More on Proofs

Case study - quicksort

Andrei Arusoaie

Outline

Goal

- Understand how to approach program verification
- Here we use Dafny, but some principles apply in general
- The focus is to develop a systematic way to develop a formal proof
- How to tackle situations when it is difficult to find why the verification does not work

Steps

It is difficult to find the general recipe for success when proving programs correct, but we can at least identify some steps that may help with that:

1. Understand the problem domain (algorithm, corner cases, ...)
2. Choose the right abstractions
3. Write informal specifications (pre/postconditions)
4. Identify loop invariants
5. Encode, prove and debug incrementally

In this lecture ...

1. We attempt to follow the previous steps for quicksort
2. We recall several things about Dafny as well
3. We provide an informal description to understand quicksort
4. We gradually choose the abstractions, write pre/postconditions, invariants
5. We encode, prove and debug everything in Dafny

Some background: arrays in Dafny

- Arrays are reference types with fixed length
- `a.Length`: the length of array `a`
- `a[i]`: accessing element at index `i`
- `a[lo..hi]`: slice from index `lo` (inclusive) to `hi` (exclusive)
- `a[..]`: entire array as a sequence

Note that arrays are mutable, sequences are immutable

`modifies` and `reads` clauses

`modifies` clause:

- Specifies which heap locations a method may modify
- `modifies a` means the method can change array `a`

`reads` clause:

- Specifies which heap locations a function reads
- Important for pure functions (no side effects)
- `reads a` means the function may read from array `a`

ghost code in Dafny

- Code that exists *only for verification*
- Completely erased during compilation
- Has no runtime cost
- Cannot affect non-ghost (compiled) code

- Example: ghost variables used to save state for later reasoning

```
1 ghost var beforeLeft := a[..];  // snapshot
```

Pre and Postconditions

Preconditions (`requires`):

- What must be true before the method executes
- Contract between caller and callee
- Caller's responsibility to ensure

Postconditions (`ensures`):

- What the method guarantees after execution
- Method's responsibility to ensure
- Can refer to `old(x)` for values before execution

Quicksort: overview

Divide and conquer sorting algorithm:

1. Partition: Choose a pivot element
2. Rearrange array so:
   - Elements $\leq$ pivot are on the left
   - Elements $>$ pivot are on the right
3. Recursively sort: left and right subarrays
4. Base case: arrays of size ≤ 1 are already sorted

Remark: After partition, pivot is in its final position!

Quicksort Visualization

Pick 7 as the pivot (last element):

Start:

5	9	8	3	2	1	7

One partition step:

5	3	2	1	7	9	8

pivot

The pivot can be any element!

The Partition Method

Goal: Rearrange array section so all elements $\leq$ pivot come before elements $\geq$ pivot

Approach:

- Choose last element as pivot
- Maintain two regions:
  - $[lo, i)$: elements $\leq$ pivot
  - $[i, j)$: elements $>$ pivot
- Scan through the array with index j
- When we find element $\leq$ pivot, swap it to the left region
- Finally, place pivot in its correct position

Partition Example

Array: $[5, 9, 8, 3, 2, 1, 7]$, pivot $= 7$

Initial:	5	9	8	3	2	1	7	$i = 0, j = 0$
	5	9	8	3	2	1	7	$i = 1, j = 1$
	5	9	8	3	2	1	7	$i = 1, j = 2$
	5	9	8	3	2	1	7	$i = 1, j = 3$
	5	3	8	9	2	1	7	$i = 2, j = 4$
	5	3	2	9	8	1	7	$i = 3, j = 5$
	5	3	2	1	8	9	7	$i = 4, j = 6$

Final step: Swap pivot with position $i \rightarrow [5, 3, 2, 1, 7, 9, 8]$

What do we prove for Quicksort?

For quicksort to be correct, we need to show:

1. **Sortedness:** The output array is sorted

$$\forall k \in [0, n-1).\ a[k] \leq a[k+1]$$

What do we prove for Quicksort?

For quicksort to be correct, we need to show:

1. **Sortedness:** The output array is sorted

$$\forall k \in [0, n-1).\ a[k] \leq a[k+1]$$

2. **Permutation:** The output contains exactly the same elements as the input

$$\text{multiset}(a[..]) = \text{multiset}(\text{old}(a[..]))$$

Remarks: useful for verification purposes!

Since quicksort works on subarrays, we may want to add frames:

1. **Sortedness:** The output array is sorted between *lo* and *hi*

$$\forall k \in [lo, hi - 1).\ a[k] \leq a[k + 1]$$

Remarks: useful for verification purposes!

Since quicksort works on subarrays, we may want to add frames:

1. **Sortedness:** The output array is sorted between lo and hi

$$\forall k \in [lo, hi - 1).\ a[k] \leq a[k + 1]$$

2. **Frame condition:** Elements outside $[lo, hi)$ are unchanged

$$\forall k.\ (k < lo \vee k \geq hi) \Rightarrow a[k] = \mathsf{old}(a[k])$$

Quicksort Specification (Tentative)

Preconditions:

$$0 \leq lo \leq hi \leq |a|$$

Postconditions:

$$\forall k \in [lo, hi - 1).\ a[k] \leq a[k + 1]$$
$$\forall k.\ (k < lo \vee k \geq hi) \Rightarrow a[k] = \text{old}(a[k])$$
$$\text{multiset}(a[..]) = \text{multiset}(\text{old}(a[..]))$$

Termination: Need to prove recursion terminates

- `decreases`: $hi - lo$ (size of subarray)
- Each recursive call operates on a strictly smaller subarray

Partition Specification (Tentative)

Preconditions:

$$0 \leq lo < hi \leq |a|$$

Postconditions:

$$lo \leq \text{pivotIndex} < hi$$
$$\forall k \in [lo, \text{pivotIndex}).\ a[k] \leq a[\text{pivotIndex}]$$
$$\forall k \in [\text{pivotIndex}, hi).\ a[k] \geq a[\text{pivotIndex}]$$
$$\forall k.\ (k < lo \vee k \geq hi) \Rightarrow a[k] = \text{old}(a[k])$$
$$\text{multiset}(a) = \text{multiset}(\text{old}(a))$$

The partitioning creates two regions with all elements on the left $\leq$ pivot and all on the right $\geq$ pivot

Loop Invariants for Partition

The partition loop needs invariants to maintain correctness:

$$0 \leq lo \leq i \leq j < hi \leq |a|$$
$$\forall k \in [lo, i).\ a[k] \leq \text{pivot}$$
$$\forall k \in [i, j).\ a[k] > \text{pivot}$$
$$a[hi - 1] = \text{pivot}$$
$$\forall k.\ (k < lo \vee k \geq hi) \Rightarrow a[k] = \text{old}(a[k])$$
$$\text{multiset}(a) = \text{multiset}(\text{old}(a))$$

These invariants:

- Hold initially
- Are preserved by each iteration
- Imply postcondition when loop exits

Partition Specification in Dafny

```
1  method partition(a: array<int>,
2                   lo: int, hi: int)
3   returns (pivotIndex: int)
4   modifies a
5   requires 0 <= lo < hi <= a.Length
6   ensures lo <= pivotIndex < hi
7   ensures forall k :: lo <= k < pivotIndex
8     ==> a[k] <= a[pivotIndex]
9   ensures forall k :: pivotIndex <= k < hi
10    ==> a[k] >= a[pivotIndex]
11  ensures forall k :: 0 <= k < lo || hi <= k < a.
        Length
12    ==> a[k] == old(a[k])
13  ensures multiset(a[..]) == multiset(old(a[..]))
```

Dafny Syntax Notes

- `::` separates quantifier variable from body
- `==>` is logical implication
- `old(a[k])` refers to value before method execution
- `a[..]` converts array to sequence
- `multiset(s)` converts sequence to multiset
- `||` is logical OR, `&&` is logical AND

Quicksort Specification in Dafny

```
1  method qs(a: array<int>, lo: int, hi: int)
2   modifies a
3   requires 0 <= lo <= hi <= a.Length
4   ensures forall k :: lo <= k < hi - 1
5      ==> a[k] <= a[k + 1]
6   ensures forall k :: 0 <= k < lo || hi <= k < a.
       Length
7      ==> a[k] == old(a[k])
8   ensures multiset(a[..]) == multiset(old(a[..]))
9   decreases hi - lo
```

For termination: `decreases hi - lo`

- The expression `hi - lo` decreases with each recursive call
- Must be bounded below (here, by 0)

Partition Implementation (Part 1)

```
1  method partition(a: array<int>, lo: int, hi: int)
2    returns (pivotIndex: int)
3    modifies a
4    requires 0 <= lo < hi <= a.Length
5    ensures lo <= pivotIndex < hi
6    ensures forall k :: lo <= k < pivotIndex
7      ==> a[k] <= a[pivotIndex]
8    ensures forall k :: pivotIndex <= k < hi
9      ==> a[k] >= a[pivotIndex]
10   ensures forall k :: 0 <= k < lo || hi <= k < a.Length
11     ==> a[k] == old(a[k])
12   ensures multiset(a[..]) == multiset(old(a[..]))
13 {
14   var pivot := a[hi - 1];
15   var i := lo;
16   var j := lo;
17   // continued...
```

Partition Implementation (Part 2)

```
1   while j < hi - 1
2     invariant 0 <= lo <= i <= j < hi <= a.Length
3     invariant forall k :: lo <= k < i ==> a[k] <= pivot
4     invariant forall k :: i <= k < j ==> a[k] >= pivot
5     invariant a[hi - 1] == pivot
6     invariant forall k :: 0 <= k < lo || hi <= k < a.Length
7       ==> a[k] == old(a[k])
8     invariant multiset(a[..]) == multiset(old(a[..]))
9   {
10    if a[j] <= pivot {
11      a[i], a[j] := a[j], a[i];
12      i := i + 1;
13    }
14    j := j + 1;
15  }
16
17  a[i], a[hi - 1] := a[hi - 1], a[i];
18  pivotIndex := i;
19 }
```

Invariants - I

Bounds invariant: `0 <= lo <= i <= j < hi <= a.Length`

- Ensures array accesses are safe

Left region invariant:

```
forall k ::  lo <= k < i ==> a[k] <= pivot
```

- All elements $\leq$ pivot are moved to left region
- This is important for for partition postcondition

Right region invariant:

```
forall k ::  i <= k < j ==> a[k] >= pivot
```

- Elements between i and j are $>$ pivot

Invariants - II

Pivot invariant: `a[hi - 1] == pivot`

- Pivot stays in last position during loop
- Needed for final swap to place pivot correctly

Frame invariant:

```
forall k ::  0 <= k < lo || hi <= k < a.Length ==>
a[k] == old(a[k])
```

- Elements outside $[lo, hi)$ are unchanged

Multiset invariant:

```
multiset(a[..])  == multiset(old(a[..]))
```

- We only swap, never create/destroy elements
- Proves permutation property

The Subset Preservation Lemma

Problem: After recursion, we need to know the subarray still contains the same elements

```
1  lemma subsetPreserved(a: array<int>, lo: int, hi: int,
2                        oldA: seq<int>)
3    requires 0 <= lo <= hi <= a.Length
4    requires |oldA| == a.Length
5    requires multiset(a[..]) == multiset(oldA)
6    requires forall k :: 0 <= k < lo || hi <= k < a.Length
7      ==> a[k] == oldA[k]
8    ensures multiset(a[lo..hi]) == multiset(oldA[lo..hi])
```

Idea: If entire array has same multiset and elements outside $[lo, hi)$ are unchanged, then $[lo, hi)$ must have same multiset

Lemma Proof Strategy

```
1  {
2    // Decompose sequences
3    assert a[..] == a[0..lo] + a[lo..hi] + a[hi..a.Length];
4    assert oldA == oldA[0..lo] + oldA[lo..hi] + oldA[hi..|oldA|];
5
6    // Unchanged parts are equal
7    assert a[0..lo] == oldA[0..lo];
8    assert a[hi..a.Length] == oldA[hi..|oldA|];
9
10   // Multiset of concatenation is union of multisets
11   assert multiset(a[..]) ==
12     multiset(a[0..lo]) + multiset(a[lo..hi]) +
13     multiset(a[hi..a.Length]);
14   // Similar for oldA...
```

Hint: Use multiset arithmetic to isolate the changed region

Lemma Proof (Conclusion)

```
1   calc {
2     multiset(a[lo..hi]);
3     ==
4     multiset(a[..]) - multiset(a[0..lo])
5                     - multiset(a[hi..a.Length]);
6     ==
7     multiset(oldA) - multiset(a[0..lo])
8                    - multiset(a[hi..a.Length]);
9     ==
10    multiset(oldA) - multiset(oldA[0..lo])
11                   - multiset(oldA[hi..|oldA|]);
12    ==
13    multiset(oldA[lo..hi]);
14  }
15 }
```

`calc:` a chain of equalities, each step must be provable

Quicksort (Part 1)

```
1  method qs(a : array<int>, lo:int, hi:int)
2    modifies a
3    requires 0 <= lo <= hi <= a.Length
4    ensures forall k :: lo <= k < hi - 1 ==> a[k] <= a[k + 1]
5    ensures forall k :: 0 <= k < lo || hi <= k < a.Length
6      ==> a[k] == old(a[k])
7    ensures multiset(a[..]) == multiset(old(a[..]))
8    decreases hi - lo
9  {
10   if lo + 1 >= hi {
11     return;
12   }
13
14   // Partition step
15   var pivotIndex := partition(a, lo, hi);
```

Quicksort (Part 2)

```
1   ghost var pivotValue := a[pivotIndex];
2   assert forall x :: x in multiset(a[lo..pivotIndex])
3     ==> x <= a[pivotIndex];
4   assert forall x :: x in multiset(a[pivotIndex+1..hi])
5     ==> x >= pivotValue;
6
7   // Left recursion
8   ghost var beforeLeft := a[..];
9   qs(a, lo, pivotIndex);
10  subsetPreserved(a, lo, pivotIndex, beforeLeft);
11  assert forall k :: lo <= k < pivotIndex
12    ==> a[k] in multiset(a[lo..pivotIndex]);
13  assert forall k :: lo <= k < pivotIndex
14    ==> a[k] <= a[k + 1];
```

Ghost variables: Exist only for verification, erased at runtime

Quicksort (Part 3)

```
1    // Right recursion
2    ghost var beforeRight := a[..];
3    qs(a, pivotIndex + 1, hi);
4    subsetPreserved(a, pivotIndex + 1, hi, beforeRight);
5    assert forall k :: pivotIndex + 1 <= k < hi
6      ==> a[k] in multiset(a[pivotIndex+1..hi]);
7    assert forall k :: pivotIndex + 1 <= k < hi - 1
8      ==> a[k] <= a[k + 1];
9  }
```

Why we need the lemma

Problem: After recursion, Dafny doesn't automatically know:

- The sorted subarray contains the same elements as before recursion
- We can connect properties of original elements to sorted elements

Solution: Call `subsetPreserved` lemma

- Proves

  $$\text{multiset}(a[lo..pivotIndex]) = \text{multiset}(beforeLeft[lo..pivotIndex])$$

- Combined with partition postcondition, shows all elements in left are $\leq$ pivot
- Similar reasoning for right side

The Final Step

After both recursive calls:

- Left subarray: Sorted, all elements $\leq$ pivot
- Pivot: In position `pivotIndex`, unchanged
- Right subarray: Sorted, all elements $\geq$ pivot

Therefore: Entire array $[lo, hi)$ is sorted!

- Within left: sorted by induction
- Within right: sorted by induction
- Between left and pivot: elements $\leq$ pivot $\leq$ pivot
- Between pivot and right: pivot $\leq$ elements $\geq$ pivot

Top-Level Quicksort

```
1  method quickSort(a: array<int>)
2   modifies a
3   ensures forall k :: 0 <= k < a.Length - 1
4     ==> a[k] <= a[k + 1]
5   ensures multiset(a[..]) == multiset(old(a[..]))
6  {
7    if a.Length > 0 {
8      qs(a, 0, a.Length);
9    }
10 }
```

Simple wrapper that:

- Handles empty array edge case
- Calls `qs` on entire array
- Postconditions follow directly from `qs`
- Note that these postconditions are precisely the ones that we aimed for from the beginning!

Lessons for Verification

1. Start with specification: Know what you're proving before coding
2. Think about invariants: What's true at each step?
3. Use the right abstractions: Multisets for permutations, sequences for order
4. Break down complexity: Helper lemmas for non-obvious facts
5. Be explicit: Assertions help the prover
6. Frame your changes: Always specify what stays the same

Conclusion

We proved:

- Quicksort correctly sorts the array
- No elements are lost or created
- Elements outside the sorted region are unchanged
- The algorithm terminates

This is the power of formal verification!

Questions?

Thank you!