Developing Verified Programs with Dafny

K. Rustan M. Leino
Microsoft Research, Redmond, WA, USA
leino@microsoft.com

***Abstract*—Dafny is a programming language and program verifier. The language includes specification constructs and the verifier checks that the program lives up to its specifications. These tutorial notes give some Dafny programs used as examples in the tutorial.**

I. Introduction

Reasoning about programs is a fundamental skill that every software engineer needs. Dafny is a programming language designed with verification in mind [0]. It can be used to develop provably correct code (e.g., [1]). Because its state-of-the-art program verifier is easy to run—it runs automatically in the background in the integrated development environment[0] and it runs at the click of a button in the web version[1]—Dafny also stands out as a tool that through its analysis feedback helps teach reasoning about programs. This tutorial shows how to use Dafny, and these tutorial notes show some example programs that highlight major features of Dafny.

The Dafny language is type-safe and sequential. It includes constructs found in common imperative programs, like loops and dynamic object allocation, as well as in functional programs, like recursive functions and inductive (and co-inductive) datatypes. Importantly, it also includes specification constructs, like pre- and postconditions, which let a programmer record the intended behavior of the program along with the executable code that is supposed to cause that behavior. The verifier checks that the code will always behave as described. It also checks that programs terminate and are free of evaluation errors, like indexing an array outside its bounds.

The Dafny verifier is invoked just like the Dafny compiler. That is, the input to the program verifier is the program text itself. The verifier responds by giving error messages for proof obligation it is not able to prove.

Other Dafny tutorials have been published as lecture notes from the 2011 Marktoberdorf Summer School [2] and 2011 LASER Summer School [3].

II. Basics

To illustrate some basic features of Dafny, consider the program in Fig. 0. It shows a *method*, which is a code procedure. The in-parameters (`a,b,c`) are passed by value, as are the out-parameters (`x,y,z`), which can also be used as local variables in the body of method. The method's *postcondition* (keyword **ensures**) spells out some conditions that the method is to establish: the out-parameters are copies of the in-parameters, but in sorted order. The body of the method shows uses of the familiar **if** statement and of Dafny's simultaneous-assignment statement.

```
method Sort(a: int, b: int, c: int) returns (x: int, y: int, z: int)
  ensures x ≤ y ≤ z ∧ multiset{a, b, c} = multiset{x, y, z};
{
  x, y, z := a, b, c;
  if z < y { y, z := z, y; }
  if y < x { x, y := y, x; }
  if z < y { y, z := z, y; }
}
```

Fig. 0. Example that shows some basic features of Dafny. The method's specification says that `x,y,z` is a sorted version of `a,b,c`, and the verifier automatically checks the code to satisfy this specification.

```
method TriangleNumber(N: int) returns (t: int)
  requires 0 ≤ N;
  ensures t = N*(N+1) / 2;
{
  t := 0;
  var n := 0;
  while n < N
    invariant 0 ≤ n ≤ N ∧ t = n*(n+1) / 2;
  { n := n + 1; t := t + n; }
}
```

Fig. 1. Illustration of the use of a loop invariant to reason about the loop.

The verifier checks that all code paths lead to the postcondition being established. If this were not so (for example, if you change the code or specification in bad ways), the verifier would complain about the error. One possible way to debug such an error is to use the Boogie Verification Debugger [4].

III. Loop Invariants

Figure 1 shows a method that computes triangle number `N`. The *precondition* (keyword **requires**) says what callers must establish in order to invoke the method (enforced by the verifier at call sites) and this condition can thus be assumed to hold on entry to the method body. To reason about the loop, it is necessary to supply a *loop invariant*, a condition that holds at the very top of each loop iteration (that is, just prior to each evaluation of the loop guard). The loop invariant is checked to hold on entry to the loop and is checked to be maintained by the loop body.

A central point in understanding how to reason about loops is that it is not possible to consider a loop's behavior by looking at all its (infinitely many) possible unrollings. Instead, *all* the information about the variables that are changed by the loop comes from the loop invariant. For example, if we

[0]Install from http://dafny.codeplex.com.

[1]Try it at http://rise4fun.com/dafny.

```
function pow2(n: int): int
  requires 0 ≤ n;
{ if n = 0 then 1 else 2 * pow2(n-1) }
```

Fig. 2. Function pow2 gives a straightforward recursive definition of the mathematical expression 2^n.

```
datatype Tree⟨T⟩ = Leaf | Node(Tree, T, Tree);
function Contains⟨T⟩(t: Tree⟨T⟩, v: T): bool
{
  match t
  case Leaf ⇒ false
  case Node(left, x, right) ⇒
    x = v ∨ Contains(left, v) ∨ Contains(right, v)
}
method Fill⟨T⟩(t: Tree⟨T⟩, a: array⟨T⟩, start: int) returns (end: int)
  requires a ≠ null ∧ 0 ≤ start ≤ a.Length;
  modifies a;
  ensures start ≤ end ≤ a.Length;
  ensures forall i • 0 ≤ i < start ⟹ a[i] = old(a[i]);
  ensures forall i • start ≤ i < end ⟹ Contains(t, a[i]);
{
  match t { case Leaf ⇒ end := start;
            case Node(left, x, right) ⇒
              end := Fill(left, a, start);
              if end < a.Length {
                a[end] := x;
                end := Fill(right, a, end + 1);
} }           }
```

Fig. 3. A program that defines an (immutable) Tree datatype and also uses a (mutable) array. Part of the method's specification is in terms of the recursive function Contains.

leave off n ≤ N from the loop invariant in Fig. 1, the verifier will not know any upper bound for n after the loop and will therefore complain that the postcondition might not hold. In other words, the verifier does not look at the loop body when reasoning about which program states are reachable after an arbitrary number of iterations; instead, it relies on the programmer-supplied invariant to provide a sufficiently constrained description of those states. This issue is the most important difference between reasoning about programs and merely executing them; be sure to play around with the tool to help you fully absorb the issue.

IV. Functions

A *function* in Dafny is a mathematical function. It is defined not by code but by an expression. Functions can be used in a functional style of programming, and they are often used in specifications of (possibly imperative) programs.

Figure 2 defines a recursive function for exponentiating 2. As for a method, the precondition says when the function is allowed to be invoked.

About the syntax, note that the function's defining body is not terminated by a semi-colon. Also, whereas the **if** statement surrounds the then branch and the optional else branch in curly braces, the analogous **if** expression uses the keywords **then** and **else** and insists on both being present.

V. Datatypes

The imperative and functional features of Dafny can be used together, as illustrated in Fig. 3. Tree⟨T⟩ is an inductive datatype representing trees of T elements. The recursively defined function Contains(t, v) returns whether or not v is an element of tree t. Method Fill copies elements from a tree t into an array a. The caller indicates where in a the copying is to begin (index start). The method is allowed to change the elements of a, as indicated by the **modifies** clause. The method leaves the first start elements of the array unchanged (second postcondition) and fills the next end-start elements with elements from the tree (third postcondition), where end is determined by the method implementation (subject to the constraints in the first postcondition).

Note that the specification of Fill does not mention every interesting aspect of the method's behavior. For example, the specification says nothing about the order or multiplicity of the tree elements copied into the array. How strong or weak to make a specification is an engineering choice—a trade-off between assurance and the price to obtain that assurance.

VI. Classes

Classes offer a way to dynamically allocate mutable data structures. References (that is, pointers) to components of these data structures gives flexibility in programming, but generally also make specifications more complicated (though, arguably, also make verification more worthwhile). A common approach to specifying a class in Dafny is to use two sets of variables, some *ghost* variables that give a simple way to understand the behavior of the class and some *physical* (i.e., non-ghost) variables that form an efficient implementation of the class. The relation between the two sets of variables is described in a *class invariant* [5], which in Dafny is typically coded into a boolean function (a *predicate*) called Valid that gets used in method specifications. The program updates both sets of variables, maintaining the validity condition, but the compiler emits only the physical variables into the executable code. The Dafny language embraces the idea of ghost constructs, not just for variables but also for other declarations and statements.

Figure 4 gives an example class: a FIFO queue implemented by an array. To avoid the queue's length exceeding that of the array, the Enqueue method sometimes has to allocate a larger array.

To reduce the tedium of writing many of the idiomatic specifications, the class is marked with the *{:autocontracts}* attribute, which has the effect of filling in many of the standard specifications. For example, auto-contracts looks for the predicate Valid() and adds it as a postcondition to each constructor and as a pre- and postcondition to each method. It also adds some bookkeeping to handle the specification of **modifies** clauses. The result is a particular encoding of *dynamic frames* [6], which can also be done manually in Dafny (see [0], [7]).

Another noteworthy aspect of the code is how Enqueue shifts the queue elements to the beginning of the (possibly newly allocated) array. This operation is easily coded with Dafny's **forall** statement, which simultaneously performs a number of assignments. Not only is this more straightforward

```
class {:autocontracts} SimpleQueue⟨Data⟩
{
  ghost var Contents: seq⟨Data⟩;
  var a: array⟨Data⟩;
  var m: int, n: int;
  predicate Valid() {
    a ≠ null ∧ a.Length ≠ 0 ∧
    0 ≤ m ≤ n ≤ a.Length ∧ Contents = a[m..n]
  }
  constructor ()
    ensures Contents = [];
  {
    a, m, n, Contents := new Data[10], 0, 0, [];
  }
  method Enqueue(d: Data)
    ensures Contents = old(Contents) + [d];
  {
    if n = a.Length {
      var b := a;
      if m = 0 { b := new Data[2 * a.Length]; }
      forall (i | 0 ≤ i < n - m) {
        b[i] := a[m + i];
      }
      a, m, n := b, 0, n - m;
    }
    a[n], n, Contents := d, n + 1, Contents + [d];
  }
  method Dequeue() returns (d: Data)
    requires Contents ≠ [];
    ensures d = old(Contents)[0] ∧ Contents = old(Contents)[1..];
  {
    assert a[m] = a[m..n][0];
    d, m, Contents := a[m], m + 1, Contents[1..];
  }
}
method Main()
{ var q := new SimpleQueue();
  q.Enqueue(5); q.Enqueue(12);
  var x := q.Dequeue();
  assert x = 5;
}
```

Fig. 4. A class that implements a queue by an array. Ghost fields are used to specify the class, physical fields are used to implement it, and the invariant that glues the two together is declared as the predicate Valid().

than using a loop, but reasoning about a loop is also more cumbersome due to having give a loop invariant to characterize the state during the loop's intermediate iterations.

Finally, note that the example shows a Main method that serves as a test harness. However, rather than relying on dynamic execution to carry out the test, the assertion is checked by the verifier, statically. This seeks to validate that the specifications are strong enough to be useful to a client.

VII. Lemmas

As a more advanced topic, let us consider an example where the verifier in unable to prove, by itself, something that is true. As programs get more advanced, such situations arise more frequently.

The ComputePow2 method in Fig. 5 computes the pow2 function from Fig. 2 in logarithmic time. The correctness of the second branch hinges on the fact that $2^n = (2^{n/2})^2$ when n is even. This fact is stated as a lemma. More precisely, the ghost method Lemma, which can be called when n is even, promises to return in a state where the property holds. By calling the lemma, ComputePow2 thus obtains the information it needs. The proof of the lemma is the body of the ghost method, where every code path must convince the verifier that

```
method ComputePow2(n: nat) returns (p: nat)
  ensures p = pow2(n);
{
  if n = 0 {
    p := 1;
  } else if n % 2 = 0 {
    p := ComputePow2(n / 2);
    p := p * p;
    Lemma(n);
  } else {
    p := ComputePow2(n-1);
    p := 2 * p;
} }
ghost method Lemma(n: nat)
  requires n % 2 = 0;
  ensures pow2(n) = pow2(n/2) * pow2(n/2);
{
  if n ≠ 0 { Lemma(n-2); }
}
```

Fig. 5. The ComputePow2 method gives a logarithmic way to compute pow2(n). **nat** is the non-negative subrange of **int**. Proving the correctness of ComputePow2 requires using a lemma, that pow2(n) is the square of pow2(n/2), which is stated and proved by the ghost method Lemma.

the postcondition is established. Since Lemma is recursive, the proof corresponds to a proof by induction. Finally, note that no executable code is generated for Lemma, since it is declared ghost.

Another lemma is the **assert** statement in method Dequeue in Fig. 4. It points out a necessary ingredient of the proof (namely, a particular property about sequences) that the prover does not figure out on its own.

VIII. Conclusions

Getting comfortable using the features presented takes practice. Use the tool to obtain this practice. Dafny also includes many other features, like customized termination specifications, CLU-style iterators, staged program development via refinement, co-recursion, and proof calculations. Look for other tutorials, documentation, and associated papers to learn more about these.

References

[0] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *LPAR-16*, ser. LNCS, vol. 6355. Springer, 2010, pp. 348–370.

[1] H. Mai, E. Pek, H. Xue, S. T. King, and P. Madhusudan, "Verifying security invariants in ExpressOS," in *ASPLOS '13*. ACM, 2013, pp. 293–304.

[2] J. Koenig and K. R. M. Leino, "Getting started with Dafny: A guide," in *Software Safety and Security: Tools for Analysis and Verification*, ser. NATO Science for Peace and Security Series D: Information and Communication Security. IOS Press, 2012, vol. 33, pp. 152–181.

[3] L. Herbert, K. R. M. Leino, and J. Quaresma, "Using Dafny, an automatic program verifier," in *LASER, International Summer School 2011*, ser. LNCS, vol. 7682. Springer, 2012, pp. 156–181.

[4] C. Le Goues, K. R. M. Leino, and M. Moskal, "The Boogie Verification Debugger (tool paper)," in *SEFM 2011*, ser. LNCS, vol. 7041. Springer, 2011, pp. 407–414.

[5] B. Meyer, *Object-oriented Software Construction*, ser. Series in Computer Science. Prentice-Hall International, 1988.

[6] I. T. Kassios, "Dynamic frames: Support for framing, dependencies and sharing without restrictions," in *FM 2006*, ser. LNCS, vol. 4085. Springer, 2006, pp. 268–283.

[7] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson, "Behavioral interface specification languages," *ACM Computing Surveys*, vol. 44, no. 3, Jun. 2012, article 16.