Editor: **Gerard J. Holzmann**
Nimble Research
gholzmann@acm.org

Accessible Software Verification with Dafny

K. Rustan M. Leino

FOR MOST OF computer science's existence, creating fully reliable software has been impossible for all but the smallest programs. Principles and techniques for ensuring program correctness, and the very idea of reasoning rigorously about program correctness, are many decades old. Yet, only in the past decade or two have tools become powerful and usable enough to make formal software verification feasible.

Projects that have taken reliability to unprecedented levels include the seL4 microkernel,[1] the CompCert C compiler,[2] the FSCQ crash-recoverable file system,[3] the Eiffel-Base2 collections library,[4] and the IronFleet distributed-system components.[5] To pull this off, these projects used formal specifications and automated verification tools during software development. That's because trying to verify software that has already been written is not only more difficult but also misses the opportunity for the verification process to inform the development process and thus aid programmers (as opposed to being a postdevelopment chore).

Dafny is a modern formal-verification system that takes a language-based approach.[6] Its programming language includes the necessary specification and proof facilities. The idea is to provide developers with an immersive experience that feels like programming but encourages thinking about program correctness every step of the way.

Dafny: Language, Verifier, and IDEs

Dafny offers features from both imperative programming (for example, assignments, loops, and classes with dynamically allocated instances) and functional programming (for example, algebraic datatypes and functions). What sets the Dafny language apart is that it was designed with reasoning in mind. As such, it builds in specification constructs and supports proof authoring.

Because the language contains specification constructs, programmers can write down their intent in a machine-checkable way. An important aspect of Dafny (following Eiffel[7]) is that expressions have the same syntax and meaning in specifications as they do in executable code, so users have only one language to learn.

The Dafny verifier runs continuously in the IDEs (Visual Studio, Emacs, and Visual Studio Code). Whenever it can't verify a proof obligation, it flags it as an error, much like a word processor immediately marks dubious spelling or grammar. The verifier is automatic but requires hints from the user, akin to how automatic type checkers require users to supply some explicit type casts or types of variables. Typically, the user declares proof ingredients (for example, loop invariants or termination metrics, and the program itself), and the Dafny verifier fills in the proof glue. When a proof requires more information, the user can write and prove lemmas (mathematical theorems), for which the Dafny language provides constructs.

Dafny's approach of putting formal verification into a programming environment seems to have worked well with systems programmers, as evidenced by the Ironclad Apps[8] and IronFleet projects, which were done in Dafny. The approach also seems to go well in educational settings, as evidenced by several dozen universities worldwide that have used Dafny to teach students to reason about programs and learn about proofs. For these two kinds of users, the language's streamlined syntax, the verifier's high degree of automation, and the IDEs' fast turnaround provide an enticing experience.

To give a sense of what the Dafny language looks like, I'll give two examples: a small imperative procedure and a lemma. My goal is not to explain all the syntax but to point

 0740-7459/17/$33.00

out some of Dafny's salient features. The program text shown also suggests the power of the Dafny verifier because the program text is the only input to the verifier. Harder to convey in a written article is the experience of getting the feedback from the verifier. To get this experience, you can download the tool from its open source repository (github.com/Microsoft/dafny) or try it in a web browser (rise4fun.com).

Verifying an Imperative Procedure

The imperative procedure (called a *method*) in Figure 1 computes the maximum segment sum of an array. The method declaration includes a specification with two postconditions (indicated by the keyword `ensures`). These declare what's expected to hold upon the method's termination—something the method body must establish and that callers of the method can assume. The first postcondition of `MaxSegSum` says that the out-parameters `k` and `m` will denote a range of indices of `a`. The second postcondition, which uses a universal quantifier (∀ in math notation), says that among all index ranges `[p..q)` of `a`, none has a sum larger than `[k..m)`.

The loop in the method body declares a *loop invariant*, a list of conditions that hold at the very top of every loop iteration. These conditions facilitate reasoning about the loop and are like specifications of each loop iteration. Being clear in your mind about what the loop invariant is helps greatly in coding the loop correctly. For example, the loop invariant in `MaxSegSum` makes manifest that `[k..m)` is a range of indices among the first `n` elements of the array, that `s` is the segment sum for that index range, and that `s` is the maximum segment sum among the first `n` elements.

```
// find the index range [k..m) that gives the largest sum of any index range in a
method MaxSegSum(a: array<int>) returns (k: int, m: int)
  ensures 0 <= k <= m <= a.Length
  ensures forall p,q :: 0 <= p <= q <= a.Length ==> Sum(a, p, q) <= Sum(a, k, m)
{
  k, m := 0, 0;
  var s, n, c, t := 0, 0, 0, 0;
  while n < a.Length
    invariant 0 <= k <= m <= n <= a.Length && s == Sum(a, k, m)
    invariant forall p,q :: 0 <= p <= q <= n ==> Sum(a, p, q) <= s
    invariant 0 <= c <= n && t == Sum(a, c, n)
    invariant forall b :: 0 <= b <= n ==> Sum(a, b, n) <= t
  {
    t, n := t + a[n], n + 1;
    if t < 0 {
      c, t := n, 0;
    } else if s < t {
      k, m, s := c, n, t;
    }
  }
}

// sum of the elements in the index range [m..n)
function Sum(a: array<int>, m: int, n: int): int
  requires 0 <= m <= n <= a.Length
  reads a
{
  if m == n then 0 else Sum(a, m, n – 1) + a[n – 1]
}
```

FIGURE 1. A Dafny program for computing the maximum segment sum of an array. With the specifications and assertions given in this program, the Dafny verifier can ascertain the program's correctness (in a fraction of a second).

The documentation of these design decisions reduces the task of reasoning about the loop to reasoning about one arbitrary iteration. The verifier checks that the loop terminates (which follows from the fact that every iteration reduces the difference between `a.Length` and `n`) and reasons that `n` will equal `a.Length` upon termination. From this, it's clear that the method establishes its postconditions.

The method specification and loop invariant use the *function* `Sum`, which is also defined in Figure 1. Whereas a method in Dafny denotes imperative statements with possible side effects, a function denotes an expression. Functions behave like mathematical functions in that they always return the same value given the same inputs and they never mutate the program state.

As exemplified in Figure 1, a function can have a precondition (indicated by the keyword `requires`), which declares what's expected to hold upon entry to the function—something that callers of the function must establish and that the function body can assume. If a function reads the program state, it needs to say so in a `reads` clause. The function `Sum` says it may read the array elements of `a`. You can think of `reads` as adding a part of the heap as a parameter to

```
datatype Op = Plus | Times
datatype Expr = Const(int) | Var(string) | Node(Op, Expr, Expr)

function Subst(e: Expr, m: map<string,int>): Expr
{
  match e
  case Const(_) => e
  case Var(s) => if s in m.Keys then Const(m[s]) else e
  case Node(op, a, b) => Node(op, Subst(a, m), Subst(b, m))
}

lemma Idempotent(e: Expr, m: map<string,int>)
  ensures Subst(Subst(e, m), m) == Subst(e, m)
{
  match e
  case Const(_) =>
  case Var(s) =>
  case Node(op, a, b) =>
    calc {
      Subst(Subst(Node(op, a, b), m), m);
    == // def. Subst on Node
      Subst(Node(op, Subst(a, m), Subst(b, m)), m);
    == // def. Subst on Node
      Node(op, Subst(Subst(a, m), m), Subst(Subst(b, m), m));
    == { Idempotent(a, m); Idempotent(b, m); }
      Node(op, Subst(a, m), Subst(b, m));
    }
}
```

FIGURE 2. A lemma in Dafny is really just a ghost method—that is, a method that gets erased by the compiler. Proving a lemma thus follows the same rules as proving a method.

the function. (Figure 1 doesn't show it, but a method can also have a precondition and a **modifies** clause, which declares the parts of the program state the method may mutate.)

To understand what **MaxSegSum** does, it suffices to inspect its specification and the definition of **Sum**. This is simpler than trying to understand the **MaxSegSum** implementation. The Dafny verifier automatically checks that the implementation satisfies the given specification.

Integers

The integers used in my example are mathematical (unbounded), so they can't overflow. In some cases, it's better to use restricted subsets of the integers. To allow this, Dafny supports user-defined integer types—for instance, a program can declare a type **int32** denoting the 32-bit two's-complement integers. The verifier checks that the program's operations on such a type never overflow. The type system separates user-defined integer types from ordinary integers, and the compiler analyzes the type constraints when choosing a runtime representation for the executable code.

Ghost vs. Compiled

In a verified program, some parts of the program text are there just to help developers reason about the program's behavior. Conversely, you could perhaps argue that some parts of the program text are there just to help the compiler generate executable code from the program's specifications.

However you view it, the Dafny language distinguishes between these parts. Some declarations (including variables and functions) can be marked as *ghost*, which means they are ignored by the compiler and have no runtime representation. Specification clauses (for example, preconditions and loop invariants) are always ghost, so they're never evaluated at runtime. For instance, the declaration that introduces **Sum** in Figure 1 designates the function as ghost. So, the function doesn't get compiled into code but can be used in specifications.

Stating and Proving a Lemma

Commonly, a program's correctness depends on subtle properties of the data structures or functions involved. To convince the verifier of one of these properties, you might need to state and prove a lemma. A lemma in Dafny is a ghost method; it takes parameters, has preconditions and postconditions, and has a body consisting of statements. To obtain the property stated by the lemma, a program "calls" the lemma where the property is needed, just as it would call a method.

To illustrate a lemma in Dafny, Figure 2 defines the abstract syntax tree **Expr**, representing simple expressions. It also defines a substitution function that replaces certain variables with constant values. This lemma proves that substitution is idempotent—that applying the same substitution twice is no different from applying it once.

As defined by the function's body, the value returned by **Subst** depends on

the form of parameter e. For example, the third case in Figure 2 says that if e has the form Node(op, a, b), the value returned is a Node whose subexpressions come from recursive calls to Subst.

The lemma in Figure 2 uses a postcondition (indicated by ensures, just like the method in Figure 1) to state the condition to be proven. Obtaining a proof of the lemma means convincing the verifier that every control path through the lemma's body leads to the postcondition. The body of Idempotent is broken up into three cases that depend on the form of e. The first two are handled automatically by the Dafny verifier and require no further assistance.

The third case illustrates a typical way to write a proof in Dafny. The *verified calculation* (indicated by the keyword calc) in Figure 2 starts with the left-hand side of the proof goal (the equality in the postcondition) and shows, through a series of equality-preserving steps, that the proof goal's left-hand side equals its right-hand side.[9] The calculation's first two steps simply rewrite Subst(Node(...)) using the definition of Subst. The third step holds on account of what's usually called the *induction hypothesis* of the lemma to be proved. In Dafny, the induction hypothesis corresponds to the information gained from a recursive call of the lemma. This example requires two recursive calls: one for each subexpression a and b. The recursive calls are given in a hint to Dafny (in braces) to explain why the calculation step holds. In contrast, the first two calculation steps are simple enough that the verifier needs no formal hint.

The typical proof in Figure 2 is actually more elaborate than necessary. A shorter way to write it would be to replace the calc statement with just the two recursive calls to Idempotent. In fact, Dafny tries to invoke the induction hypothesis automatically.[10] So, for this example, even these two recursive calls can be omitted, and the Dafny verifier can prove the lemma even if the entire lemma body is replaced by the empty body, {}.

Formal software verification includes specifications, tools, and interactivity with the developer. By combining these key components into a programming language and a familiar programming environment with high automation, Dafny makes verification more easily accessible to programmers and students.

Program safely!

ABOUT THE AUTHOR

K. RUSTAN M. LEINO is a principal researcher in the Research in Software Engineering group at Microsoft Research and a visiting professor in Imperial College London's Department of Computing. Contact him at leino@acm.org.

References

1. G. Klein et al., "seL4: Formal Verification of an OS Kernel," *Proc. ACM SIGOPS 22nd Symp. Operating Systems Principles* (SOSP 09), 2009, pp. 207–220.
2. X. Leroy, "Formal Verification of a Realistic Compiler," *Comm. ACM*, vol. 52, no. 7, 2009, pp. 107–115.
3. H. Chen et al., "Using Crash Hoare Logic for Certifying the FSCQ File System," *Proc. 25th Symp. Operating Systems Principles* (SOSP 15), 2015, pp. 18–37.
4. N. Polikarpova, J. Tschannen, and C.A. Furia, "A Fully Verified Container Library," *Proc. 20th Int'l Symp. Formal Methods* (FM 15), LNCS 9109, Springer, 2015, pp. 414–434.
5. C. Hawblitzel et al., "IronFleet: Proving Practical Distributed Systems Correct," *Proc. 25th Symp. Operating Systems Principles* (SOSP 15), 2015, pp. 1–17.
6. K.R.M. Leino, "Dafny: An Automatic Program Verifier for Functional Correctness," *Proc. 16th Int'l Conf. Logic for Programming, Artificial Intelligence, and Reasoning* (LPAR 16), LNCS 6355, Springer, 2010, pp. 348–370.
7. B. Meyer, *Object-Oriented Software Construction*, Prentice-Hall Int'l, 1988.
8. C. Hawblitzel et al., "Ironclad Apps: End-to-End Security via Automated Full-System Verification," *Proc. 11th USENIX Symp. Operating Systems Design and Implementation* (OSDI 14), 2014, pp. 165–181.
9. K.R.M. Leino and N. Polikarpova, "Verified Calculations," *Proc. 5th Int'l Conf. Verified Software: Theories, Tools, and Experiments* (VSTTE 13), LNCS 8164, Springer, 2014, pp. 170–190.
10. K.R.M. Leino, "Automating Induction with an SMT Solver," *Proc. 13th Int'l Conf. Verification, Model Checking, and Abstract Interpretation* (VMCAI 12), LNCS 7148, Springer, 2012, pp. 315–331.