Abstract data types (ADT)

The verification of a ADT (or other class) needs three definitions:

Representation (*footprint/dynamic frame*) set of objects of the *heap* associated to each instance and can be modified/created: set that includes also the instance and field objects (if any)

```
ghost  var Repr: set<object>
```

Model ADT Specification for verification (functional)

```
ghost  var Model: seq<A>
```

Invariant identify the valid instances of the model: `Valid()`

```
ghost predicate Valid()
reads this, Repr
```

even if `this` in `Repr` that only happens if `Valid()` returns `true`, thus `reads` has to include `this`.

Abstraction/ Model

- Contracts must be abstract and do not reflect the internal implementation
- For instance data structures as sets or priority queues can be represented using arrays, hash, binary trees, etc.
- but contracts should not reflect those structures
- Abstract state representations can be,
  - `ghost` variables that internally are associated with the concrete implementation
  - `ghost` functions, that allow a abstract view of the internal state

Summary on Dynamic Frames

Representation Set `ghost var Repr:  set<object>`

Invariant `ghost predicate Valid()`

```
reads this, Repr
ensures Valid() ==> this in Repr
{ this in Repr && a.Valid() && a in Repr && b.Valid() && b.Repr <= Repr
  && this !in b.Repr  &&
 //
}
```

for each `a` that is a field object with a simple frame and each `b` field object with a dynamic frame. Other disjointness conditions may apply

Constructor `constructor ()`

```
ensures Valid() && fresh(Repr)
{
//
new;
Repr := {this,a,b} + b.Repr
//
}
```

where `b` is a constituent object with a dynamic frame.

Functions `function F(x: X): Y`

```
  requires Valid()
  reads Repr
```

Methods A mutating method has Repr as its write frame

```
method M(x: X) returns (y: Y)
  requires Valid()
  modifies Repr
  ensures Valid() && fresh(Repr - old(Repr))
```

Summary of techniques to deal with heap updates

- Explicit representation sets are programmer-defined. Examples include dynamic frames and the Dafny tool.
- Implicit representation sets are derived from predicates in specialized logics. Examples include separation logics and the VeriFast and Viper tools.
- Predefined representation sets are derived from predefined heap topologies (commonly, a tree topology). Examples include the ownership relations in Spec#, VCC and regions in whyML.

Set specification

```
type T = int
class {:autocontracts} Set
{  ghost var elems: set<T>;
  constructor ()
    ensures elems == {}
  function  size(): nat
    ensures size() == |elems|
  method insert(x : T)
    requires x !in elems
    ensures elems == old(elems) + {x}
  method delete(x: T)
    requires x in elems
    ensures elems == old(elems) - {x}
  function  contains(x: T) : bool
    ensures contains(x) <==> x in elems
}
```

Set specification

```
type T = int
class {:autocontracts} Set
{  ghost var elems: set<T>;
  constructor ()
    ensures elems == {}
  function  size(): nat
    ensures size() == |elems|
  method insert(x : T)
    requires x !in elems
    ensures elems == old(elems) + {x}
  method delete(x: T)
    requires x in elems
    ensures elems == old(elems) - {x}
  function  contains(x: T) : bool
    ensures contains(x) <==> x in elems
}
```

Set implementation

```
  static const initialCapacity := 10;
  var list: array<T>;
  var used : nat;
  ghost predicate Valid()
  {
    used <= list.Length && list.Length >= initialCapacity
    && (forall i, j :: 0 <= i < j < used ==> list[i] != list[j])
```

```
  && |elems| == used && elems =(set x | x in multiset(list[..used]))
}
constructor ()
  ensures elems == {}
{
  list := new T[initialCapacity];
  used := 0;
  elems := {};
}


function size(): nat
  ensures size() == |elems|
{  used  }

method insert(x : T)
  requires x !in elems & used < list.Length
  ensures elems == old(elems) + {x}
{
  list[used] := x;
  used := used + 1;
  elems := elems + {x};
  }
```

Set implementation

```
method delete(x: T)
  requires x in elems
  ensures elems == old(elems) - {x}
{
  var i :| 0 <= i < used && list[i] == x;
  list[i] := list[used-1];
  used := used-1;
  elems := elems - {x};
}
function contains(x: T) : bool
  ensures contains(x) <==> x in elems
{
  exists i :: 0 <= i < used && list[i] == x
}
```

Lazily Inicializied Arrays

In general if we want to create an array and initialize all elements to a given element that can be achieved in linear time. But one can do it (lazily) in constant time with the additional structures.

Lazy Arrays Specification

```
class LazyArray<T(0)> {
 ghost var Elements: seq<T>
 ghost var Repr: set<object>
 ghost predicate Valid()
   reads this, Repr
   ensures Valid() ==> this in Repr
 constructor (length: nat, initial: T)
  ensures Valid() && fresh(Repr) && |Elements| == length
  ensures forall i :: 0 <= i < |Elements| ==> Elements[i] == initial
 function Get(i: int): T
  requires Valid() && 0 <= i < |Elements|
  reads Repr
  ensures Get(i) == Elements[i]
 method Update(i: int, x: T)
  requires Valid() && 0 <= i < |Elements|
  modifies Repr
  ensures Valid() && fresh(Repr - old(Repr)) && Elements == old(Elements)[i := x]
}
```

Lazy Arrays Implementation

- a is the array of type T
- we use two additional integers arrays b and c and an integer n such that
  - n number of used elements and also the number of used values of c
  - to record that $a[i]$ is set, the next free value of c stores i and the its index is stored in $b[i]$, i.e.,
  - $0 \leq b[i] < n$, $b[i] = j$ and $c[j] = i$

```
const default: T
const a: array<T>
const b: array<int>
const c: array<int>
var n: int

predicate IsInitialized(i: int)
  requires 0 <= i < |Elements| == b.Length == c.Length
  requires 0 <= n <= |Elements|
   reads this, b, c
  {
    0 <= b[i] < n && c[b[i]] == i
  }
```

Valid()

```
ghost predicate Valid()
   reads this, Repr
   ensures Valid() ==> this in Repr && N==|Elements|
   {
    this in Repr &&
   a in Repr && b in Repr && c in Repr &&
   a != b && b != c && c != a &&
    N==|Elements|==a.Length == b.Length == c.Length  && 0<= n<= N &&
(forall i :: 0 <= i < N ==> Elements[i] == if IsInitialized(i) then a[i]
  else default)
   }
```

Constructor()

```
constructor (length: nat, initial: T)
  ensures Valid() && fresh(Repr)
  ensures |Elements| == length
  ensures forall i :: 0 <= i < |Elements| ==>
   Elements[i] == initial
   {N := length;
   Elements := seq(length, _ => initial);
   default := initial;
   a, b, c := new T[length], new int[length], new int[length];
   n := 0;
   Repr := {this, a, b, c};
  }
```

Get()

```
function Get(i: int): T
  requires Valid() && 0 <= i < |Elements|
  reads Repr
  ensures Get(i) == Elements[i]
  {if IsInitialized(i) then a[i] else default
  }
```

Update()

```
 method  Update(i: int, x: T)
  requires Valid() && 0 <= i < |Elements|
  modifies Repr
```

```
 ensures Valid() && fresh(Repr - old(Repr))
 ensures Elements == old(Elements)[i := x]
  {
   if !IsInitialized(i) {
     b[i], c[n] := n, i; // index out of bounds (as n can be N),
     // one needs to be sure that there is room
    n :=  n + 1;
   }
   a[i] := x;
   Elements := Elements[i := x];
  }
```

This method does not verify!

Room()

```
lemma  Room(i:int)
requires Valid() && 0 <= i < N && !IsInitialized(i)
ensures n < N
{
n == |set j | 0<=j <N && IsInitialized(j)|< |set j | 0<=j < N| == N;
}
```

Does not work either!

- one needs to build the set of integers explicitly, and
- to know that a proprer subset of a set is smaller than the set

A set of k elements

```
ghost function Upto(k: nat): set<int>
 ensures forall i :: i in Upto(k) <==> 0 <= i < k
 ensures |Upto(k)| == k
{
if k == 0 then {} else Upto(k - 1) + {k - 1}
}
```

Set cardinalities

The set $u \subset U$ and $x \notin u$.

```
lemma SetCardinalities(u: set<int>, U: set<int>, x: int)
requires u <= U
ensures |u| <= |U|
```

```
ensures x !in u && x in U ==> |u| < |U|
{
if u == {} {
 } else {
  var y :| y in u;
  SetCardinalities(u - {y}, U - {y}, x);
 }
}
```

where `var y :| y in u;` assigns y a value of u.

Valid() again

```
 ghost var s: set<int>
 ghost predicate Valid()
   reads this, Repr
   ensures Valid() ==> this in Repr && N==|Elements|
   {
    this in Repr && a in Repr && b in Repr && c in Repr &&
   a != b && b != c && c != a && N==|Elements|==a.Length == b.Length == c.Length
    && 0<= n<= N &&
    (forall i :: 0 <= i < N ==>
 Elements[i] == if IsInitialized(i) then a[i] else default) &&
 s == (set i | 0 <= i < N && IsInitialized(i)) &&
    n == |s|
   }
```

One has to add in the constructor:

```
s,n := {},0;
```

and

```
lemma  Room(i:int)
requires Valid() && 0 <= i < N && !IsInitialized(i)
ensures n < N
{
    var S := Upto(N);
    SetCardinalities(s, S, i);
}
```

New Update()

```
method Update(i: int, x: T)
  requires Valid() && 0 <= i < |Elements|
  modifies Repr
 ensures Valid() && fresh(Repr - old(Repr))
  ensures Elements == old(Elements)[i := x]
  {
if !IsInitialized(i) {
    Room(i);
    b[i], c[n] := n, i;
    s, n :=  s+{i}, n + 1;
   }
   a[i] := x;
   Elements := Elements[i := x];
  }
```

Matching Triggers

To deal with universally quantified formulas solvers have to instantiate them. But of course there may exist to many terms for that.

For instance,

```
predicate isSorted(a: array<int>, from: nat, to: nat)
  requires 0 <= from <= to <= a.Length
  reads a
{
    forall i, j :: from <= i < j < to ==> a[i] <= a[j]
}
```

where here arrays are treated as functions.

Uninterpreted functions (first-order terms) are important for automated solvers. The solver keeps track of equivalent terms. Using *triggers* for a quantifier the solver looks to terms that match that trigger for instantiate.

In the above example, the triggers can be $a[i]$ and $a[j]$.

Matching Loop

However there is a problem when the arguments of the functions are other functions or arithmentic expressions.

For instance,

```
forall x: int:: 0 < x  ==> f(x) == f(x-1) + f(g(x))
```

- The trigger $f(g(x))$ will give rise to another instantiation of $g(x)$ (so it may be to discriminating)

- aritmetic expressions like $x - 1$ use interperted functions (subtraction) for which there are no information in the equivalent terms

:trigger

Dafny can suggest triggers to the SMT solver. In general expressions with aritmetic expressions are *trigger killers*. One can also mark a term as a trigger.

```
lemma {:axiom} PHoldEvenly()
  ensures  forall i (:trigger Q(i)} :: P(i) ==> P(i + 2) && Q(i)
lemma PHoldsForTwo()
  ensures forall i :: P(i) ==> P(i + 4)
{}
```

To prove PHoldsForTwo one needs to use universal introduction:

```
forall k | P(k)
  ensures Q(k) {}
```

```
lemma PHoldsForTwo()
  ensures forall i :: P(i) ==> P(i + 4)
{
   forall j: int | P(j)
      ensures P(j+4)
  {
    if P(j) {
    PHoldEvenly();
    assert Q(j);
    assert P(j + 2);
    assert P(j + 4); // fails if the trigger is Q(i)
     }
 }
```

In this case, if the trigger annoation is removed Dafny finds a correct one.

For more: `https://dafny.org/latest/DafnyRef/`