program

$P \triangleq$ if (x>y) then z := x else z := y

- We (mechanically) check the triple

{ true } P { z == max(x,y) }

{ x == max(x,y) } z := x { z == max(x,y) }
{ x > y } z := x { z == max(x,y) }

{ y == max(x,y) } z := y { z == max(x,y) }
{ y >= x } z := y { z == max(x,y) }

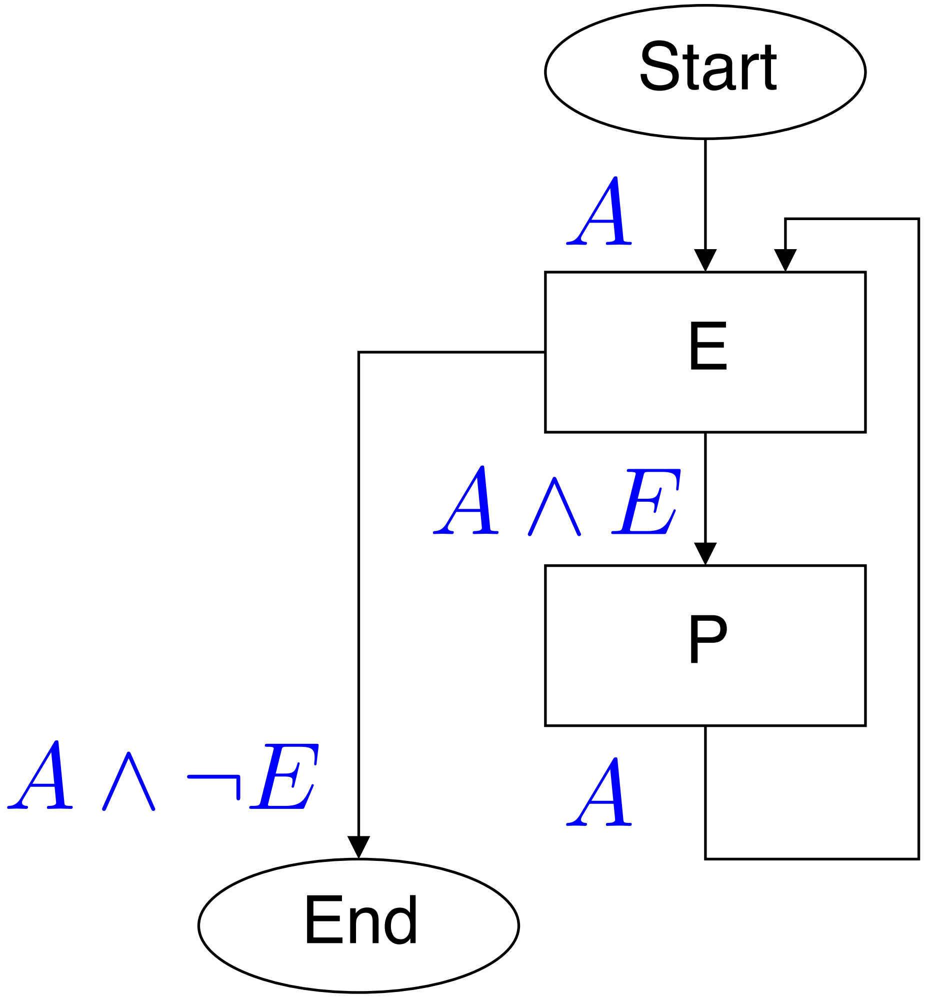
Rule for Iteration

$$\frac{\{? \wedge E\}\ P\ \{?\}}{\{A\}\ \texttt{while}\ E\ \texttt{do}\ P\ \{\neg E \wedge ?\}}$$

Any precise post condition depends on how many times P is executed … P can be executed 0, 1, 2 … n times, n not really known at verification time.

Rule for Iteration

$$\frac{\{A \wedge E\}\ P\ \{A\}}{\{A\}\ \texttt{while}\ E\ \texttt{do}\ P\ \{A \wedge \neg E\}}$$



Rule for Iteration

INV = Invariant Condition

$$\frac{\{Inv \land E\}\ P\ \{Inv\}}{\{Inv\}\ \texttt{while}\ E\ \texttt{do}\ P\ \{Inv \land \neg E\}}$$

- We cannot predict in general how many iterations will the while loop do (undecidability of the halting problem).
- We approximate all iterations by an invariant condition
- A loop invariant is a condition that holds at loop entry and at loop exit.

Rule for Iteration

INV = Invariant Condition

$$\frac{\{Inv \wedge E\}\ P\ \{Inv\}}{\{Inv\}\ \texttt{while}\ E\ \texttt{do}\ P\ \{Inv \wedge \neg E\}}$$

- If the invariant holds initially and is preserved by the loop body, it will hold when the loop terminates!
- It does not matter how many iterations will run
- Unlike for other rules of Hoare logic, finding the invariant requires human intelligence (you are a programmer :-)

Rule for Iteration

INV = Invariant Condition

$$\frac{\{Inv \wedge E\}\ P\ \{Inv\}}{\{Inv\}\ \texttt{while}\ E\ \texttt{do}\ P\ \{Inv \wedge \neg E\}}$$

- The invariant depicts the state in all iterations of a loop.
- The invariant works like the induction hypothesis in a proof. The base case is the loop executed 0 times, the loop body is the induction step that iterates from step n to n+1.There must exist a valid induction measure.

Loop Invariants

$$\{0 \leq n\}$$
$$i := 0;$$
$$\texttt{while } i < n \texttt{ do } \{$$
$$\quad i := i + 1$$
$$\}$$
$$\{i == n\}$$

Loop Invariants

$\{0 \leq n\}$
$i := 0;$
$\{i == 0 \land 0 \leq n\}$
$\{0 \leq i \leq n\}$
$\texttt{while}\ i < n\ \texttt{do}\ \{$
$\quad \{0 \leq i \leq n \land i < n\}$
$\quad \{0 \leq i < n\}$
$\quad \{0 \leq i + 1 \leq n\}$
$\quad i := i + 1$
$\quad \{0 \leq i \leq n\}$
$\}$
$\{0 \leq i \leq n \land i >= n\}$
$\{i == n\}$

Loop Invariants

- Consider program P defined by

$$P \triangleq s := 0;\ i := 0;\ \texttt{while}\ i < n\ \texttt{do}\ \{i := i + 1;\ s := s + i\}$$

- What is the specification of P? What does P do?

$$\{A\}\ P\ \{B\}$$

Loop Invariants

- Consider program P defined by

$$P \triangleq s := 0;\ i := 0;\ \texttt{while}\ i < n\ \texttt{do}\ \{i := i + 1;\ s := s + i\}$$

- What is the specification of P? What does P do?

$$\{n \geq 0\}\ P\ \{s = \sum_{j=0}^{n} .j\}$$

Is this a good specification for program P?

Can we mechanically check the Hoare triple?

Loop Invariants

$$\{0 \leq n\}$$

$$s := 0;$$

$$i := 0;$$

$$\texttt{while } i < n \texttt{ do } \{$$

$$\quad i := i + 1;$$

$$\quad s := s + i$$

$$\}$$

$$\{s == \textstyle\sum_j^n .j\}$$

Loop Invariants

$$\{0 \leq n\}$$
$$s := 0;$$
$$\{s = 0 \wedge 0 \leq n\}$$
$$i := 0;$$
$$\{s = 0 \wedge 0 \leq i \leq n\}$$
$$\texttt{while } i < n \texttt{ do } \{$$
$$\quad i := i + 1;$$
$$\quad s := s + i$$
$$\}$$
$$\{s = \textstyle\sum_j^n .j\}$$

Loop Invariants

$$\{0 \leq n\}$$
$$s := 0;$$
$$\{s = 0 \wedge 0 \leq n\}$$
$$i := 0;$$
$$\{s = 0 \wedge 0 \leq i \leq n\}$$
$$\{0 \leq i \leq n \wedge s = \textstyle\sum_{j=0}^{i} .j\}$$
$$\texttt{while } i < n \texttt{ do } \{$$
$$\quad i := i + 1;$$
$$\quad s := s + i$$
$$\}$$
$$\{i = n \wedge s = \textstyle\sum_{j=0}^{i} .j\}$$
$$\{s = \textstyle\sum_{j}^{n} .j\}$$

Loop Invariants

$\{0 \leq n\}$
$s := 0;$
$\{s = 0 \wedge 0 \leq n\}$
$i := 0;$
$\{s = 0 \wedge i = 0 \wedge 0 \leq i \leq n\}$
$\{0 \leq i \leq n \wedge s = \sum_{j=0}^{i} .j\}$
$\texttt{while}\ i < n\ \texttt{do}\ \{$
$\quad \{0 \leq i \leq n \wedge s = \sum_{j=0}^{i} .j\}$
$\quad i := i + 1;$
$\quad s := s + i$
$\quad \{0 \leq i \leq n \wedge s = \sum_{j=0}^{i} .j\}$
$\}$
$\{i = n \wedge s = \sum_{j=0}^{i} .j\}$
$\{s = \sum_{j}^{n} .j\}$

Invariant holds

Loop Invariants

- The loop invariant may be broken inside the body of the loop, but must be re-established at the end.
- Notice the assignment rule

$$\{A[^E/_x]\}\ x := E\ \{A\}$$

- that breaks the invariant…

$$\{0 \leq i \leq n \wedge i < n \wedge s = \textstyle\sum_{j=0}^{i} \cdot j\}$$
$$\{0 \leq i < n \wedge s = \textstyle\sum_{j=0}^{i} \cdot j\}$$
$$i := i + 1$$
$$\{0 \leq i - 1 < n \wedge s = \textstyle\sum_{j=0}^{i-1} \cdot j\}$$
$$\{0 \leq i \leq n \wedge s = \textstyle\sum_{j=0}^{i-1} \cdot j\}$$

Loop Invariants

- The loop invariant may be broken inside the body of the loop, but must be re-established at the end.
- Notice the assignment rule

$$\{A[{}^{E}/_{x}]\}\ x := E\ \{A\}$$

- and then re-establishes it

$$\{0 \leq i \leq n \wedge s = \textstyle\sum_{j=0}^{i-1} \cdot j\}$$
$$s := s + i$$
$$\{0 \leq i \leq n \wedge s = (\textstyle\sum_{j=0}^{i-1} \cdot j) + i\}$$
$$\{0 \leq i \leq n \wedge s = (\textstyle\sum_{j=0}^{i} \cdot j)\}$$

Loop Invariants

$\{0 \leq n\}$
$s := 0;$
$\{s = 0 \wedge 0 \leq n\}$
$i := 0;$
$\{s = 0 \wedge i = 0 \wedge 0 \leq i \leq n\}$
$\{0 \leq i \leq n \wedge s = \sum_{j=0}^{i} \cdot j\}$
$\texttt{while}\ i < n\ \texttt{do}\ \{$
$\quad \{0 \leq i \leq n \wedge s = \sum_{j=0}^{i} \cdot j\}$
$\quad i := i + 1;$
$\quad s := s + i$
$\quad \{0 \leq i \leq n \wedge s = \sum_{j=0}^{i} \cdot j\}$
$\}$
$\{i = n \wedge s = \sum_{j=0}^{i} \cdot j\}$
$\{s = \sum_{j}^{n} \cdot j\}$

Invariant holds

Loop Invariants

$\{0 \leq n\}$
$s := 0;$
$\{s = 0 \wedge 0 \leq n\}$
$i := 0;$
$\{s = 0 \wedge i = 0 \wedge 0 \leq i \leq n\}$
$\{0 \leq i \leq n \wedge s = \sum_{j=0}^{i} \cdot j\}$
$\texttt{while } i < n \texttt{ do } \{$
$\quad \{0 \leq i \leq n \wedge i < n \wedge s = \sum_{j=0}^{i} \cdot j\}$
$\quad \{0 \leq i < n \wedge s = \sum_{j=0}^{i} \cdot j\}$
$\quad i := i + 1;$
$\quad \{0 \leq i - 1 < n \wedge s = \sum_{j=0}^{i-1} \cdot j\}$ ← Invariant broken
$\quad \{0 \leq i \leq n \wedge s = \sum_{j=0}^{i-1} \cdot j\}$
$\quad s := s + i$
$\quad \{0 \leq i \leq n \wedge s = (\sum_{j=0}^{i-1} \cdot j) + i\}$ ← Invariant restored
$\quad \{0 \leq i \leq n \wedge s = \sum_{j=0}^{i} \cdot j\}$
$\}$
$\{i = n \wedge s = \sum_{j=0}^{i} \cdot j\}$
$\{s = \sum_{j}^{n} \cdot j\}$

Loop Invariants

$\{0 \leq n\}$
$s := 0;$
$\{s = 0 \wedge 0 \leq n\}$
$i := 0;$
$\{s = 0 \wedge i = 0 \wedge 0 \leq i \leq n\}$
$\{0 \leq i \leq n \wedge s = \sum_{j=0}^{i} \cdot j\}$
$\texttt{while}\ i < n\ \texttt{do}\ \{$
$\quad \{0 \leq i \leq n \wedge i < n \wedge s = \sum_{j=0}^{i} \cdot j\}$
$\quad \{0 \leq i < n \wedge s = \sum_{j=0}^{i} \cdot j\}$
$\quad i := i + 1;$
$\quad \{0 \leq i - 1 < n \wedge s = \sum_{j=0}^{i-1} \cdot j\}$
$\quad \{0 \leq i \leq n \wedge s = \sum_{j=0}^{i-1} \cdot j\}$
$\quad s := s + i$
$\quad \{0 \leq i \leq n \wedge s = (\sum_{j=0}^{i-1} \cdot j) + i\}$
$\quad \{0 \leq i \leq n \wedge s = \sum_{j=0}^{i} \cdot j\}$
$\}$
$\{i = n \wedge s = \sum_{j=0}^{i} \cdot j\}$
$\{s = \sum_{j}^{n} \cdot j\}$

Invariant holds

Hints for finding loop invariants

- **First**: carefully think about the post condition of the loop
  - Typically the post-condition talks about a property “accumulated” across a “range”
    (this is why you are using a loop, right?)
  - e.g., maximum of all elements of an array
  - e.g., sort visited elements in a data structure

Hints for finding loop invariants

- **Second**: design a “generalised” version of the post-condition, in which the already visited part of the data is made explicit as a function of the “loop control variable”

- The loop body may temporarily break the invariant, but must restore it at the end of the body

- **Important**: make sure that the invariant together (&&) with the termination condition really implies your post-condition

Examples, what kind of invariant we need for…

- **Max of an array**
  - All elements to the left are smaller than the max so far
- **Array Searching (unsorted)**
  - All elements left of the index are different from the value being searched
- **Array Searching (sorted)**
  - The element is between the lower and the higher limits
- **Sorting (bubblesort, insertion sort, etc.)**
  - Everything to the left of the cursor is sorted
- **List Reversing**
  - All elements to the left of the cursor are placed on the right of the result

Exercise - Fibonacci

```
function fib(n : int) : int
// this is the recursive spec of fibonacci
requires n>=0;
{
  if (n==0) then 1 else
  if (n==1) then 1 else fib(n-1)+fib(n-2)
}

// the method fibo below should implement fib efficiently
// “bottom up” using a while loop
method fibo(n : int) returns (f : int)
requires n>=0;
ensures f == fib(n);
{
    …
}
```

Exercise - Fibonacci

```
method Fib(n:int) returns (f:int)
  requires n >= 0
  ensures f == fib(n)
{
  if( n == 0 || n == 1 ) { return 1; }
  var a := 1;
  var b := 1;
  var i := 1;
  while i < n
    decreases n - i
    invariant 1 <= i <= n
    invariant a == fib(i-1)
    invariant b == fib(i)
  {
    a, b := b, a+b; // fib(i-1) + fib(i) = fib(i+1)
    i := i + 1;
  }
  f := b;
}
```

Exercise

```
// return the maximum of the values in array a[-]
// in positions i such that 0 <= i < n

method Max(a:array<int>, n:int) returns (m:int)




// 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 - Max of an array

```
method maxArray(a:array<int>, n:int) returns (max:int)
  requires 0 < n <= a.Length
  ensures forall j :: 0 <= j < n ==> a[j] <= max
{
  max := a[0];
  var i := 1;
  while i < n
     …
  {
    if max < a[i] {
      max := a[i];
    }
    i := i + 1;
  }
}
```

Exercise 6 - daffy solution

```
method maxArray(a:array<int>, n:int) returns (max:int)
  requires 0 < n <= a.Length
  ensures forall j :: 0 <= j < n ==> a[j] <= max
{
  max := a[0];
  var i := 1;
  while i < n
    decreases n-i
    invariant 0 <= i <= n
    invariant forall j :: 0 <= j < i ==> a[j] <= max
  {
    if max < a[i] {
      max := a[i];
    }
    i := i + 1;
 }
}
```

Abstract Data Types
Classes and Objects

Abstract Data Types (Liskov, 78)

- ADTs are the building blocks for software construction
  - Consists of:
    - A description of the data elements of the type
    - A set of operations over the data elements of the ADT
  - A software system is a composition of ADTs
  - ADTs behave like regular types in a programming language
  - Promotes modularity, encapsulation, information hiding, and hence reuse, modifiability, and correctness.

ADTs (Liskov & Zilles,78)

PROGRAMMING WITH ABSTRACT DATA TYPES

Barbara Liskov
Massachusetts Institute of Technology
Project MAC
Cambridge, Massachusetts

Stephen Zilles
Cambridge Systems Group
IBM Systems Development Division
Cambridge, Massachusetts

Abstract

The motivation behind the work in very-high-level languages is to ease the programming task by providing the programmer with a language containing primitives or abstractions suitable to his problem area. The programmer is then able to spend his effort in the right place; he concentrates on solving his problem, and the resulting program will be more reliable as a result. Clearly, this is a worthwhile goal.

Unfortunately, it is very difficult for a designer to select in advance all the abstractions which the users of his language might need. If a language is to be used at all, it is likely to be used to solve problems which its designer did not envision, and for which the abstractions embedded in the language are not sufficient.

This paper presents an approach which allows the set of built-in abstractions to be augmented when the need for a new data abstraction is discovered. This approach to the handling of abstraction is an outgrowth of work on designing a language for structured programming. Relevant aspects of this language are described, and examples of the use and definitions of abstractions are given.

Barbara Liskov (MIT)

BARBARA LISKOV
United States – **2008**

For contributions to practical and theoretical foundations of programming language and system design, especially related to data abstraction, fault tolerance, and distributed computing.

Abstract Data Type

Abstract types are intended to be very much like the built-in types provided by a programming language. The user of a built-in type, such as integer or integer array, is only concerned with creating objects of that type and then performing operations on them. He is not (usually) concerned with how the data objects are represented, and he views the operations on the objects as indivisible and atomic when in fact several machine instructions may be required to perform them. In addition, he is not (in general) permitted to decompose the objects. Consider, for example, the built-in type integer. A programmer wants to declare objects of type integer and to perform the usual arithmetic operations on them. He is usually not interested in an integer object as a bit string, and cannot make use of the format of the bits within a computer word. Also, he would like the language to protect him from foolish misuses of types (e.g., adding an integer to a character) either by treating such a thing as an error (strong typing), or by some sort of automatic type conversion.

Abstract Data Type (External View)

- External View
  - A public opaque data type (that clients will use)

    Note: opaque means = behaves as a primitive type
  - A set of operations on this data type
  - Operations must neither reveal, nor allow a client to invalidate the internal representation of the ADT
  - pre and post conditions on these operations must be expressed in terms of the abstract type (the only type known to the client)
  - This is why ADTs promote reuse, modifiability, and correctness: the developer can change the implementation anytime, without breaking contracts

Abstract Data Type (Internal View)

- Internal View
  - A **representation** data type (hidden from clients)
  - A set of operations on the representation data type
- ***important remarks***
  - A programmer must define the operations in such a way that the representation state (invisible to clients) is kept consistent with the intended abstract state
  - Pre-conditions on the public operations, expressed on the abstract state, must map into pre-conditions expressed in terms of the representation state
  - The same for post-conditions
  - At all times the concrete state must represent a well defined abstract state (otherwise something is wrong!)

Example (Positive Set ADT)

```
class PSet {
// an abstract PSet aset

    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 {…}
    // returns number of elems equal to v in aset

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

}
```

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 choose 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

- 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-*** *and* ***post- conditions***
  - expressed for the representation type
  - also expressed for the abstract type (for client code)

Bank Account ADT

- Abstract State
  - the account balance (`bal`)
  - `bal` is of type `int` subject to the constraint (`bal >= 0`)

Bank Account ADT

- Representation type
  - an integer `bal`
  - in this simple case the representation type is the same as the abstract type
  - the true "meaning" of the representation and abstract types are different
  - not all operations on integers are valid on account balances (e.g., to multiply bank accounts)

Bank Account ADT

- Representation type
  - an integer `bal`
  - in this simple case the representation type is the same as the abstract type
  - the true “meaning” of the representation and abstract types are different
  - not all operations on integers are valid on account balances (e.g., to multiply bank accounts)
- Representation invariant
  - (`bal >= 0`)
  - this time, pretty simple

Example (Account)

```
class Account {
    var bal: int;

    function RepInv():bool
    // specifies the representation invariant
        reads this ;
    {
        bal >= 0
    }
   …
}
```

Example (Account)

```
class Account {
    var bal: int;

    function RepInv():bool
    // specifies the representation invariant
      reads this ;
    {
      bal >= 0
    }

    method Init()
        modifies this;
        ensures RepInv()
    { bal := 0; }
    …
}
```

Example (Account)

```
class Account {
    var bal: int;
…
    // All operations must require the representation invariant
    // All operations must ensure the representation invariant
    method deposit(v:int)
        modifies this;
        requires RepInv() && v >= 0
        ensures RepInv()
    { bal := bal + v; }

    method withdraw(v:int)
        modifies this;
        requires RepInv() && v >= 0
        ensures RepInv()
    { if (bal>=v) { bal := bal - v; } }
}
```

Example (Account)

```
class Account {
    var bal: int;
…
   function getBal():int
         reads this
    { bal }

    method withdraw(v:int)
         modifies this;
         requires RepInv() && 0 <= v <= getBal()
         ensures RepInv()
    { bal := bal - v;  }
}
```

Set ADT

```
class ASet {
// an abstract Set aset

    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.