Dafny Searching

Markus Roggenbach

November 2025

What you are going to learn

- Ingredients for Program Verification
- Two Algorithms proven to be correct
- Loop Invariants are Essential
- Yeah – Dafny can Deal with Quantifiers

Recap – Contract in Dafny

Informal Semantics of a Dafny Contract

Let

```
method m  (x:int) returns  (result:int)
  requires E(x)
  ensures G(x,result)
{...}
```

be a method such that Dafny proved the guarantee `G(x,result)` to be valid (i.e., shows a green line) under the expectation that `G(x)` holds for the actual parameter `x`.

Then, for all actual parameters that fulfill `E(x)`, after execution of method `m`, guarantee `G(x,result)` holds.

The lines

```
  requires E(x)
  ensures G(x,result)
```

are called the *contract* of method `m`.

Design by Contract

Design by contract (DbC), also known as contract programming, programming by contract and design-by-contract programming, is an approach for designing software.

The term was coined by Bertrand Meyer in connection with his design of the Eiffel programming language and first described in 1986.

If a method of a class provides a certain functionality, it may:

- Expect a certain condition to be guaranteed on entry by any client module that calls it.
- Guarantee a certain property on exit.

Freely following Wikipedia

Turning the Multiply Problem into Dafny Dode

Multiply
Input: integers x, y
Output: integer x * y

The method declaration takes care of the *type information:*

```
method mult (x: int, y:int) returns (r: int)
```

The contract takes care of *expectations & guarantees:*

```
requires true
ensures result == x * y
```

Note: `true` stands for "there are no expectations on the input"

Proving Correctness with Dafny

```
11  method Mult1 (x: int, y: int) returns (result: int)
12      requires true
13      ensures result == x * y
14  {
15       result := x * (y - 1) + x;
16  }
```

Recap – Loop Invariants

Invariant

The word *invariant* means “not changing”

Here: a predicate concerning a loop, that

- holds before the body of the loop is executed and
- after the body of the loop is executed.

I.e., a predicate whose validity does not change “troughout” the loop.

Loop Invariant – formal

Definition

Consider the following piece of Dafny code:

```
while B
   invariant J
{
   Body
}
```

The predicate `J` is an *(loop) invariant* iff

```
(B && J) ==> WP [[ Body, J]]
```

Design Principles for Loop Invariants

Invariant design principle: Express relations (which value is larger than another one) as an invariant.

Invariant design principle: State the intermediate result computed in the i-th run of the loop.

Algorithms

Algorithm

An algorithm is “a tool for solving a well-specified computational problem. The statement of the problem specifies in general terms the desired input/output relationship.”

[THOMAS H. CORMEN CHARLES E. LEISERSON RONALD L. RIVEST CLIFFORD STEIN, INTRODUCTION TO ALGORITHMS, MIT Press]

Example: Sorting Problem as a Computational Problem

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

[THOMAS H. CORMEN CHARLES E. LEISERSON RONALD L. RIVEST CLIFFORD STEIN, INTRODUCTION TO ALGORITHMS, MIT Press]

Our Program of Study

Given: a Computational Problem in natural language

Activities:

Model: the Problem as a Dafny Contract
Take: a program in Dafny supposed to implement the contract
Annotate: the program till Dafny says things are fine.

Outcome: Dafny program that has been proven to be correct w.r.t. the given contract

There always is the Question of Trust

Possible sources of error:

1. *Modelling of the problem as a contract might go wrong*
   - Likelihood: "medium" [an experienced software engineer should be able to do this well]
   - Consequences: "catastrophic" [if we write down the wrong contract, we have proved nothing]

2. *Dafny wrongly says that things are proven*
   - Likelihood: "low" [Dafny has been proven by use]
   - Consequences: "catastrophic" [there is no proof]

The Searching Problem
Unsorted Case

Searching – as Computational Problem in Natural Language

Given

- an array A of numbers
- a number k

Compute

- an index i within A such that $A[i] = k$
- if no such index exists, return -1

Searching – Getting the Method Declaration Right

Given

- an array A of numbers
- a number k

Compute

- an index i within A such that $A[i] = k$
- if no such index exists, return -1

```
method Search(a: array<int>, k: int) returns (result: int)
{
  ...
}
```

Searching – Contract: Guarantees – 1st Case

Given

- an array A of numbers
- a number k

Compute

- an index i within A such that $A[i] = k$
- if no such index exists, return -1

```
method Search(a: array<int>, k: int) returns (result: int)
   requires true
   ensures   exists i: int :: (0 <= i < a.Length && a[i] == k)
                          ==> index == i
{
  ...
}
```

Searching – Contract: Guarantees – Adding the 2nd Case

Given

- an array A of numbers
- a number k

Compute

- an index i within A such that $A[i] = k$
- if no such index exists, return -1

```
method Search(a: array<int>, k: int) returns (result: int)
   requires true
   ensures   exists i: int :: (0 <= i < a.Length && a[i] == k)
                           ==> index == i
   ensures !(exists i: int :: (0 <= i < a.Length && a[i] == k))
                           ==> index == -1
{
  ...
}
```

Searching – Now the Program

```
method Search(a: array<int>, k: int) returns (result: int)
   requires true
   ensures   exists i: int :: (0 <= i < a.Length && a[i] == k)
                          ==> index == i
   ensures !(exists i: int :: (0 <= i < a.Length && a[i] == k))
                          ==> index == -1
{
  var l: int := 0;
  while (l < a.Length && a[l] != k)
  {
    l := l + 1;
  }
  if l < a.Length { index := l; }
  else  { index := -1; }
}
```

Searching – Annotate I

We need a relation for the loop variable: lower & upper boundary:

```
{
  var l: int := 0;
  while (l < a.Length && a[l] != k)
  {
    invariant 0 <= l <= a.Length
    l := l + 1;
  }
  if l < a.Length { index := l; }
  else  { index := -1; }
}
```

Searching – Annotate II

We need to state an intermediate result: the key can't be "left" of l

```
{
  var l: int := 0;
  while (l < a.Length && a[l] != k)
  {
    invariant 0 <= l <= a.Length
    invariant forall i: int :: (0 <= i < l ==> a[i] != k)
    l := l + 1;
  }
  if l < a.Length { index := l; }
  else  { index := -1; }
}
```

The Searching Problem
Sorted Case

Discussion

Why is it useful for searching to start with a sorted array?

Searching in Sorted Array – as Computational Problem in Natural Language

Given

- a **sorted** array A of numbers
- a number k

Compute

- an index i within A such that $A[i] = k$
- if no such index exists, return -1

Searching – Contract: Expectations

Given

- a sorted array A of numbers
- a number k

Compute

- an index i within A such that $A[i] = k$
- if no such index exists, return -1

```
method Search(a: array<int>, k: int) returns (result: int)
   requires forall i, j :: 0 <= i < j < a.Length ==> a[i] < a[j]
{
   ...
}
```

Searching – Contract: The Expectations Stay

Given

- a sorted array A of numbers
- a number k

Compute

- an index i within A such that $A[i] = k$
- if no such index exists, return -1

```
method Search(a: array<int>, k: int) returns (result: int)
   requires  forall i, j :: 0 <= i < j < a.Length ==> a[i] < a[j
   ensures   exists i: int :: (0 <= i < a.Length && a[i] == k)
                           ==> index == i
   ensures !(exists i: int :: (0 <= i < a.Length && a[i] == k))
                           ==> index == -1
{
  ...
}
```

Searching – Now the Program

```
method binSearch(a:array<int>, k:int) returns (index: int)
{
  var lo: int := 0 ;
  var hi: int := a.Length ;
  while (lo < hi)
  { var mid: int := (lo + hi) / 2 ;
    if (a[mid] < k) {
      lo := mid + 1 ;
    } else if (a[mid] > k) {
      hi := mid ;
    } else {
      return mid ;
    }
  }
  return -1 ;
}
```

Searching – Annotate I

We need a relation for the loop variables:

```
var lo: int := 0 ;
  var hi: int := a.Length ;
  while (lo < hi)
    invariant 0 <= lo <= hi <= a.Length
  { var mid: int := (lo + hi) / 2 ;
    if (a[mid] < k) {
      lo := mid + 1 ;
    } else if (a[mid] > k) {
      hi := mid ;
    } else {
      assert a[mid] == k;
      return mid ;
    }
  }
  return -1 ;
```

Searching – Annotate II

We need an intermediate result where the key k will not be:

```
var lo: int := 0 ;
  var hi: int := a.Length ;
  while (lo < hi)
    invariant 0 <= lo <= hi <= a.Length
    invariant forall i : int ::
                    (0 <= i < lo || hi <= i < a.Length)
                        ==> a[i] != k
  { var mid: int := (lo + hi) / 2 ;
    if (a[mid] < k) {
      lo := mid + 1 ;
    } else if (a[mid] > k) {
      hi := mid ;
    } else {
      assert a[mid] == k;
      return mid ;
    }
  }
  return -1 ;
```

Searching – Annotate III

We need an intermediate result where the key k could be:

```
var lo: int := 0 ;
  var hi: int := a.Length ;
  while (lo < hi)
    invariant 0 <= lo <= hi <= a.Length
    invariant forall i : int ::
                    (0 <= i < lo || hi <= i < a.Length)
                        ==> a[i] != k
    invariant exists i: int ::
                   (0 <= i < a.Length && a[i] == k)
                        ==> lo <= i <= hi
    ...
```

Reflections

Invariants

It's astonishing that the two design principles appear to be enough.

Dafny

It's astonishing that

- for linear search we did not need an invariant where the key might be
- for binary search we needed an invariant where the key might be

Dafny is "unpredictable" in what it can prove / what it can't prove

Program Verification

Even with a system like Dafny is (a lot of) **work**.

What you have learned

What you should be able to explain

- Where modelling happens in program verification
- How to develop invariants following the two design principles