

Connecting programs and proof

K. Rustan M. Leino



5 October 2026
SPLASH/ISSTA 2026 summer school
Oakland, CA, USA

Goals of tutorial

- Demonstrate writing and debugging proofs of
 - Imperative programs
 - Functional programs
 - Math theorems
- Entice you to use Dafny for your next programming and proof project

Language

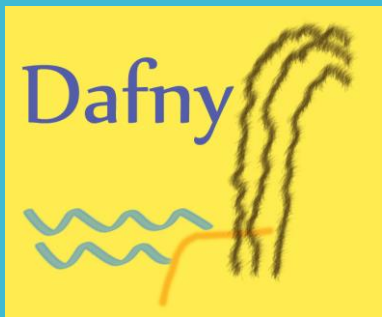
- Dafny (dafny.org)
- Programming language
 - Verification-aware (proof-oriented)
- Constructs for
 - Imperative programming
 - Functional programming
 - Specifications
 - Proofs
- VS Code integration



Tools

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





Some uses of Dafny

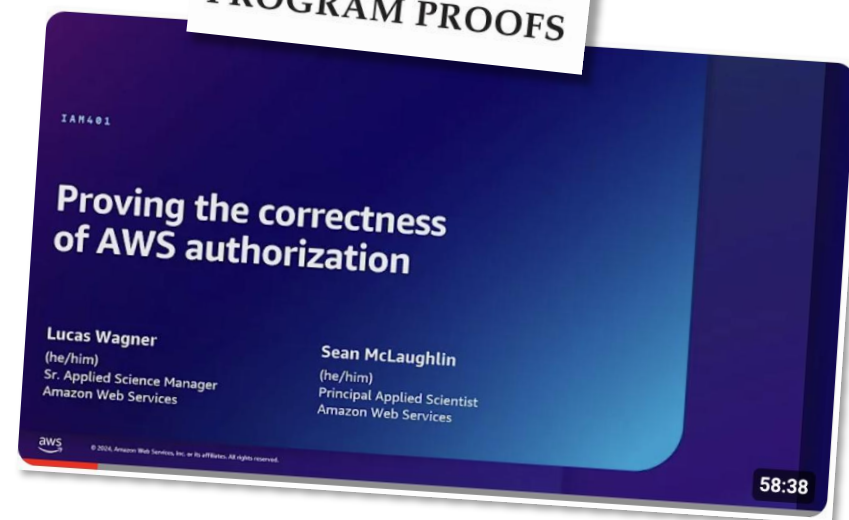
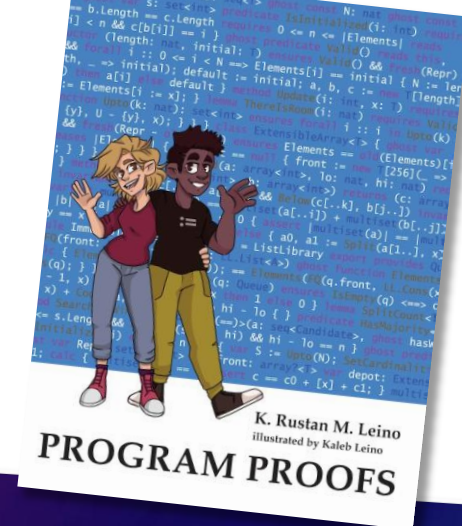
- Teaching
 - Over a decade, many universities

- Industrial use

- AWS's authorization engine
 - Paper at ICSE 2025.
 - Talk at AWS re:Inforce 2024:
<https://youtu.be/oshxAJGrwMU>

- Research projects

- Ironclad Apps, IronFleet, Armada, Vale, ...
 - Meta theory



1 Formalization of a Realistic Verification-Condition 2 Generator for an Intermediate Verification 3 Language

4 Vladimir Gladshtein ✉

5 National University of Singapore, Singapore

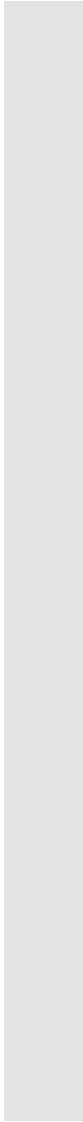
6 K. Rustan M. Leino ✉

7 Amazon Web Services, Seattle, WA, USA

ITP 2026

8 — Abstract —

9 Intermediate Verification Languages (IVLs) play the same role as
10 Representations of programs in the verification of programs.



Demonstrations

```
newtype int32 = x: int | -0×8000_0000 ≤ x < 0×8000_0000
```

```
datatype Option<T> = None | Some(value: T)
```

```
method BinarySearch(a: array<int>, key: int) returns (r: Option<int32>)
  requires a.Length < 0×8000_0000
  requires forall i, j :: 0 ≤ i < j < a.Length ⇒ a[i] ≤ a[j]
  ensures r.Some? ⇒ 0 ≤ r.value as int < a.Length && a[r.value] = key
  ensures r = None ⇒ 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..]
    {
      var mid := lo + (hi - lo) / 2;
      if key < a[mid] {
        hi := mid;
      } else if a[mid] < key {
        lo := mid + 1;
      } else {
        return Some(mid);
      }
    }
  }

  return None;
}
```

```
function Sum(s: seq<int>, lo: nat, hi: nat): int
  requires lo ≤ hi ≤ |s|
{
  if lo = hi then 0 else Sum(s, lo, hi - 1) + s[hi - 1]
}
```

```
method PrefixSum(a: array<int>)
  modifies a
  ensures forall i :: 0 ≤ i < a.Length ⇒ a[i] = Sum(old(a[..]), 0, i + 1)
{
  var ps := 0;
  for n := 0 to a.Length
    invariant forall i :: 0 ≤ i < n ⇒ a[i] = Sum(old(a[..]), 0, i + 1)
    invariant ps = Sum(old(a[..]), 0, n)
    invariant a[n..] = old(a[n..])
  {
    a[n] := ps + a[n];
    ps := a[n];
  }
}
```

```

method DetectDuplicate(a: array<int>, b: array<int>) returns (r: bool)
  requires forall i :: 0 ≤ i < a.Length ⇒ 0 ≤ a[i] < 65536
  requires b.Length = 65536 && a ≠ b
  modifies b
  ensures r ⇔ exists i, j :: 0 ≤ i < j < a.Length && a[i] = a[j]
{
  if 65536 < a.Length {
    PigeonHolePrinciple(65536, a[..]);
    return true;
  }
  for n := 0 to a.Length
    invariant forall i, j :: 0 ≤ i < j < n ⇒ a[i] ≠ a[j]
    invariant forall x :: 0 ≤ x < 65536 ⇒
      (x in a[..n] ⇔ 0 ≤ b[x] < n && a[b[x]] = x)
    {
      var x := a[n];
      if 0 ≤ b[x] < n && a[b[x]] = x {
        return true;
      }
      b[x] := n;
    }
  return false;
}

```

```

lemma PigeonHolePrinciple(n: nat, s: seq<int>)
  requires n < |s|
  requires forall i :: 0 ≤ i < |s| ⇒ 0 ≤ s[i] < n
  ensures exists i, j :: 0 ≤ i < j < |s| && s[i] = s[j]
{
  if exists i, j :: 0 ≤ i < j < |s| && s[i] = s[j] {
    return;
  }
  var x := n - 1;
  if x in s {
    var i :| 0 ≤ i < |s| && s[i] = x;
    var t := s[..i] + s[i+1..];
    PigeonHolePrinciple(n - 1, t);
  } else {
    PigeonHolePrinciple(n - 1, s);
  }
}

```

Conclusion

Use Dafny to

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

Program safely!