Object invariants

Conceptually an object is rather like a loop:

The lifetime behaviour of an object of class

```
class C
    private var f0 : T0
    private var f1 : T1
    public constructor() U end constructor

    public method m0() S0 end m0

    public method m1() S1 end m1
end C
```

is described by this loop

```
var f0 : T0
var f1 : T1
U
while true do
    var m : message := get next message
    switch( m )
    case m0 do S0 end case
    case m1 do S1 end case
    end switch
end while
```

This loop may have a loop invariant.

Such an invariant reflects the “consistent” states of the object. I.e. the ones that “make sense”.

This loop invariant expresses what we expect to be true before and after each message.

We call such an invariant an **object invariant** or a **class invariant**.

It is implicitly a postcondition of construction and a pre- and postcondition of each method. I.e. each method must **preserve** the object invariant

Example

A date class

class GregorianDate
  private var d : int, m : int, y : int
  invariant $y \geq 1 \quad \wedge \quad 1 \leq m \leq 12 \quad \wedge \quad 1 \leq d \leq 31$
  invariant $m \in \{4, 6, 9, 11\} \Rightarrow d \leq 30$
  invariant $m = 2 \Rightarrow d \leq 29$
  invariant $m = 2 \wedge y \nmid 400 \wedge (y \nmid 4 \vee y \mid 100) \Rightarrow d \leq 28$
  ...
  public method *incr*()
    $d := d + 1$
    if $d = 32$
    $\vee\ m \in \{4, 6, 9, 11\} \wedge d = 31$
    $\vee\ m = 2 \wedge d = 30$
    $\vee\ m = 2 \wedge y \nmid 400 \wedge (y \nmid 4 \vee y \mid 100) \wedge d = 29$
    then $d := 1$ $m := m + 1$
      if $m = 13$ then $y := y + 1$ $m := 1$ end if end if
  end *incr*
end Date

Example

class DynamicArray$\langle T \rangle$
  private var a : array$\langle T \rangle$:= new array$\langle T \rangle$(1)
  private var len: int := 0
  invariant $0 \le len \le a.\text{length}$
  public method *getLength*( ) : int
    return *len*
  end *getLength*
  public method *get*(i : int) : T
    precondition $0 \le i < getLength()$
    return $a(i)$
  end *get*
  public method *set*(i : int, v : T)
    precondition $0 \le i \le getLength()$
    ...
  end *set*
  public method *clipTo*(i : int)
    precondition $0 \le len \le a.\text{length}$
    changes a, *len*
    $len := i$
  end *clipTo*
end DynamicArray

In order for *get* to work we need that $0 \le len \le a.\text{length}$. Thus this is an invariant.

We need to ensure the invariant is true after object construction and after execution of each public method.

In return we can assume that it is true at the start of each public method.

Now let's look at implementing DynamicArray so as to preserve the invariant.

```
class DynamicArray⟨T⟩
    private var a : array⟨T⟩ := new array⟨T⟩(1)
    private var len: int := 0
    invariant 0 ≤ len ≤ a.length ∧ 0 < a.length
    ...
    public method set( i : int, v : T )
        precondition 0 ≤ i ≤ getLength()
        if i = len then
            len := len + 1
            if len > a.length then
                    // Must restore the object invariant
                    var b := new array ⟨T⟩ (a.length × 2)
                    for j ← {0, ..a.length} do b(j) := a(j)
                    end for
                    a := b
            end if
        end if
        a(i) := v
    end set
end DynamicArray
```

As with a loop invariant, the object invariant may be temporarily violated, but should be restored by the end of the routine.

For set to work, we need that the array has a positive length. This is why $0 < a.\text{length}$ is in the invariant. (Having changed the invariant, we must now check that all methods preserve it.

Concrete data type

For a client programmer to know what the methods "mean", they must read the implementation! This is not good.

For the implementing programmer, how can they know that they have implemented the methods correctly?

Can we describe the "meaning" of each method by pre- and postconditions? This will serve as a *contract* between the client and the implementer.

Let's try.

[Notational notes:

- In postconditions, we will use v_0 to represent the value of a variable v at the start of a subroutine invocation.
- In postconditions, we use $result$ to stand for the result.
- Variables not mentioned after "changes" do not change.]

```
class DynamicArray⟨T⟩
    private var a : array⟨T⟩ := new array⟨T⟩(1)
    private var len: int := 0
    invariant 0 ≤ len ≤ a.length ∧ 0 < a.length
    public method getLength( ) : int
        postcondition result = len
        return len
    end getLength
    public method get( i : int ) : T
        precondition 0 ≤ i < getLength()
        postcondition result = a(i)
        return a(i)
    end get
    public method set( i : int, v : T )
        precondition 0 ≤ i ≤ getLength()
        changes a, len
        postcondition len = max(i + 1, len₀) ∧ a(i) = v
         ∧ (∀j : {0, ..len₀} · j ≠ i ⇒ a(j) = a₀(j))
        ... as before ...
    end set
    public method clipTo( i : int )
        precondition 0 ≤ i ≤ length(s)
        changes a, len
        postcondition len = i ∧ (∀j : {0, ..i} · a(j) = a₀(j))
        len := i
    end clipTo
end DynamicArray
```

But these contracts:

- are in terms of private variables that are really none of the client's business.
- are tied to the particular data representation that we chose to use.
- must change if that representation changes.
- (therefore, client code may become incorrect!)
- force reasoning about the client code to involve reasoning about the array-based implementation.

In short, they *violate information hiding*.

They don't really describe how a DynamicArray can be useful to a client in a way that abstracts away from the implementation particulars and leaves only the logical essence of the DynamicArray.

Let's start again.

Abstract data type

This time we will ignore implementation in terms of arrays and simply make use of a “sequence” type.

Notations for sequences

- $\mathrm{Seq}\langle T\rangle$ — the type of all finite sequences with items of type T
- $[\,]$ — a sequence of length 0
- $[a]$ — a sequence of length 1
- $s\hat{\ }t$ — the catenation of two sequences
- $s(i)$ — item i (starting at 0, of course)
- $s[i,..j]$ — a segment from position i up to, but not including j
- $length(s)$ — the length of s

(Abstract data type for dynamic array.)

interface *DynamicArrayI*$\langle T\rangle$

  ghost public readonly var $s : \mathrm{Seq}\,\langle T\rangle := [\,]$

  public method *getLength*( ) : int

    postcondition $result = length(s)$

  public method *get*(i : int) : T

    precondition $0 \le i < length(s)$

    postcondition $result = s(i)$

  public method *set*(i : int, v : T)

    precondition $0 \le i \le length(s)$

    changes s

    postcondition $s = s_0[0,..i]\,\hat{}\,[v]\,\hat{}\,s_0[i+1,..length(s)]$

  public method *clipTo*(i : int)

    precondition $0 \le i \le length(s)$

    changes s

    postcondition $s = s_0[0,..i]$

end *DynamicArrayI*

This version precisely documents the **interface** of an abstract data type.

We call s an **abstract** field or a **ghost** field

But what about the implementation?

We data refine the class.

Introduce (**concrete**) fields len and a to represent s.

The linking invariant is

$$\begin{aligned} LI :\ & length(s) = len \le a.\mathrm{length} \\ & \wedge\ a.\mathrm{length} > 0 \\ & \wedge\ (\forall j : \{0,..len\} \cdot s(j) = a(j)) \end{aligned}$$

Once we have decided on the linking invariant, the individual methods can be implemented independently of each other.

The complete class is this

class *DynamicArray*$\langle T\rangle$ implements *DynamicArrayI*$\langle T\rangle$
// ghost variable $s : \mathrm{Seq}\,\langle T\rangle := [\,]$ is inherited
private var a : array$\langle T\rangle$:= new array$\langle T\rangle$(1)
private var len: $\mathrm{int} := 0$
invariant $length(s) = len \leq a.\mathrm{length} \wedge a.\mathrm{length} > 0$
$\wedge\ (\forall j : \{0, ..len\} \cdot s(j) = a(j))$
public method *getLength*( ) : int
postcondition $result = length(s)$
return len
end *getLength*
public method *get*(i : int) : T
precondition $0 \leq i < length(s)$
postcondition $result = s(i)$
return $a(i)$
end *get*
public method *set*(i : int, v : T)
precondition $0 \leq i \leq length(s)$
changes s
postcondition $s = s_0[0, ..i]\hat{}[v]\hat{}s_0[i+1, ..length(s)]$
...as before...
ghost $s := s[0, ..i]\hat{}[v]\hat{}s[i+1, ..length(s)]$
end *set*

public method *clipTo*(i : int)
precondition $0 \le i \le length(s)$
changes s
postcondition $s = s_0[0, ..i]$
$len := i$
ghost $s := s[0, ..i]$
end *DynamicArray*

The annotation "ghost" indicates code that is only needed for verification — it should not be compiled.

When is a subroutine correct? Consider $clipTo$. We should show

$$\{0 \le i \le length(s) \wedge s_0 = s \wedge LI\}$$
$$len := i \; s := s[0, ..i]$$
$$\{s = s_0[0, ..i] \wedge LI\}$$

The object invariant that we had earlier

$$len \le a.\text{length} \wedge 0 < a.\text{length}$$

is merely the part of the linking invariant that involves only the concrete variables

We call this the **concrete invariant.**

(Technically the concrete invariant can be obtained from the linking invariant, by “projecting” onto the concrete space $CI = (\exists s \cdot LI)$.)

This concrete invariant can (and likely should) be checked using run-time assertion checking. (Unless 0-false-negative static verification technology is used, in which case, run-time checking would be superfluous.)

There is also an abstract invariant. In this example it is true, as we put no restrictions on the sequence being represented.

(Technically the abstract invariant can be obtained from the linking invariant, by "projecting" onto the abstract space $AI = (\exists len \cdot \exists a \cdot LI)$.)

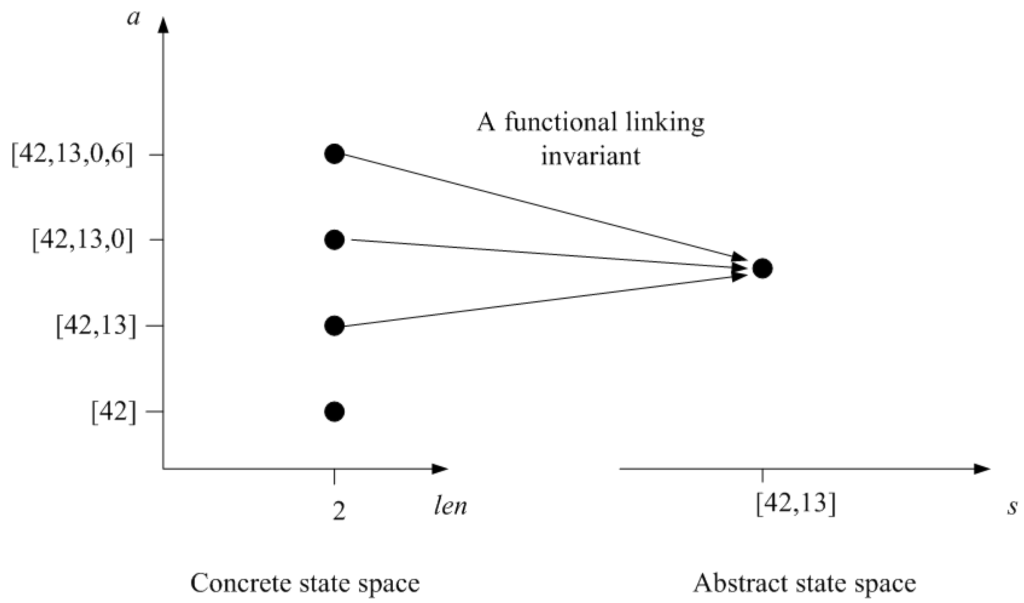
You can think of the ghost variables as being the axes of a state space. We call this the **abstract state space** of the ADT.

In the dynamic array example there is only one axis s of the abstract state space

The concrete variables are the axes of a **concrete state space**.

In the dynamic array example, the axes of the concrete state space are a and len.

The linking invariant links points in the concrete state space with points in the abstract state space.



For example $s = [42, 13]$ is a point in the abstract state space.

It is linked to every concrete point such that $len = 2 \leq a.\text{length} \wedge a(0) = 42 \wedge a(1) = 13$.

In this example:

$*$ Every point in the abstract space is linked to an infinity of points in the concrete space.

* Every point in the concrete space, which obeys the concrete invariant, is linked to exactly one point in the abstract space.