Reflections on automated software verification

K. Rustan M. Leino

23 March 2026
ETH Zurich
Zurich, Switzerland

Dafny: language

- Dafny (`dafny.org`)
  - 18 years old
- Programming language
  - General purpose
  - Designed to support verification
  - Designed to be familiar to mainstream developers
- Constructs for
  - Imperative programming
  - Functional programming
  - Specifications
  - Proofs

Dafny: tooling

- Automated deductive verifier
  - Built on Boogie, uses SMT
- VS Code integration
- Cross compilers
- AI friendly
  - LLMs do well with Dafny

Show and tell

Binary search

Wildcard match

Impact

- Industry
- Research
- Teaching
- Community

Impact: industry

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

  - AWS Encryption SDK
  - AWS Database Encryption SDK
  - …

💡 Verification can be done for performant, relevant applications
💡 Proofs can be done by software engineers

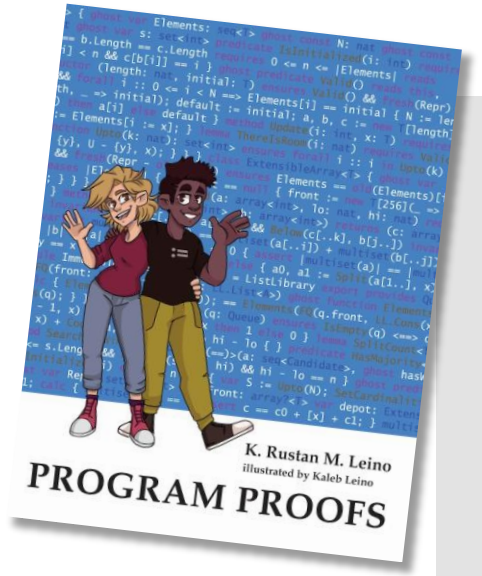
Impact: research

- Ironclad Apps
- IronFleet
- Vale
- Armada
- EVM formalization
- NFS/Daisy
- …

💡 Systems researchers can use verification effectively

Impact: teaching

- 15+ years, dozens of universities
  - The University of Queensland
  - MIT (Spring 2025)
  - …
- Books
  - *Program Proofs*, KRML (MIT Press, 2023) `program-proofs.com`
  - *Algoritmos y estructuras de datos: Con programas verificados en Dafny*, Ricardo Peña Marí (Ibergarceta Publicaciones, 2019)



💡 A language designed for verification does well in teaching

Impact: community

- Dafny workshop
  - 2024, 2025, 2026
  - co-located with POPL
  - for Dafny and Dafny-style languages/verifiers
- Appetite for verifiers of mainstream languages
  - Verus (Rust)
  - Viper-based tools (Rust, Go, Python)
  - Stainless (Scala)
  - SPARK toolset (Ada)
  - …
- LLM 💗 Dafny
  - 1000's of problems in *A benchmark for vericoding: formally verified program synthesis*, Bursuc et al., 2025
    - Dafny, 82%
    - Verus, 44%
    - Lean, 27%
  - (the similar language/verifier F* also does well with LLMs)
  - …

💡 Verification is becoming popular

Evolving from origins

- Immediate predecessors: ESC/Java … Spec# …
- Original motivation: new specifications of object data structures
  - No subclassing
  - No built-in `invariant` for objects
  - Modest need for language support
    - `modifies`/`reads` frames, sets, ghost variables
- No existing libraries, build systems, users
- 💡 Any new user is already interested in verification!

Dafny logic

- Higher-order, partial, classical logic with empty types
- Expressions are deterministic
  - User experience phases:
    obvious → subtle → questioning → at ease
- Expressions (and statements) are partial
  - Proof obligations for well-definedness 😊
  - Well-definedness properties "linger" in verifier 🤔

Dafny logic: references

- `new` yields a reference to a previously unreachable object
-
```
method M(c: C) {
  var d := new C;
  assert c ≠ d;
}
```
-
```
function P(): bool {
  forall c: C :: c.x == 10
}
```

💡 New-object allocation is important and tricky

Types

```
class Point {
  var x: int
  var y: int
}
```

- Mutable fields
- Identity
- Reference equality

```
datatype List =
  | Nil
  | Cons(head: int, tail: List)
```

- Immutable
- Structural equality

💡 A language does well to provide both of these directly

Types: members

```
class Point {
  var x: int
  var y: int
  function Sum(): int { … }
}
```

```
datatype List =
  | Nil
  | Cons(head: int, tail: List)
{
  function Sum(): int { …}
}
```

Types: traits (original)

```
trait Summable {
  function Sum(): int
}
```

```
class Point extends Summable {
  var x: int
  var y: int
  function Sum(): int { … }
}
```

```
datatype List =
  | Nil
  | Cons(head: int, tail: List)
{
  function Sum(): int { …}
}
```

Types: traits (general)

```
trait Summable {
  function Sum(): int
}
```

```
class Point extends Summable {
  var x: int
  var y: int
  function Sum(): int { … }
}
```

```
datatype List extends Summable =
  | Nil
  | Cons(head: int, tail: List)
{
  function Sum(): int { …}
}
```

💡 Design the language to treat types equally, not being biased for either kind of type

Subset types
(aka refinement types)
(aka constrained types)
(aka predicate subtypes)

- `type nat = x: int | 0 ⩽ x`
- ```
  var i: int
  var n: nat
  …
  i := n;  ✅
  …
  n := i;  ✅ Checked by verifier
  ```
- Type inference is difficult
- but essential, and cannot rely on first assignment

```
forall i, j :: 0 ⩽ i < j < a.Length ⟹ a[i] ⩽ a[j]
```

```
forall i: int, j: int :: 0 ⩽ i < j < a.Length ⟹ a[i] ⩽ a[j]
```

💡 Subset types are important, but I'm wishing for better understood type inference
```

Nullable vs non-null reference types

- `class Point { … }`

  gives rise to two types:
  - nullable reference type `Point?`
  - subset type
    `type Point = p: Point? | p ≠ null`
- Complicates type inference with reference traits
  - `Summable? :> Summable :> Point`
  - `Summable? :> Point? :> Point`
- 💡 A union type would be better, perhaps like
  - `type Point? = Point | {null}`

Uniform treatment of ghost code

- Variables, methods, functions, statements, expressions can be either
  - compiled or
  - ghost (used by the verifier, erased by the compiler)
- Ghosts can be used in unexpected ways:
  - Write specifications
  - Allow expressions that cannot be compiled
  - Define and prove lemmas
  - Call lemmas from compiled code
  - Tail recursion modulo ghosts
  - Specify and implement uninitialized storage
  - …

💡 All languages ought to support ghost code

Termination

- Once a skeptic, I am now a believer
- Part of well-formedness checks
- Prevents unsoundness of logic
  - Ghost code must always terminate
  - Prevents inconsistent function definitions
- Fixed well-founded order
- Default `decreases` clauses

💡 Termination specifications/checks are easy and are good to support

Termination: fixed well-founded order

Example:
`decreases hi - lo, a[lo]`

- Dafny fixes a well-founded order for each type, e.g.

type	X ≻ y
`int`	0 ≤ X ∧ y < X
`bool`	X ∧ ¬y
inductive datatypes	X structurally includes y
`set<T>`	y ⊊ X
`iset<T>`	`false`

- Its termination metrics use

$$\mathcal{C} \times \underbrace{\mathcal{A}_\mathsf{T} \times \mathcal{A}_\mathsf{T} \times \cdots \times \mathcal{A}_\mathsf{T}}_{k}$$

where
  - $\mathcal{A}$ is the union of all types
  - $\mathcal{C}$ is the set of strongest connected components (SCCs) of the call graph
  - k is the length of the longest given `decreases` clause in the program

- `decreases` clauses are automatically `T`-padded to the longest length

Termination: default `decreases` clauses

- `method M(x: X, y: Y, z: Z)`
  `decreases x, y, z` 💡 default
- `while a < b`
  `decreases b - a` 💡 default

Compiler(s)

- Dafny supports full compilation to 5 target platforms:
  - .NET
  - JVM
  - JavaScript
  - Go
  - Python
- Piggybacks on runtime systems of target platforms (garbage collector, …)
- Attracts users
- Burden to support
- Compilation is just part of the story – also need scaffolding
- Non-idiomatic target code
- 💡 Would do better with flagship compiler

Foreign function interface (FFI)

- Supported
  - `newtype int32 = x: int | -0x8000_0000 ≤ x < 0x8000_0000`
  - `{:extern}`
- but easy for users to make mistakes
  - `method Move(pt: Point) ensures pt.x == old(pt.x) + 1`
  - `method Get() returns (pt: Point)`
  - `method Write(stream: IO, txt: string) modifies stream`
  - `function Random(n: nat): (r: nat) requires 1 ≤ n ensures 0 ≤ r < n`

💡 Need better checks for FFI specifications

Verifier performance

- Often nearly instant
- 💡 Quick turnaround is crucial
- Automation uses
  - Automatic function unfolding, fuel, literal wrapping, can-call certificates, automatic induction, …
- 💡 Stability is important
- SMT community has not paid enough attention to deductive verification
  - Failing quickly is as important as succeeding quickly
  - Quantifier support

Verification stability: example

```
method M()
  requires A
{
  if B {
    … mentions X …
  } else {
    …
  }
  assert C;
}
```

Suppose the verifier knows how to use A to prove C, provided the context mentions X

In this branch, the verifier learns the lemma

$A \implies C$

💡 Clause learning needs to take "matching patterns" into account

Conclusions

- A full general-purpose programming language can be designed with powerful verification support
- Verifiers are being built for mainstream languages
- AI helps the case for verification
- Hope to see more verification influence on programming languages in the future

Program safely!