Construction and Verification of Software

2021 - 2022

MIEI - Integrated Master in Computer Science and Informatics
Consolidation block

Lecture 5 - Abstract Data Types

Bernardo Toninho (btoninho@fct.unl.pt)
based on previous editions by **João Seco** and **Luís Caires**

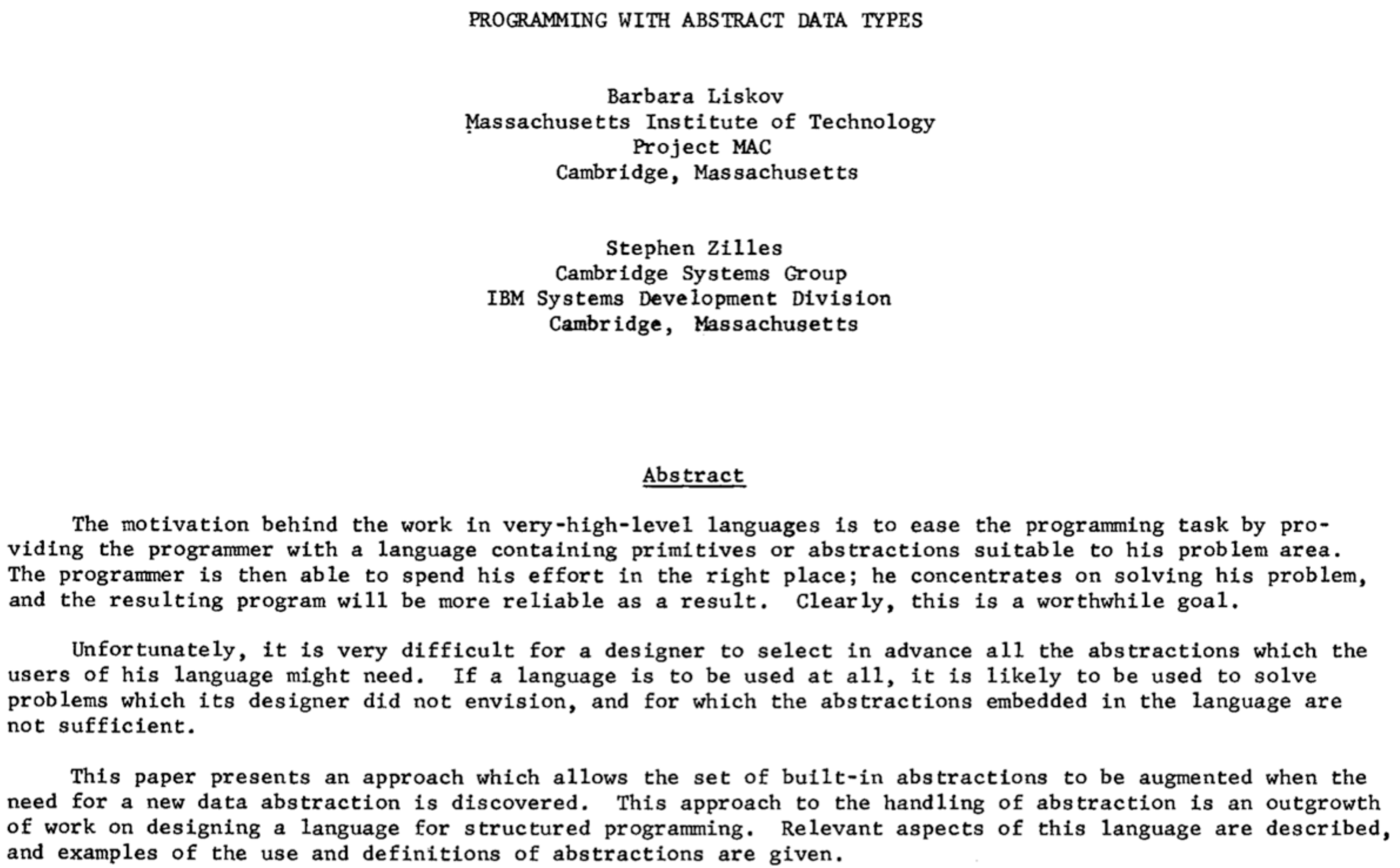
NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Part I
Abstract Data Types
(intro)

Abstract Data Types (Liskov, 78)

- ADTs are the building blocks for software construction
  - Consist 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.

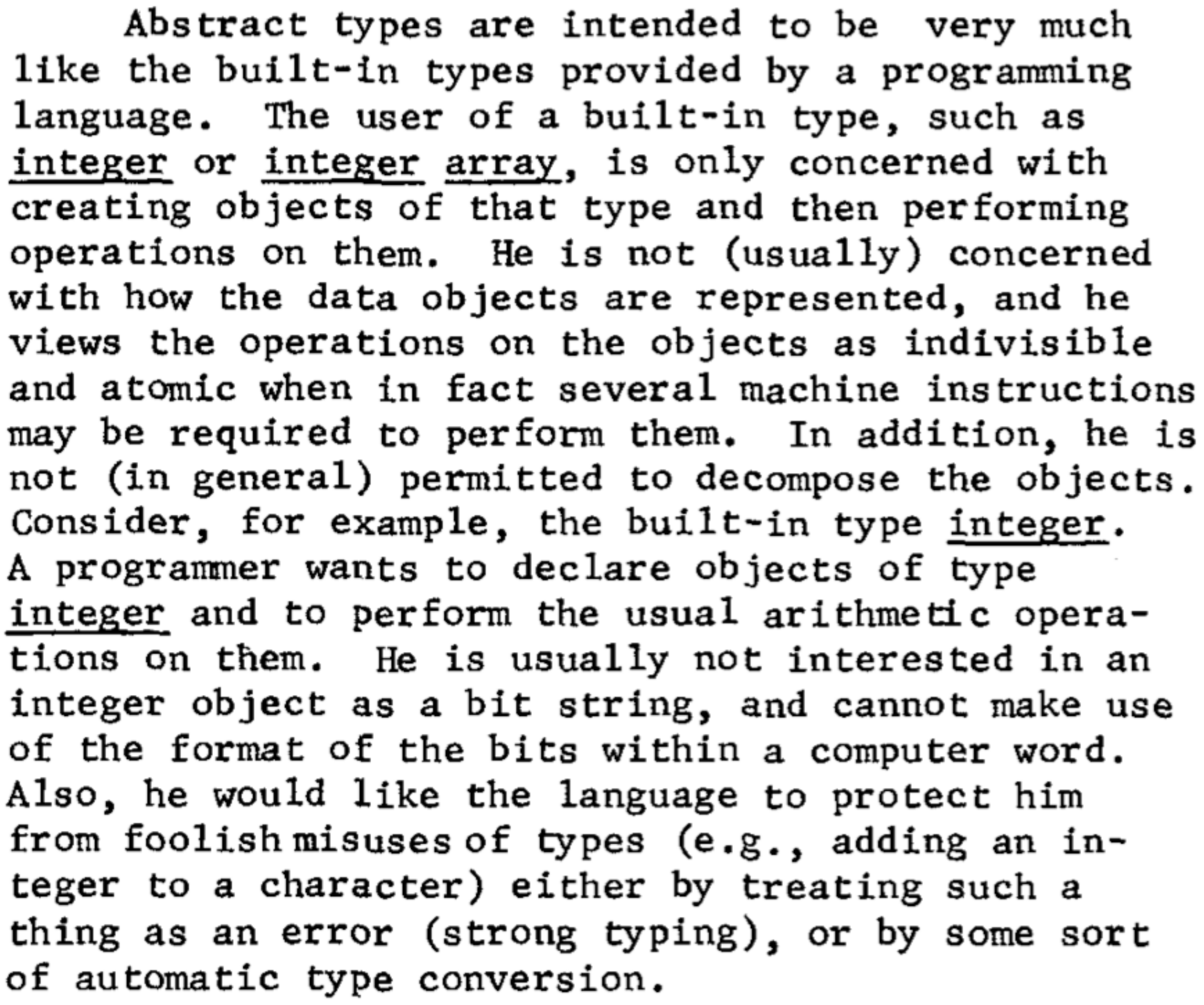
Barbara Liskov (MIT)

A.M. TURING AWARD

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
- ***Key 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 set of positive numbers

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

    method add(v:int) {…}
    // adds v to the set if space available

    function size() : int {…}
    // returns number of elems in the set

    function contains(v:int) : bool {…}
    // returns number of elems equal to v in the set

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

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)

Part II
Abstract Data Types (with objects)

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;

    predicate RepInv()
    // specifies the representation invariant
       reads this
    {
       bal >= 0
    }
   …
}
```

Example (Account)

```
class Account {
    var bal: int;

    predicate RepInv()
    // specifies the representation invariant
      reads this
    {
      bal >= 0
    }

    constructor()
        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 method getBal():int
        reads this
    { bal }

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

Set ADT

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

    constructor(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
  - $<nelems=n, store=[v_0,v_1,\ldots v_{store.Length-1}]> \rightarrow \{v_0,\ldots,v_{n-1}\}$
  - more later ....

Set ADT

```
class ASet {

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

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

…
```

Set ADT

```
class ASet {

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

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

  predicate RepInv()
    reads this,a
  {
   …
  }
…
```

Set ADT

```
class ASet {

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

…

  predicate RepInv()
    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;

…
  predicate unique(b:array<int>, l:int, h:int)
  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 method count():int
  reads this,a;
  requires RepInv()
  { size }

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

  method add(x:int)
  modifies this,a;
  requires RepInv() && 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 := 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);
  }
}
```

Part III
Soundness and Abstraction Map

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 state, specification operations, and abstraction map soundness check.

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

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
- 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 abstract state with ***ghost state***.

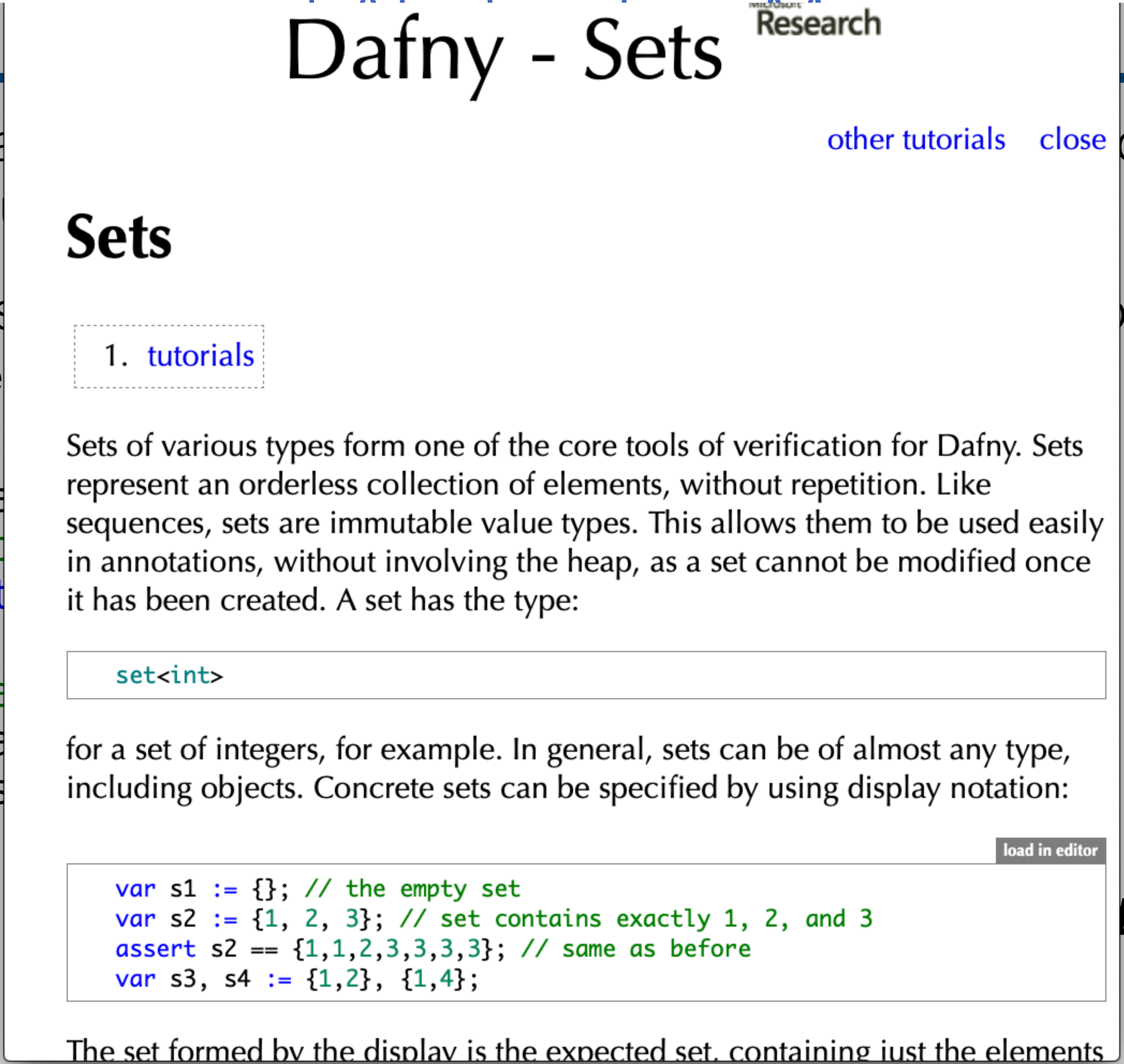
Sound

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

```
class ASe
    // Ab
    ghost

    // Re
    var a
    var s
```

- We the *t variable*.



Dafny - Sets

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 predicate Sound() that specifies the precise relationship the abstract and concrete state:

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

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

   predicate Valid()
       reads this,a
   { RepInv() && Sound() }

   // Spec
   predicate unique(b:array<int>, l:int, h:int)
       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 Valid() && 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 Valid()
        ensures  Valid()
        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 Valid()
            requires count() < maxsize()
            ensures  Valid() && 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);
            }
        }
    …
```

Next lecture

– Framing in Hoare Logic

– Changing state and Dynamic Frames

– Typestates

Changing State

- Tracking state changes and invalidating knowledge along the proof is crucial in the verification of imperative programs.
- This is captured by the derived (constancy or frame) rule

$$\frac{\{A\}\ P\ \{B\}}{\{A \wedge C\}\ P\ \{B \wedge C\}} \quad \text{where } M(P) \cap V(C) = \emptyset$$

- Provided that the variables modified by P (M(P)) are not referred by C (V(C)).

$$\frac{\{x > 0\}\ y := x\ \{y > 0 \wedge x = y\}}{\{x > 0 \wedge z < 0\}\ y := x\ \{y > 0 \wedge x = y \wedge z < 0\}}$$

Changing State

- Tracking state changes and invalidating knowledge along the proof is crucial in the verification of imperative programs.
- This is captured by the derived (constancy or frame) rule

$$\frac{\{A\}\ P\ \{B\}}{\{A \wedge C\}\ P\ \{B \wedge C\}} \text{ where } M(P) \cap V(C) = \emptyset$$

- Provided that the variables modified by P (M(P)) are not referred by C (V(C)).
- Updates to variables do not allow framing the modified variables.

```
method deposit(v:int)
  modifies this`bal
```

$\longleftarrow$ like one assignment

Changing State

- Tracking changes with dynamic memory is not covered by the original Hoare Logic. Each tool adopts some kind of strategy to make the frame rule sound.

```
function AbsInv():bool
  reads this`a, this`size, this`s, this.a
{ RepInv() && Sound() }
```

memory referred by

```
method add(x:int)
  requires AbsInv()
  ensures AbsInv()
  modifies this`a, this.a, this`size, this`s
```

field of

contents of

memory modified by

- Dafny refers to allocated memory areas. Objects and arrays. Modification of fields are modifications to the container object.

Changing State

- Tracking state changes and invalidating knowledge along the proof is crucial in the verification of imperative programs.

$$\frac{\{A\}\ P\ \{B\}}{\{A \wedge C\}\ P\ \{B \wedge C\}}$$

- Information in the interface is important to know modified and referred memory.

```
method Main() {
  var a:Account := new Account();
  a.deposit(10);
  a.withdraw(20); <<<<< ????
  if a.getBalance() > 10
  { a.withdraw(10); a.deposit(10); }
}
```

Changing State

- Tracking state changes and invalidating knowledge along the proof is crucial in the verification of imperative programs.

$$\frac{\{A\}\ P\ \{B\}}{\{A \wedge C\}\ P\ \{B \wedge C\}}$$

- Information in the interface is important to know modified and referred memory.

```
method Main() {
  var a:Account := new Account(); { a.bal >= 0 }
  { a.bal >= 0 } a.deposit(10); { a.bal >= 0 }
  { a.bal >= 0 } a.withdraw(20); <<<<< ????
  if a.getBalance() > 10
  { { a.bal > 10 } a.withdraw(10); a.deposit(10); }
}
```