Verifying imperative programs with Dafny

Bogdan Kyuchukov

Ludwig Maximilian University of Munich
Course: Deductive Software Verification

Lecturer: Prof. Gidon Ernst

February 25, 2025

Abstract

The goal of this work is to show how to formally reason about imperative programming constructs such as loops, arrays and dynamically allocated objects. 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. Because analyzing objects in the heap is more challenging, searching and modifying arrays will be covered. The final chapter will include a detailed discussion about Dafny's dynamic frames and their significance.

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). This process is visualized in Figure 1.

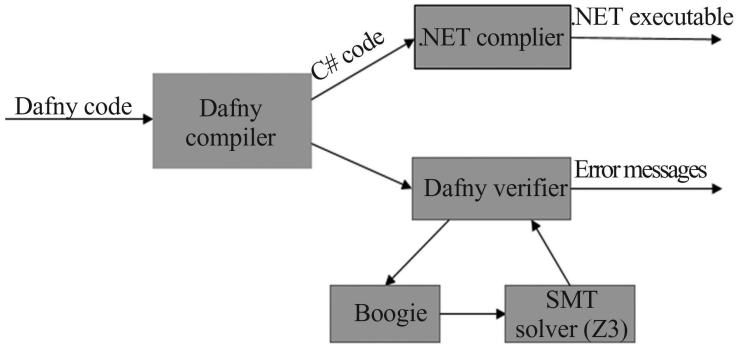

Figure 1: The Dafny build system as shown in Herbert et al. (2012)

[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: A 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: A 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: A 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: A 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.

```
1 predicate IsNode(t: Tree<T>) {
2   match t
3   case Leaf(_) => false
4   case Node(_, _) => true
5 }
```

Listing 7: A 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.

```
1 function GetLeft(t: Tree<T>): Tree<T>
2  requires t.Node? {
3   match t
4   case Node(left, _) => left
5 }
```

Listing 8: A predicate checking if the argument t is a Node.

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

6 Results

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua (**?**, p. 48). At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

7 Conclusion

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

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.

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.