Verifying imperative programs with Dafny

Bogdan Kyuchukov

Ludwig Maximilian University of Munich
Course: Deductive Software Verification

Lecturer: Prof. Gidon Ernst

February 28, 2025

Abstract

The goal of this work is to show how to formally reason about imperative programming constructs such as assignments, loops and heap allocated objects like arrays. To be able to achieve that, basic Dafny constructs will be shown, such as functions, methods, pre- and postconditions. The discussion thereafter will move towards recursion and termination as well as inductive datatypes. Having learned from those chapters, loop invariants and their usage will be explored. The final chapter explain the concept of frames and provide a detailed analysis of arrays.

Contents

1 Introduction

1.1 Why Dafny?

When software engineers encounter the field of formal methods and verification, this usually happens in an academic setting, where proof techniques are learned and done by hand. Moreover, actually taking advantage of those methods in practice involves a steep learning curve and lots of time. This unfortunately leads to less acceptance of verification techniques in an industry where fast time to market is essential Reid et al. (2020). Dafny promises to solve those problems. As a programming language designed to support specifications and proofs, it comes with an automated verifier that integrates seamlessly into most modern IDEs [1] making rigorous verification part of the software development process, thus reducing costly late-stage bugs that may be missed by testing. The language was designed by Rustan Leino in 2009 and it's current version at the time of this writing is 4.9.0 from 31. October 2024. Dafny is heavily used at Amazon Web Services to develop critical components of their access management, storage, and cryptography infrastructures Chakarov et al. (2022).

1.2 Dafny's build system

The main idea in such verification-aware programming languages is that code is divided into two parts - the specification part and the implementation part Leino (2023). The built-in verifier in Dafny acts as an extended type checker and constantly proves that the provided implementation actually meets the behavior stated in the specification part of the given function, method or class. This is done by transforming the code into an intermediary that a tool called Boogie can understand. The correctness of the Boogie program implies the correctness of the Dafny program. Boogie then generates first-order verification rules that are passed to the Z3 SMT solver. Any violations of these conditions are passed back as verification errors Herbert et al. (2012). Apart from C#, Dafny can be compiled to many other programming languages. This process is visualized in Figure 1.

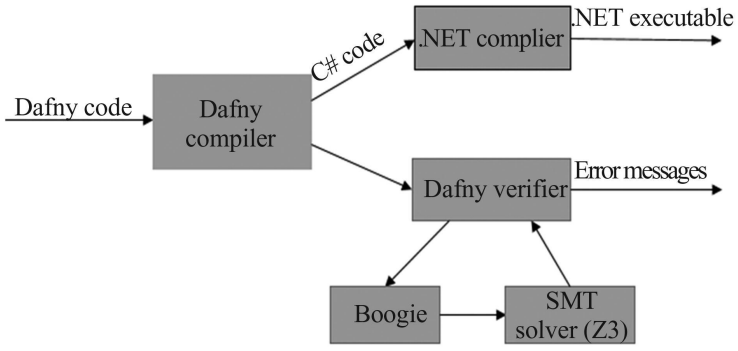

Figure 1: The Dafny build system as shown in Herbert et al. (2012)

[1] Integrated Development Environments (IDEs)

2 Building Blocks of Dafny

This work assumes the reader is familiarized with the fundamentals of program-semantics such as Floyd logic and Hoare triples, as they build the reasoning framework of Dafny and help to understand it more effectively.

2.1 Methods

As in other programming languages, a *method* is a block of code that prescribes some behavior. For instance, Listing 1 shows the declaration of a method called `Triple`.

```
1 method Triple(x: int) returns (r: int)
2  ensures Average(x) == 3 * x {
3   var y := 3 * x;
4   r := x + y;
5 }
```

Listing 1: Simple method in Dafny.

This method takes an *in-parameter* x of type integer and returns an *out-parameter* r, also of type integer. The **ensures** keyword is used to specify the method's *postcondition*. A postcondition expresses a property that must hold after every invocation of the method through all possible return points. Postconditions are part of the method's specification and appear before the body block Herbert et al. (2012). The body of a method is a list of statements that give the method's implementation. In Dafny, methods can have any number of in- and out-parameters. Inside the method, the out-parameters are like local variables and can be assigned and read. When the method ends, whatever values were assigned to the out-parameters will be returned to the caller. The in-parameters however cannot be re-assigned Leino (2023).

2.2 Functions

Functions in Dafny are very mathematical in nature, because they are deterministic. Any two invocations of a function with the same arguments will result in the same value. Listing 2 shows how a simple function named `Average` looks like.

```
1 function Average(a: int, b: int): int
2  requires 0 <= a && 0 <= b {
3   (a + b) / 2
4 }
```

Listing 2: Simple function in Dafny.

Whereas a method is declared to have some number of out-parameters, a function instead declares a result type, and while a method body is a statement, the body of a function is an expression Leino (2023). Another important difference between functions and methods is, that functions are *transparent*. This means that their implementation is directly visible to the caller. That's the reason why functions can also be used to describe pre- and postconditions like the precondition on line 2 in Listing 2. Methods on the other hand are *opaque*, so callers fully rely on the postcondition to reason about their return values.

2.3 Ghost constructs

Sometimes when reasoning about a program more information is needed than what the compiler has at runtime. A declaration, variable, statement, etc., that is used only for specification purposes is called a *ghost*. The verifier takes all ghosts into account, whereas the compiler erases all ghosts when it produces executable code. Other ghost constructs include pre- and postconditions (declared by **requires** and **ensures** clauses). They are used solely to specify the behavior of the program and establish a contract between callers and implementations Leino (2023).

```
1 method InvalidAssignment() returns (y: int) {
2   ghost var x := 10;
3   y := 2 * x; // cannot assign to compiled variable using a ghost
4 }
```

Listing 3: Invalid assignment to ghost.

To make sure ghost constructs can be successfully deleted after compilation, Dafny makes sure the compiled code doesn't rely on any ghost constructs. Listing 3 demonstrates a violation of this rule, which results in a verification error.

3 Recursion and termination

A recursive function is one that calls itself directly or indirectly. To be able to terminate, all recursive functions need a base case - a condition where the function stops calling itself and returns a result. An example is shown with a function called `SeqSum` in Listing 4, which computes the sum of all integers in a sequence. On line 4, if the base case isn't met, the function invokes itself with an incremented value for lo.

```
1 function SeqSum(s: seq<int>, lo: int, hi: int): int
2   requires 0 <= lo <= hi <= |s|
3   decreases hi - lo {
4   if lo == hi then 0 else s[lo] + SeqSum(s, lo + 1, hi)
5 }
```

Listing 4: Function computing the sum of all integers in a sequence recursively.

3.1 The decreases clause

Dafny can prove termination by using the **decreases** clause. If we can label each function invocation with a natural number and make sure that successive invocations strictly decrease that label, then it follows that at run time the recursive calls can only execute a finite number of times, and that is all the information needed to prove that the recursion eventually terminates Herbert et al. (2012). As shown on line 3 in Listing 4, the label of each invocation of `SeqSum` decreases gradually, because with each invocation the difference between hi and lo is smaller. When it reaches zero, the recursion terminates.

3.2 Well-founded relations

Termination metrics are not restricted to natural numbers. Termination can also be proven when two labels, which represent successive recursive invocations, are

in a relation that is *well-founded* Leino (2023). This relation, denoted as $a \succ b$, signifies that a reduces to b. For it to be *well-founded*, the following three conditions must be true:

- $\succ$ is *reflexive*: The relation never relates an element to itself. $a \succ a$ never holds.
- $\succ$ is *transitive*: Whenever $a \succ b$ and $b \succ c$ hold, then so does $a \succ c$.
- $\succ$ satisfies the *descending chain condition*: the relation has no *infinite descending chain*.

From these conditions it follows that a well-founded relation is a strict partial order that additionally satisfies the descending chain condition. Dafny predefines $\succ$ for each data type, and in some cases also for values between different types (but in most cases, values from different types are not related in this partial order) Leino (2023).

4 Inductive datatypes

In order to accomplish meaningful tasks, programs need to define their own data structures. A useful way to do that is to use *datatypes*. They define what *variants* data can have and what *payload* each variant carries. In contrast to classes, which describe mutable *state*, datatypes describe immutable *values* Leino (2023). Datatypes are often defined recursively. That is, values can be built up from simple variants or from variants that contain other datatypes. For this reason, they are known as *inductive datatypes* Leino (2023).

```
1 // enumeration
2 datatype Color = White | Yellow | Blue | Red
3
4 // inductive datatype, parameterized
5 datatype Tree<T> = Leaf(data: T) |
6                   Node(left: Tree<T>, right: Tree<T>)
```

Listing 5: A simple and an inductive datatype.

The type `Tree<T>` shown in Listing 5 can be parameterized with the type of its payload, that is, the data stored in the tree. One can also define functions on datatypes.

```
1 function Size<T>(t: Tree<T>): nat {
2   match t
3   case Leaf(_) => 1
4   case Node(left, right) => Size(left) + Size(right)
5 }
```

Listing 6: Function defined on a datatype.

Listing 6 shows a function `Size`, parameterized by a type `T` and a value t of type `Tree<T>`, that returns the number of leaf nodes in t. When calling a function, the type arguments can usually be inferred. If so, as seen on line 4 in Listing 6, we can omit them (along with the angle brackets) Leino (2023).

4.1 Discriminators and destructors

Very often when working with datatypes, there is a need to know the variant of the datatype value. That is, to know which constructor was used to create the value. One could write a function like the one shown in Listing 7 that determines this.

```
predicate IsNode(t: Tree<T>) {
  match t
  case Leaf(_) => false
  case Node(_, _) => true
}
```

Listing 7: Predicate checking if the argument t is a Node.

Because this is such a common operation, Dafny pre-defines such a *discriminator* for each constructor. The name of the discriminator is the name of the constructor followed by a question mark as shown on line 2 in Listing 8.

```
function GetLeft(t: Tree<T>): Tree<T>
 requires t.Node? {
  match t
  case Node(left, _) => left
}
```

Listing 8: Function returning the left subtree of a tree.

The discriminators are *members* of the type, which means they are accessed by following an expression with a dot and the name of the member Leino (2023). The precondition of `GetLeft` says the function can only be applied to a tree of the `Node` variant. This means that the `match` expression only needs to consider one case. Because `GetLeft` is also a common operation, Dafny provides a convenient way to declare such a *destructor* for each parameter of the constructor. This is done by simply naming the parameter Leino (2023). Line 6 in Listing 5 demonstrates this with the parameters `left` and `right`. Because destructors are also members of the type, they can be accessed the same way as discriminators.

5 Loop invariants

Many interesting computations are described as repetitions of steps. Apart from recursion, the other way of repeating code is the *loop*. The main way to reason about any loop is the *loop invariant*. It's part of the loop specification and restricts the state space that the loop is allowed to explore Leino (2023). Another key component is the *loop guard*, which controls whether the loop continues to perform steps or it stops doing so.

```
i := 0;
while i < 100
 // decrease 100 - i is added implicitly
 invariant 0 <= i <= 100 {
 i := i + 1;
}
assert i == 100; // true
```

Listing 9: Simple while loop.

In the example shown in Listing 9 the loop guard is the condition of the while loop on line 2. The loop invariant is defined on line 3 and specifies that i is only allowed to be in the range between 0 and 100. It's important that, on entry to the loop, both the loop invariant and the loop guard hold. On exit from the loop, only the loop invariant holds. The loop guard is negated. When working with loops, it's a good practice to make the loop invariant as restrictive as possible in order to have a clear specification of what needs to happen inside the loop.

5.1 Binary search

To effectively illustrate the advantages of Dafny, we will analyze the implementation of the binary search algorithm and provide a detailed discussion of each step. In this context, loop invariants play a crucial role.

5.1.1 Precondition

Binary search requires the array it searches to be sorted. Listing 10 specifies this precondition with the help of the universal quantifier.

```
1 forall i, j :: 0 <= i < j < a.Length ==> a[i] <= a[j]
```

Listing 10: All elements of array `a` are sorted.

It says that if we take two indices into the array, i and j where i is placed to the left of j, then the elements at those two indices are ordered properly. The operator `==>` means implication, i.e. if the condition on its left side holds, then the condition to its right side must also hold.

5.1.2 Postconditions

To begin with, the caller has to be assured that the returned index is inside the array bounds. Line 3 in Listing 11 specifies this. If the element we search for exists in the array, we want to return the index of its leftmost occurrence. Otherwise we want to return its earliest insertion point. The specification has to ensure that all elements in the array that appear before the element at the returned index must be smaller than it, as shown on line 4 in Listing 11. Similarly, all elements that appear after it must be greater than or equal to it, as indicated on line 5 of Listing 11.

```
1 method BinarySearch(a: array<int>, key: int) returns (n: int)
2   requires forall i, j :: 0 <= i < j < a.Length ==> a[i] <= a[j]
3   ensures 0 <= n <= a.Length
4   ensures forall i :: 0 <= i < n ==> a[i] < key
5   ensures forall i :: n <= i < a.Length ==> key <= a[i]
```

Listing 11: Binary search pre- and postconditions.

5.1.3 Missing loop invariants

The main idea of binary search is that we have two indices, lo and hi, that form a window inside the array. Anything to the left of the window is too small to be our element and anything to the right is at least as large. This window keeps

shrinking until it reaches 0, at which point both lo and hi meet. A logarithmic number of iterations is obtained by halving the window with each time. Part of the implementation is shown in Listing 12.

```
1  // Error: A postcondition couldn't be proved on this return path.
2  var lo, hi := 0, a.Length;
3  while lo < hi {
4    var mid := (lo + hi) / 2;
5    if a[mid] < key { // Error: index out of range.
6      lo := mid + 1;
7    } else {
8      hi := mid;
9    }
10 }
```

Listing 12: Binary search implementation with verifier errors.

An essential part of the implementation is missing in Listing 12, namely the loop invariants. Since the loop guard on line 3 holds, we enter the loop, but the automated verifier reports an error on line 5, because there is no loop invariant to bind the integer state space of lo and hi to be between 0 and `a.Length` respectively. Furthermore, the verifier reports another error directly at the beginning of the method, because the postconditions formulated on lines 4 and 5 in Listing 11 are not guaranteed to hold neither during, nor after the loop.

5.1.4 Adding loop invariants

There exists a very neat, refined design technique for loop specifications called "Replace Constant By Loop Variable" described in detail in Leino (2023). Basically, it says you can take your desired postconditions, copy them to use as loop invariants, and then simply swap out their constant, in this case n, for the loop index, in this case both lo and hi. Listing 13 demonstrates these changes.

```
1  {
2    var lo, hi := 0, a.Length;
3    while lo < hi
4    invariant 0 <= lo <= hi <= a.Length
5    invariant forall i :: 0 <= i < lo ==> a[i] < key
6    invariant forall i :: hi <= i < a.Length ==> key <= a[i] {
7      var mid := (lo + hi) / 2;
8      if a[mid] < key {
9        lo := mid + 1;
10     } else {
11       hi := mid;
12     }
13   }
14   n := lo;
15 }
```

Listing 13: Binary search implementation withiout verifier errors.

Finally, the added loop invariants ensure correctness at each loop iteration and establish the postconditions afterwards. Moreover, because of the implicit `decreases` clause, the verifier automatically proves the loop is not going to run infinitely. The whole `BinarySearch` method is a combination of both Listing 11 and Listing 13.

6 Arrays and simple frames

An *array* is a simple, mutable data structure stored in the heap and allocated using the `new` keyword. Because heap-allocated storage is accessed via references, the storage itself is not passed as parameter to functions and methods Leino (2023). So that specifications are able to identify relevant pieces of the heap, a *frame* is used.

6.1 Write frame

Apart from pre- and postconditions, the specification of a method also describes what the method is allowed to modify and what must be left unchanged. This is expressed by the method's *write frame*, which is defined using the `modifies` clause Leino (2023). If a method wants to change the contents of an array passed to it as a parameter, then it must include a `modifies` clause in its specification Leino (2023) as shown on line 3 in Listing 14.

```
1 method UpdateElements(a: array<int>)
2   requires a.Length == 10
3   modifies a
4   ensures old(a[4]) < a[4]
5   ensures a[6] <= old(a[6])
6   ensures a[8] == old(a[8]) {
7     a[4], a[8] := a[4] + 3, a[8] + 1;
8     a[7], a[8] := 516, a[8] - 1;
9 }
```

Listing 14: Method updating the contents of an array parameter.

This way the caller of `UpdateElements` will know that by calling the method and passing an array reference as an argument, the actual array stored in the heap will have changed according to what the method promises in its postcondition.

6.2 Old values

Often, when writing a postcondition for a method, we want to compare the state of an array or object before (pre-state) and after (post-state) an update. Usually whatever is written in an `ensures` clause refers to the post-state. To capture a pre-state value, the `old()` construct is used as demonstrated on lines 4, 5 and 6 in Listing 14. It should be emphasized that `old()` only affects heap dereferences in its argument. If only a value is passed as an argument or the actual dereferencing happens outside of the `old()`, this would not have the expected behavior.

6.3 Fresh arrays

A method that allocates a new array internally does not require a `modifies` clause, given that its caller has no way of accessing that array anyway. However, if that newly created array is returned to the caller, there is no information that it has been freshly allocated in the method. Thus, the caller is not allowed to modify it. To resolve this, the `fresh()` predicate is used in the postcondition to promise that the array is allocated inside the method and hence on behalf of the caller. At this point, the array can be modified as if it was created locally. Listing 15 demonstrates this.

```
method NewArray() returns (a: array<int>)
  ensures fresh(a) && a.Length == 20 {
    a := new int[20];
    var b := new int[30];
    a[6] := 216;
    b[7] := 343;
}

method Caller() {
  var a := NewArray();
  a[8] := 512; // not allowed without knowing 'a' is fresh
}
```

Listing 15: Method that guarantees a freshly allocated array.

6.4 Read frame

Given that a function cannot modify anything, it does not need a write frame. However, a function may need a *read frame*, which announces the function's dependencies on the heap and is specified using a `reads` clause, as shown on line 3 in Listing 16. This dependency information is important, because it's used to determine whether various mutations of the heap affect the value of the function Leino (2023).

```
predicate IsZeroArray(a: array<int>, lo: int, hi: int)
  requires 0 <= lo <= hi <= a.Length
  reads a
  decreases hi - lo {
    lo == hi || (a[lo] == 0 && IsZeroArray(a, lo + 1, hi))
}
```

Listing 16: Function that reads an array.

In summary, using *read and write frames* specify which parts of the heap a function may depend on and which parts of the heap a method is allowed to modify. Both can denote not only a single reference to an array or an object, but also a set of references.

7 Conclusion

To summarize, this work demonstrated some of Dafny's primary capabilities that enable software engineers to use formal verification methods during development and thus to increase code quality and to greatly reduce bugs. The future of Dafny looks bright, with currently a lot of ongoing research expanding its usage. At present, AI-powered coding assistants and code generators are rapidly gaining popularity. Although their generated code is oftentimes faulty, it's being used nonetheless. Li et al. (2025) propose a way to considerably improve that. By first guiding the LLM to generate Dafny code, it can then be automatically validated for correctness against the agreed-upon specifications. Only then the correct Dafny program is compiled to the target language and returned to the user. Even though writing lemmas and calculational proofs wasn't covered in this work, they are an integral part of Dafny and can be quite challenging. Álvaro Silva et al. (2024) also leverage LLMs to assist in generating proofs when they are needed to support the Dafny verifier.

List of Figures

Listings

References

Chakarov, A., Fedchin, A., Rakamaric, Z., and Rungta, N. (2022). Better counterexamples for dafny.

Herbert, L., Leino, K. R. M., and Quaresma, J. (2012). *Using Dafny, an Automatic Program Verifier*, pages 156–181. Springer Berlin Heidelberg, Berlin, Heidelberg.

Leino, K. R. M. (2023). *Program Proofs*. MIT Press.

Li, Y. C., Zetzsche, S., and Somayyajula, S. (2025). Dafny as verification-aware intermediate language for code generation.

Reid, A., Church, L., Flur, S., de Haas, S., Johnson, M., and Laurie, B. (2020). Towards making formal methods normal: meeting developers where they are.

Álvaro Silva, Mendes, A., and Ferreira, J. F. (2024). Leveraging large language models to boost dafny's developers productivity.