Writing proofs in Dafny

K. Rustan M. Leino

Amazon Web Services

14 October 2024
Tutorial, FMCAD 2024
Prague, Czech Republic

Goals of tutorial

- Show how to write and debug proofs
- Expand your mind about how to describe mechanical proofs
- Demonstrate Dafny as a competent, automated proof assistant
- Inspire you to write your next verified programs and proofs in Dafny (and *teach* using Dafny)

Language

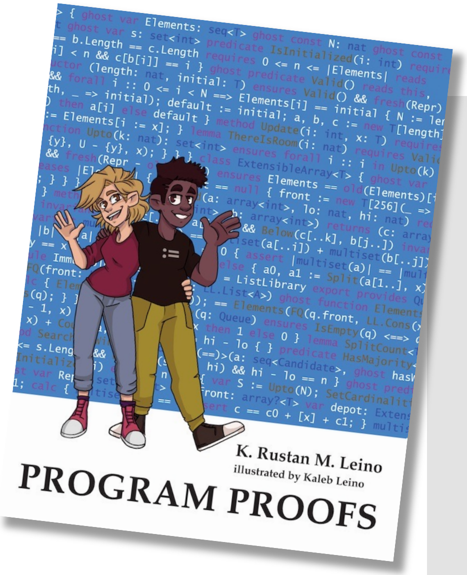
- Dafny (`dafny.org`)
- Programming language
  - Java-like
  - Verification-aware (proof-oriented)
- Constructs for
  - Imperative programming
  - Functional programming
  - Specifications
  - Proofs
- VS Code integration

Tools

- Static verifier
- Compilers from Dafny to
  - Java
  - C#
  - JavaScript
  - Go
  - Python
  - Rust
- Auditor
- Test generator

Some uses of Dafny

- Teaching
  - Over a decade, many universities
  - Book (MIT Press): `program-proofs.com`



- At AWS
  - AWS Encryption SDK
  - AWS Database Encryption SDK
  - AWS's authentication engine
    Talk at AWS re:Inforce 2024:
    https://youtu.be/oshxAJGrwMU?si=2HRf2XvNR-8RMIXZ
  - …
- …

A comparison with Lean*

(actually, Dafny is more similar to Why3, F*, SPARK Ada, or Viper)

- Program-centered – built-in support for pre- and postconditions and invariants
- Preconditions are used everywhere
- Built-in mutable objects (no monads, but `:-` for handling failures)
- `nat` is a subset of `int` (not an inductively defined type)
- A `predicate` is just a `function` that returns `bool`
- There is no `Prop` and no values of type "proof"
- Traits (abstract supertypes) are used more often than higher-order functions
- One, fixed well-founded order for termination checking
- Order of declarations is immaterial
- `least`/`greatest predicate`s ("inductive/condinductive predicates") are functions, not data
- Codatatypes, corecursion, and coinduction are supported
- Supports induction recursion
- No tactics
- IDE: no window for proof state
- …

*) since this Dafny tutorial at FMCAD was preceded by a tutorial on Lean and many audience members attended both

Methods vs functions

```
method M(x: int) returns (y: int) {
  <statements>
}
```

- Can mutate state
- Can be nondeterministic
- Opaque (reasoning uses specification, never the body)

```
function F(x: int): int {
  <expr>
}
```

- Cannot mutate state
- Deterministic
- Transparent (reasoning uses body) – default or opaque – by declaration

Statements vs expressions

statement	expression
if G { *<statements>* } else { *<statements>* optional }	if G then *<expr>* else *<expr>*
match E case Nil => *<statements>* case Cons(x, tail) => *<statements>*	match E case Nil => *<expr>* case Cons(x, tail) => *<expr>*

Preliminaries

```
method Double(x: int) returns (r: int)
  ensures r == 2 * x
{
  r := x + x;
}

method Double'(x: int) returns (r: int)
  ensures r == 2 * x
{
  if x < 0 {
    r := x + x;
  } else {
    var y := 3 * x;
    r := y - x;
  }
}
```

Preliminaries

```
method Double''(x: nat) returns (r: int)
  ensures r == 2 * x
{
  if x == 0 {
    return 0;
  } else {
    r := Double''(x - 1);
    assert r == 2 * (x - 1) == 2 * x - 2;
    r := r + 2;
  }
}

method Double'3(x: nat) returns (r: int)
  ensures r == 2 * x
{
  r := 0;
  var i := 0;
  while i < x
    invariant 0 <= i <= x
    invariant 0 <= r
    invariant r == 2 * i
  {
    r := r + 2;
    i := i + 1;
  }
  assert i == x;
}
```

Binary search

```
method BinarySearch(a: array<int>, key: int) returns (r: int)
  requires forall i, j :: 0 <= i < j < a.Length ==> a[i] <= a[j]
  ensures 0 <= r ==> r < a.Length && a[r] == key
  ensures r < 0 ==> key !in a[..]
{
  var lo, hi := 0, a.Length;
  while lo < hi
    invariant 0 <= lo <= hi <= a.Length
    invariant key !in a[..lo] && key !in a[hi..]
  {
    var mid := (lo + hi) / 2;
    if a[mid] == key {
      return mid;
    } else if key < a[mid] {
      hi := mid;
    } else {
      lo := mid + 1;
    }
  }
  return -1;
}
```

Binary search with bounded integers

```
newtype int32 = x | -0x8000_0000 <= x < 0x8000_0000

method BinarySearch(a: array<int>, key: int) returns (r: int32)
  requires forall i, j :: 0 <= i < j < a.Length ==> a[i] <= a[j]
  requires a.Length < 0x8000_0000
  ensures 0 <= r ==> r < a.Length as int32 && a[r] == key
  ensures r < 0 ==> key !in a[..]
{
  var lo, hi := 0, a.Length as int32;
  while lo < hi
    invariant 0 <= lo <= hi <= a.Length as int32
    invariant key !in a[..lo] && key !in a[hi..]
  {
    assert (lo as int + hi as int) / 2 ==
           (lo + (hi - lo) / 2) as int;
    var mid := lo + (hi - lo) / 2;
    if a[mid] == key {
      return mid;
    } else if key < a[mid] {
      hi := mid;
    } else {
      lo := mid + 1;
    }
  }
  return -1;
}
```

Minimum

```
method Min(s: seq<int>) returns (m: int)
  requires |s| != 0
  ensures m in s
  ensures forall j :: 0 <= j < |s| ==> m <= s[j]
{
  m := s[0];
  for i := 1 to |s|
    invariant m in s
    invariant forall j :: 0 <= j < i ==> m <= s[j]
  {
    if s[i] < m {
      m := s[i];
    }
  }
}
```

Logic

```
predicate P(x: int)
predicate Q(x: int)

lemma {:axiom} EstablishP(x: int)
  ensures P(x)

lemma {:axiom} EstablishQ(x: int)
  ensures Q(x)

lemma EstablishPAndQ(x: int)
  ensures P(x) && Q(x)
{
  EstablishP(x);
  EstablishQ(x);
}

lemma EstablishPOrQ(x: int)
  ensures P(x) || Q(x)
{
  if * {
    EstablishP(x);
  } else {
    EstablishQ(x);
  }
}
```

Logic.dfy $^1/_4$

Logic

```
predicate P(x: int)
predicate Q(x: int)

lemma {:axiom} GivenPEstablishQ(x: int)
  requires P(x)
  ensures Q(x)

lemma EstablishPImpliesQ(x: int)
  ensures P(x) ==> Q(x)
{
  if P(x) {
    GivenPEstablishQ(x);
  }
}

lemma Or(x: int, y: int)
  requires P(x) || P(y)
  ensures Q(x) || Q(y)
{
  if P(x) {
    GivenPEstablishQ(x);
  } else {
    GivenPEstablishQ(y);
  }
}
```

Logic

```
predicate P(x: int)
predicate Q(x: int)

lemma ModusPonens(x: int)
  requires L0: P(x)
  requires L1: P(x) ==> Q(x)
  ensures Q(x)
{
  calc {
    true;
  ==> { reveal L0; }
    P(x);
  ==> { reveal L1; }
    Q(x);
  }
}
```

```
lemma ModusTollens(x: int)
  requires L0: !Q(x)
  requires L1: P(x) ==> Q(x)
  ensures !P(x)
{
  if P(x) {
    assert Q(x) by {
      reveal L1;
    }
    assert false by {
      reveal L0;
    }
  } else {
    // easy!
  }
}
```

Logic

```
predicate P(x: int)
predicate Q(x: int)

lemma {:axiom} GivenPEstablishQ(x: int)
  requires P(x)
  ensures Q(x)

lemma AllQ()
  ensures forall x :: P(x) ==> Q(x)
{
  forall x | P(x)
    ensures Q(x)
  {
    GivenPEstablishQ(x);
  }
}
```

```
lemma Exists()
  requires exists x :: P(x)
  ensures exists z :: Q(z)
{
  var y :| P(y);
  GivenPEstablishQ(y);
}

lemma Exists'(x: int) returns (z: int)
  requires P(x)
  ensures Q(z)
{
  GivenPEstablishQ(x);
  z := x;
}
```

Opaque functions

```
opaque function F(x: int): int {
  x + 15
}

lemma Test(x: int) {
  reveal F();
  assert x < F(x);
}

lemma AboutF(x: int)
  ensures F(x) == x + 15
{
  reveal F();
}

lemma TestOne(x: int) {
  AboutF(x);
  assert x < F(x);
}
```

Coincidence count

```
function InBoth(a: seq<int>, b: seq<int>): set<int> {
  set x | x in a && x in b
}

method CoincidenceCount(a: seq<int>, b: seq<int>) returns (c: nat)
  requires forall i, j :: 0 <= i < j < |a| ==> a[i] < a[j]
  requires forall i, j :: 0 <= i < j < |b| ==> b[i] < b[j]
  ensures c == |InBoth(a, b)|
{
  c := 0;
  var i, j := 0, 0;
  while i < |a| && j < |b|
    invariant 0 <= i <= |a| && 0 <= j <= |b|
    invariant |InBoth(a, b)| == c + |InBoth(a[i..], b[j..])|
  {
    if a[i] == b[j] {
      assert InBoth(a[i..], b[j..]) ==
             {a[i]} + InBoth(a[i + 1..], b[j + 1..]);
      c, i, j := c + 1, i + 1, j + 1;
    } else if a[i] < b[j] {
      assert InBoth(a[i..], b[j..]) == InBoth(a[i + 1..], b[j..]);
      i := i + 1;
    } else {
      assert InBoth(a[i..], b[j..]) == InBoth(a[i..], b[j + 1..]);
      j := j + 1;
    }
  }
}
```

A variation on Fibonacci

```
ghost function Fibly(x: nat): int {
  if x < 2 then x else Fibly(x - 2) - Fibly(x - 1)
}

method ComputeFibly(n: nat) returns (r: int)
  ensures r == Fibly(n)
{
  r := 0;
  var s := 1;
  for i := 0 to n
    invariant r == Fibly(i) && s == Fibly(i + 1)
  {
    r, s := s, r - s;
  }
}
```

Termination

- A `decreases` clause defines the termination metric for each method/function activation
- It is evaluated *on entry* to the method/function
- The termination metric from one method/function activation to the next must decrease in a well-founded order $\succ$
- Dafny fixes $\succ$

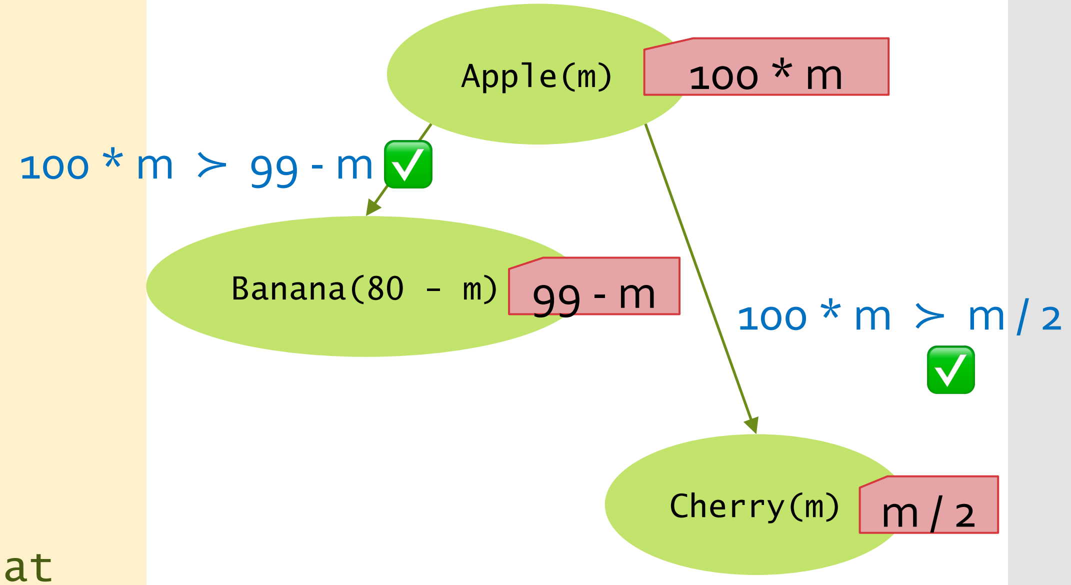
Defining termination metrics

```
method Apple(m: nat)
  decreases 100 * m
{
  if 1 <= m < 50 {
    Banana(80 - m);
    var c := Cherry(m);
  }
}

method Banana(n: nat)
  decreases n + 19
…

function Cherry(p: nat): nat
  decreases p / 2
…
```

For a general activation of Apple



Well-founded ordered for other types

type	$X \succ y$
inductive datatype	X structurally includes y
`int`	$0 \le X \;\land\; y < X$ equivalently: $0 \le X \;\land\; y + 1 \le X$
`real`	$0.0 \le X \;\land\; y + 1.0 \le X$
`bool`	$X \land \neg y$
`object?`	$X \neq$ `null` $\land\; y =$ `null`
`set`<T>	$y \subsetneq X$
`seq`<T>	Y is proper subsequence of X
`iset`<T>	`false`

Commutativity of `Mul`

```
opaque function Mul(a: nat, b: nat): nat {
  if b == 0 then 0 else a + Mul(a, b - 1)
}
```

How cool is this step? Yep, pretty cool, alright!

Here's another interesting step. Notice how the arguments are reversed in the recursive call (that is, to call to the induction hypothesis).

```
lemma {:induction false} MulCommutative(a: nat, b: nat)
  ensures Mul(a, b) == Mul(b, a)
{
  if a == b {
  } else if a == 0 {
    assert Mul(a, b) == Mul(a, b - 1) by { reveal Mul(); }
    assert Mul(b, a) == 0 == Mul(b - 1, a) by { reveal Mul(); }
    MulCommutative(a, b - 1);
  } else if b < a {
    MulCommutative(b, a);
  } else {
    calc {
      Mul(a, b);
      { reveal Mul(); }
      a + Mul(a, b - 1);
      { MulCommutative(a, b - 1); }
      a + Mul(b - 1, a);
      { reveal Mul(); }
      a + b - 1 + Mul(b - 1, a - 1);
      { MulCommutative(a - 1, b - 1); }
      a + b - 1 + Mul(a - 1, b - 1);
      { reveal Mul(); }
      b + Mul(a - 1, b);
      { MulCommutative(a - 1, b); }
      b + Mul(b, a - 1);
      { reveal Mul(); }
      Mul(b, a);
    }
  }
}
```

Shorter, but less readable, proof

```
function Mul(a: nat, b: nat): nat {
  if b == 0 then 0 else a + Mul(a, b - 1)
}

lemma MulCommutative(a: nat, b: nat)
  ensures Mul(a, b) == Mul(b, a)
{
  if 0 < a < b {
    MulCommutative(a - 1, b);
  }
}
```

INC.dfy $^0/_3$

Least predicates (predicates defined as a least fixpoint)

```
datatype Cmd = Inc | Seq(Cmd, Cmd) | Repeat(Cmd)

type State = int

least predicate BigStep(c: Cmd, s: State, t: State) {
  match c
  case Inc =>
    t == s + 1
  case Seq(c0, c1) =>
    exists s' :: BigStep(c0, s, s') && BigStep(c1, s', t)
  case Repeat(c0) =>
    || s == t
    || exists s' :: BigStep(c0, s, s') && BigStep(c, s', t)
}
```

Lemma about a least predicate

```
least lemma {:induction false}
    Monotonic(c: Cmd, s: State, t: State)
  requires BigStep(c, s, t)
  ensures s <= t
{
  match c
  case Inc =>
  case Seq(c0, c1) =>
    var s' :| BigStep(c0, s, s') && BigStep(c1, s', t);
    Monotonic(c0, s, s');
    Monotonic(c1, s', t);
  case Repeat(c0) =>
    if s != t {
      var s' :| BigStep(c0, s, s') && BigStep(c, s', t);
      Monotonic(c0, s, s');
      Monotonic(c, s', t);
    }
}
```

Same lemma, but with automatic induction

```
least lemma Monotonic(c: Cmd, s: State, t: State)
  requires BigStep(c, s, t)
  ensures s <= t
{
  // proof is automatic
}
```

Brief show and tell

- Possibly infinite unary numbers
  - `codatatype`
  - Co-recursive calls
  - `greatest lemma`
- Wildcard
  - Operational vs declarative specification

Conclusion

Use Dafny to

- Write verified programs
- Write a verified part of a larger program written in some other language
- Formalize your models
- Teach!

Program safely!

Additional examples

Pow

```
opaque ghost function Pow(n: nat): nat {
  if n == 0 then 1 else 2 * Pow(n - 1)
}

lemma PowPlus(m: nat, n: nat)
  ensures Pow(m + n) == Pow(m) * Pow(n)
{
  if m == 0 {
    assert Pow(0) == 1 by {
      reveal Pow();
    }
  } else {
    calc {
      Pow(m + n);
      { reveal Pow(); }
      2 * Pow(m - 1 + n);
      { PowPlus(m - 1, n); }
      2 * Pow(m - 1) * Pow(n);
      { reveal Pow(); }
      Pow(m) * Pow(n);
    }
  }
}
```

FastPow

```
function FastPow(n: nat): nat {
  if n == 0 then 1 else
    var p := FastPow(n / 2);
    if n % 2 == 0 then
      p * p
    else
      2 * p * p
}
```

```
lemma {:induction false} Same(n: nat)
  ensures Pow(n) == FastPow(n)
{
  if n == 0 {
    assert Pow(n) == 1 by {
      reveal Pow();
    }
  } else {
    var h := n / 2;
    Same(h);
    PowPlus(h, h);
    if n % 2 == 0 {
      assert n == h + h;
    } else {
      assert n == h + h + 1;
      assert Pow(n) == 2 * Pow(n - 1) by {
        reveal Pow();
      }
    }
  }
}
```

Sum

```
ghost function Sum(s: set<int>): int {
  if s == {} then 0 else
    var x := Pick(s);
    Sum(s - {x}) + x
}

ghost function Pick(s: set<int>): int
  requires s != {}
{
  var x :| x in s; x
}
```

```
lemma ExtendSum(s: set<int>, x: int)
  requires x !in s
  ensures Sum(s + {x}) == Sum(s) + x
{
  var s' := s + {x};
  var y := Pick(s');
  if x != y {
    var t := s - {y};
    calc {
      Sum(s');
      Sum(s' - {y}) + y;
      { assert s' - {y} == t + {x}; }
      Sum(t + {x}) + y;
      { ExtendSum(t, x); }
      Sum(t) + x + y;
      { ExtendSum(t, y); }
      Sum(t + {y}) + x;
      { assert t + {y} == s; }
      Sum(s) + x;
    }
  }
}
```

Optimize.dfy $^0/_2$

Optimize

```
datatype Expr =
  | Const(value: int)
  | Var(name: string)
  | Binary(op: BinOp, 0: Expr, 1: Expr)
{
  function Eval(env: Env): int {
    match this
    case Const(n) => n
    case Var(name) =>
      if name in env then env[name] else 0
    case Binary(op, e0, e1) =>
      op.Eval(e0.Eval(env), e1.Eval(env))
  }

  // …
}
```

```
type Env = map<string, int>

datatype BinOp = Plus | Minus
{
  function Eval(a: int, b: int): int {
    match this
    case Plus => a + b
    case Minus => a - b
  }
}
```

Optimize.dfy 1/2

Optimize

```
datatype Expr = // …
{ // …

  function Optimize(): Expr {
    match this
    case Binary(op, e0, e1) =>
      var e0, e1 := e0.Optimize(), e1.Optimize();
      if e0.Const? && e1.Const? then
        Const(op.Eval(e0.value, e1.value))
      else
        this
    case _ => this
  }

  lemma OptimizeCorrect(env: Env)
    ensures Eval(env) == Optimize().Eval(env)
  {
    match this
    case Binary(op, e0, e1) =>
      e0.OptimizeCorrect(env);
      e1.OptimizeCorrect(env);
    case _ =>
  }
}
```