Lectures Notes on
Dynamic State and ADTs

Construction and Verification of Software
FCT-NOVA
Bernardo Toninho

11 April, 2023

Section 2 draws heavily from [HLQ11].

1 Framing – Hoare Logic and Dafny

A key component of *Hoare Logic* is a way to manage state change in a compositional way. To this effect, we include in Hoare logic the frame rule:

$$\frac{\{A\}\ P\ \{B\}}{\{A \wedge C\}\ P\ \{B \wedge C\}} \quad \text{P does not modify } vars(C)$$

This rule, while not primitive to the system, captures the fact that all assertions in the pre-condition of a program P that refer to state that is not modified by P are still valid as a post-condition.

Framing allows the reasoner to preserve as much knowledge as possible by tracking the individual changes to state variables. Identifying state changes in a sequence of assignments, with no dynamic allocation of memory and no aliasing, is a straightforward process. When the language includes abstraction mechanisms like functions or methods we need the state changes to be listed explicitly in the signature of such methods. Languages like Dafny provide mechanisms for method signatures to state what parts of an object's state and method parameters are *modified* by the method. For example the **`modifies`** clause below indicates that the method modifies the field `bal`:

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

The dual annotation (**`reads`**) is necessary to determine what part of the state is read by a given program. Thus, functions need to declare what part of the state they rely on, so that the frame rule can be applied when they are used in pre-conditions and post-conditions.

Another case of controlling state changes concerns the use of arrays. It is often necessary to use post-conditions and loop invariants that ensure that a part of the array was not changed. For this, the **`old`** keyword is helpful when referring to the value of a given expression before the method call.

```
ensures ∀ k • 0 ≤ k < pos ⟹ a[k] = old(a[k])
```

See the lecture notes on loop invariants for sorting for illustrative examples of the use of this pattern. Also, the example of the `ASet` ADT, with abstract state, given in appendix to these notes requires a lemma (an intermediate proof) in the form of an **`assert`** statement in method `add` to be able to prove the post-condition. This lemma states that all existing positions of the array are kept unchanged.

A third mechanism that is needed for the frame rule is the declaration of newly allocated memory by means of function **`fresh`**. See below an example of a method that creates a new array:

```
method Grow() returns (na:array<int>)
    requires RepInv()
    ensures size < na.Length
    ensures fresh(na)
    ensures ∀ 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 ∀ k • (0≤k<i) ⟹ na[k] = a[k];
    {
        na[i] := a[i];
        i := i + 1;
    }
}
```

2 Framing and Abstraction

One of the key features of ADTs is abstraction through information hiding. When designing the interface of an ADT, we aim to ensure that its internal representation does not "leak" into the interface operations. By enabling clients of the ADT to be agnostic to the concrete implementation, which enables modularity and interchangeability of software components.

At the level of ADT specifications, similar concerns arise: we avoid the explicit use of concrete state in specifications so as to maintain the representation and the abstract state separate. A general way of achieving this is to explicitly maintain the abstract state as ghost state and/or provide specification-level operations that provide a level of indirection when accessing concrete state. For instance, let us consider a Counter ADT, providing increment decrement and query operations. The abstract state is best modelled using a ghost variable that encodes the counter value, hiding whatever concrete state representation is chosen. The consistency between the abstract and concrete state representation is maintained by the abstraction invariant. For an implementation of the Counter ADT using an integer count of increments and decrements, the invariant is that the abstract counter value is the difference between the number of increments and decrements:

```
class Counter
{
    //Concrete state
    var incs:int;
```

```
    var decs: int;

  //Abstract state
    ghost var Value : int;

    predicate Valid()
        reads this
    {
        Value = incs - decs
    }

    ...

}
```

The constructor establishes the invariant, initializing the abstract state accordingly. Moreover, the constructor further ensures that the initial value of the counter is 0:

```
    constructor ()
        ensures Valid()
        ensures Value = 0
    {
        incs, decs, Value := 0, 0, 0;
    }
```

The counter operations both maintain the invariant and express the effect of the operations in terms of the abstract state:

```
    method getValue() returns (x: int)
        requires Valid()
        ensures x = Value
    {
        x := incs - decs;
    }

    method inc()
        modifies this
        requires Valid()
        ensures Valid()
        ensures Value = old(Value) + 1
    {
        incs, Value := incs + 1, Value + 1;
    }

    method dec()
        modifies this
        requires Valid()
        ensures Valid()
        ensures Value = old(Value) - 1
    {
        decs, Value := decs + 1, Value - 1;
    }
```

There is, however, a tension between the information hiding principle of ADTs and the framing specifications required to reason about programs that manipulate dynamic memory. In the example above, the state used by the Counter is completely static (two integer variables) and so the **reads** and **modifies** clauses need only mention **this**.

Let us now consider a Counter that relies on dynamically allocated (private) memory cells:

```
class Cell
{
    var data: int;

    constructor (n: int)
        ensures this.data = n
    {
        this.data := n;
    }
}
```

The counter now has two fields of type `Cell`:

```
class CounterCell
{
    var incs: Cell;
    var decs: Cell;


    ghost var Value : int;

    predicate Valid()
        reads this, incs, decs
    {
        incs ≠ decs ∧ Value = incs.data - decs.data
    }
    ...
}
```

Since the invariant accesses the fields of the two memory cells, we must explicitly list `incs` and `decs` in the **reads** clause of the predicate. Similarly, the `inc` and `dec` methods must indicate they modify the corresponding field:

```
    method inc()
        requires Valid()
        modifies this
        ensures Valid()
        ensures Value = old(Value) + 1
        modifies incs
    {
        incs.data, Value := incs.data + 1, Value + 1;
    }
```

Unfortunately, this specification does not allow clients of the ADT to effectively use it. The following usage fails to verify the invocation of `inc`:

```
method Main()
```

```
{
    var c := new CounterCell();
    c.inc();
}
```

The issue is that the specification of the constructor does not provide enough information about the `incs` cell and so the verifier cannot guarantee that the call to `inc()` does not affect state that it should not. We can fix this by strengthening the post-condition of the constructor to ensure that the two memory cells are created by the counter:

```
constructor ()
    ensures Valid()
    ensures Value = 0
    ensures fresh(incs) ∧ fresh(decs)
{
    incs := new Cell(0);
    decs := new Cell(0);
    Value := 0;
}
```

Now, the call to `inc` above goes through, but any subsequent calls to `inc` and `dec` fail. The reason is that after the first call to `inc`, we have no way to know that the two `Cell` objects referred to by the counter have not been changed (e.g. to other cells that are "owned" by some other object). We can address this issue by appropriately framing the `inc` and `dec` operations, ensuring that the two references to the cells are unchanged.

```
method inc()
    requires Valid()
    modifies this
    ensures Valid()
    ensures Value = old(Value) + 1
    modifies incs
    ensures incs = old(incs)
    ensures decs = old(decs)
{
    incs.data, Value := incs.data + 1, Value + 1;
}
```

While this version of the Counter can be used in the correct way, its specification essentially completely leaks the internal representation and fundamentally breaks the information hiding principle of ADTs. To reconcile dynamic memory with information hiding, we use a Dafny idiom based on *dynamic frames* [Kas06].

2.1 Dynamic Frames in Dafny

To abstract over dynamically allocated fields in specifications we add to the ADT implementation we are specifying a piece of ghost state that consists of the set (of objects) that our ADT may read or write, encoding the ADTs memory *frame* or *footprint*. Unlike in the specification of the Counter from the previous section, this frame is *dynamic* insofar as it may evaluate to different sets of objects, depending on the program state, whereas in our previous attempt we used a statically determined listing of objects.

The first step is to introduce a ghost variable which will encode the set of objects in the given object's representation: `ghost var Repr: set<object>;` we use the dafny set type, instantiated at type **`object`**. For the Counter, the set will consist of the objects **`this`**, `incs` and `decs`.

Now, the modifier methods need only list `Repr` in their modifies clause, rather than the concrete representation field. To be precise, we are stating that the modifier methods are justified in modifying the fields of any object that, at the time the method is invoked, is in the set `Repr`.

The Counter invariant must now characterize the contents of the frame accordingly:

```
predicate Valid()
    reads this, Repr
{
    this in Repr ∧
    incs in Repr ∧ decs in Repr ∧
    incs ≠ decs ∧ Value = incs.data - decs.data
}
```

Note the **`reads`** clause for the invariant. We must list both **`this`** and `Repr`, even though it would seem `Repr` would suffice, because when checking the body of `Valid` the verifier makes no assumptions about `Repr`. Specifically, to verify the first conjunct the verifier must know that **`this`** is allowed by the frame.

The specification of the constructor now becomes:

```
constructor ()
    ensures Valid() ∧ fresh(Repr - {this})
    ensures Value = 0
{ ... }
```

The postcondition states that when the constructor returns, all objects in `Repr` were allocated after the constructor invocation, with the exception of **`this`**. This is achieves our goal of maintaining the specification abstract (we need not mention the fields directly) and allowing calls to modifier methods declared with **`modifies`** `Repr`, since newly allocated objects in the constructor are also newly allocated in the calling context, and so the caller is allowed to modify objects contained in `Repr`. We can generally add a similar post-condition to the two modifier methods: **`ensures fresh`**`(Repr - old(Repr))`, stating that any extra objects in `Repr` are also newly allocated.

In more sophisticated scenarios, for instance where objects are deeply or dynamically nested (e.g. an object with an object array field), it is often necessary to use additional or other constraints on `Repr`. A common scenario arises while looping over an array that is contained in `Repr`. In many such situations it will be necessary to state that both `Repr` and the reference to the array are unchanged between loop iterations.

3 Typestates

So far, we have designed ADTs to explicitly have as pre- and post-conditions some Invariant. Specifically, we use a representation invariant restricting the values of the representation type that form a valid ADT value and identifying the state "owned" by the ADT (see Section 2.1), and an abstract invariant that maps the valid states of an ADT to an observable layer. To achieve this in a modular way we use ghost variables to represent the (observable) state of an ADT, while hiding the concrete representation state:

```
method add(v: int)
```

```
    requires Valid() ∧ size() < maxSize()
    ensures Valid() ∧ s = old(s) + {v}
```

This post-condition allows a client module to determine the exact elements of the ADT, without having to know the internal structure of the ADT. Notice that the pre-condition uses functions `size` and `maxSize` to determine the legal states where the function can be called.

4 Refining ADT States

The condition to apply method `add` above is to have enough space to store another element. This can be easily abstracted by a refined state called `NotFull`.

Consider now a general resource, with the following interface:

```
class Resource {

    predicate OpenState()

    predicate ClosedState()

    constructor()
    ensures  ClosedState();

    method Use(K:int)
    requires OpenState()
    ensures  OpenState()

    method Open()
    requires ClosedState()
    ensures  OpenState()

    method Open()
    requires OpenState()
    ensures ClosedState()
}
```

Notice the two predicates that denote two possible states for the resource. We dub these as *typestates*.

We also enforce that all methods and constructor(s) have as pre-conditions and post-conditions the allowed start and end typestates. With this idiom we can verify that the following usage of the resource

```
method UsingTheResourceOk()
{
    var r:Resource := new Resource();
    r.Open();
    r.Use(2);
    r.Use(9);
    r.Close();
}
```

is legal according to the valid state changes of the ADT, and that the client code below is not:

```
1  method UsingTheResourceNotOk()
2  {
3      var r:Resource := new Resource();
4      r.Close();
5      r.Open();
6      r.Use(2);
7      r.Use(9);
8      r.Close();
9      r.Use(2);
10 }
```

Notice that the call on line 4 is performed in a state where the resource is closed, thus not satisfying its pre-condition. Also, the call on line 9 is done after the resource has been closed, which also does not satisfy the appropriate pre-conditions.

To ensure the correct functioning of the ADT, the predicates defining the typestates imply the representation invariant of the ADT (or include it directly). The set of type states thus forms the abstract state of the ADT, which may also refer to ghost values.

5 Exercises

1. Given the leaky specification of class `Set` found in Appendix **??**, use the techniques from class (the use of ghost state and dynamic frames) so that the specification no longer leaks the internal representation. Produce client code that correctly connects to your revised `Set` class.

2. Using the corrected version of `Set` as a baseline, implement a `PositiveSet` class that enforces the invariant that all numbers in the set are strictly positive.

3. Implement the bank account described in the Savings Account appendix (see below) with a pair of balances. One for the savings balance and another for the checking balance.

4. Change your specification and implementation of the `ASet` ADT to include a growing array of integer values.

References

[HLQ11] Luke Herbert, K. Rustan M. Leino, and Jose Quaresma. Using dafny, an automatic program verifier. In Bertrand Meyer and Martin Nordio, editors, *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, volume 7682 of *Lecture Notes in Computer Science*, pages 156–181. Springer, 2011.

[Kas06] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2006.

A Dafny source code

A.1 Set with Leaky Spec

```
class Set {

    var store:array<int>;
    var nelems: int;

    ghost predicate RepInv()
    reads `store,store,`nelems
    {
        0 < store.Length
        && 0 <= nelems <= store.Length
        && forall i,j :: 0 <= i < j < nelems ==> store[i] != store[j]
    }

    // the construction operation
    constructor(n: int)
    requires 0 < n
    ensures RepInv()
    ensures fresh(store)

    {
        store := new int[n];
        nelems := 0;
    }

    // returns the number of elements in the set
    function size():int
        requires RepInv()
        ensures RepInv()
        reads `store,store,`nelems
    { nelems }

    // returns the maximum number of elements in the set
    function maxSize():int
        requires RepInv()
        ensures RepInv()
        reads `store,store,`nelems
    { store.Length }

    // checks if the element given is in the set
    method contains(v:int) returns (b:bool)
        requires RepInv()
        ensures RepInv()
        ensures b <==>
            exists j::(0 <= j < nelems) && v == store[j]
    {
```

```
    var i := find(v);
    return i >= 0;
}

// adds a new element to the set if space available
method add(v:int)
    requires RepInv()
    requires size() < maxSize()
    ensures RepInv()
    modifies store,`nelems
{
    var f:int := find(v);
    if (f < 0) {
        store[nelems] := v;
        nelems := nelems + 1;
    }
}

// private method that should not be in the
method find(x:int) returns (r:int)
    requires RepInv()
    ensures RepInv()
    ensures r < 0 ==> forall j::(0<=j<nelems) ==> x != store[j];
    ensures r >=0 ==> r < nelems && store[r] == x;
{
    var i:int := 0;
    while (i<nelems)
        decreases nelems-i
        invariant 0 <= i <= nelems;
        invariant forall j::(0<=j< i) ==> x != store[j];
    {
        if (store[i]==x) { return i; }
        i := i + 1;
    }
    return -1;
}

method Main()

 {
    var s := new Set(10);
    if (s.size() < s.maxSize()) {
         s.add(2);
         var b := s.contains(2);
         if (s.size() < s.maxSize()) {
             s.add(3);
         }
    }
```

```
    }
}
```

A.2 Savings Account

```
/*
 * Implement a savings account.
 * A savings account is actually made up of two balances.
 *
 * One is the checking balance, here account owner can deposit and withdraw
 * money at will. There is only one restriction on withdrawing. In a regular
 * bank account, the account owner can make withdrawals as long as he has the
 * balance for it, i.e., the user cannot withdraw more money than the user has.
 * In a savings account, the checking balance can go negative as long as it does
 * not surpass half of what is saved in the savings balance. Consider the
 * following example:
 *
 * Savings = 10
 * Checking = 0
 * Operation 1: withdraw 10      This operation is not valid. Given that the
 *                               the user only has $$10, his checking account
 *                               can only decrease down to $$-5 (10/2).
 *
 * Operation 2: withdraw 2        Despite the fact that the checking balance of
 *                                                          the user is zero,
 *                               money in his savings account, therefore, this
 *                               operation is valid, and the result would be
 *                                                          something like:
 * Savings = 10;
 * Checking = -2
 *
 * Regarding depositing money in the savings balance (save), this operation has
 * one small restrictions. It is only possible to save money to the savings
 * balance when the user is not in debt; i.e. to save money into savings, the
 * checking must be non-negative.
 *
 * Given the states:
 * STATE 1                                  STATE 2
 * Savings = 10                             Savings = 10
 * Checking = -5                            Checking = 0
 *
 * and the operation save($$60000000000), the operation is valid when executed
 * in STATE 2 but not in STATE 1.
 *
 * Finally, when withdrawing from the savings balance, an operation we will
 * call rescue, the amount the user can withdraw depends on the negativity of
 * the user's checking account. For instance:
```

```
 *
 * Savings: 12
 * Checking: -5
 *
 * In the case, the user could withdraw at most two double dollars ($$). If the
 * user withdrew more than that, the balance of the checking account would
 * go beyond the -50% of the savings account; big no no.
 *
 */

class SavingsAccount {

  constructor() {}

  method deposit(amount:int){}

  method withdraw(amount:int){}

  method save(amount: int){}

  method rescue(amount: int){}

}
```