A Self-Verifying Quine in Lean 4

Formal Proof of Quineness via Elaboration-Time Computation
and the Side-by-Side Blueprint

Eric Vergo Claude (Anthropic)

2026

Abstract

We present a Lean 4 program that is simultaneously a quine (it outputs its own source code), a formal proof that the output equals the source (verified by the Lean kernel), and a self-referential artifact whose correctness proof is part of the quined output. The proof exploits Lean's elaboration-time computation to reduce string equality to definitional equality, allowing the kernel to verify the quine property via `rfl`. We further demonstrate that the program's blueprint annotations—the metadata that generates a Side-by-Side Blueprint dependency graph and this very paper—are themselves part of the quined source, closing a loop between formal verification, documentation, and self-reference. The entire construction lives in a single Lean file with no axioms, no `sorry`, and no `native_decide`.

1 Introduction

A *quine* is a program that, when executed, produces a copy of its own source code as output. The existence of quines is guaranteed by Kleene's recursion theorem [**?**], which shows that any sufficiently expressive formal system admits self-referential constructions.

The classical quine construction decomposes a program into two parts:

- A *prefix* containing the reconstruction logic.
- A *data string* d encoding the suffix of the source.

The program reconstructs its source as $\text{prefix} \| \text{quote}(d) \| d$, where quote adds the syntactic delimiters and escaping needed to embed d as a literal within the source.

Most quines are verified empirically: one runs the program and checks that the output matches the source. We ask a stronger question: *can the program itself contain a machine-checked proof that its output equals its source?*

In Lean 4, the answer is yes. We construct a single file that is:

1. A **quine**: compiling and running it produces an exact copy of the source.
2. A **formal proof**: the Lean kernel verifies that the quine formula equals the source content at type-checking time.
3. **Self-referential**: the proof text is part of the quined output, so the program proves a property about an artifact that includes the proof itself.

1.1 Contribution

We make the following contributions:

- A Lean 4 quine with kernel-verified correctness, using only `rfl` (definitional equality). No axioms, no `sorry`, no `native_decide`.
- Three custom elaborators (`include_str!`, `quine_formula!`, `file_contains!`) that perform computation at elaboration time, producing string literals that the kernel can compare.
- A demonstration that the program's Side-by-Side Blueprint annotations are part of the quined output, closing a self-referential loop between code, proof, and documentation.

2 Background

2.1 Kleene's Recursion Theorem

The key insight is the *Kleene trick*: given any total computable function f, there exists a program e such that the output of e equals f applied to e's own source code. For a quine, f is the identity function: the program outputs its source unchanged.

The standard construction works as follows. Let S denote the source code, decomposed as $S = P \cdot Q$ where P is the prefix and Q is the suffix. Define a data string d whose value, when unescaped, equals Q. Then:

$$S = P \cdot \text{quote}(d) \cdot d$$

where $\text{quote}(d)$ wraps d in delimiters and escape characters so that $\text{quote}(d)$ is the source-level representation of the string d.

2.2 Lean's Elaboration Pipeline

Lean 4 processes source files through an elaboration pipeline that resolves syntax, type-checks terms, and reduces definitions. Critically, Lean supports *custom elaborators*: user-defined macros that execute arbitrary `IO` actions during elaboration and return typed terms.

This means we can read files, compute string transformations, and inject the results as compile-time constants—all before the kernel sees the term. The kernel then compares these constants using definitional equality, which for string literals reduces to syntactic identity.

2.3 Side-by-Side Blueprint

The Side-by-Side Blueprint (SBS) is a toolchain for Lean 4 that generates interactive documentation from annotated source code. The `@[blueprint]` attribute marks declarations with metadata (title, statement, proof sketch, dependency edges), and the toolchain extracts this into a dependency graph, per-declaration HTML pages, and optionally a paper.

A key property of SBS is that the annotations live *in the source code itself*. They are Lean attributes processed at compile time. This means that for our quine, the blueprint annotations are part of the quined output. The documentation that describes the quine is itself described by the quine.

3 Construction

The entire construction lives in a single file, `Quine.lean`. We describe each component.

3.1 Custom Elaborators

Three elaborators perform computation at elaboration time:

`include_str!` Reads a file at elaboration time and returns its content as a string literal:

```
open Lean Elab Term in
elab "include_str!" path:str : term => do
  let content <- IO.FS.readFile path.getString
  return mkStrLit content
```

After elaboration, `include_str! "Quine.lean"` is indistinguishable from a 6,669-character string literal in the kernel's view.

`quine_formula!` Computes the quine output from the data string d alone, without reading the source file:

```
open Lean Elab Term Meta in
elab "quine_formula!" : term => do
  let fileContent <- IO.FS.readFile "Quine.lean"
  let marker := "def d := \""
  let some idx := findSubstr fileContent marker
    | throwError "marker not found"
  let pfx := (fileContent.take
    (idx + marker.length - 1)).toString
  let defn <- getConstInfoDefn `Quine.d
  let dVal <- whnf defn.value
  let .lit (.strVal dStr) := dVal
    | throwError "d is not a string literal"
  return mkStrLit (pfx ++ dStr.quote ++ dStr)
```

This elaborator extracts the prefix from the source, retrieves `d`'s value from the environment, computes `prefix ++ d.quote ++ d`, and returns the result as a string literal.

`file_contains!` Checks whether a file contains a given substring at elaboration time, returning `Bool.true` or `Bool.false`:

```
open Lean Elab Term in
elab "file_contains!" path:str ", " sub:str
    : term => do
  let content <- IO.FS.readFile path.getString
  let needle := sub.getString
  if (content.splitOn needle).length >= 2 then
    return mkConst ``Bool.true []
  else
    return mkConst ``Bool.false []
```

3.2 The Data String

The data string d encodes the suffix of the source file—everything after `def d := "..."`. In the actual source, d is a string literal spanning approximately 4,000 characters of escaped text. For

readability, we show it schematically:

```
def d := "<escaped suffix: ~4000 chars>"
```

The value of d, when Lean unescapes it, is exactly the source text from the newline after d's closing quote through the end of the file (including the closing `end Quine` and trailing newline). This is the Kleene payload.

3.3 The Quine Executable

The main function reconstructs the source without reading any files at runtime:

```
def _root_.main : IO Unit := do
  let file := include_str! "Quine.lean"
  let marker := "def d := \""
  let pfx := (file.take
    ((findSubstr file marker).get!
      + marker.length - 1)).toString
  IO.print pfx
  IO.print d.quote
  IO.print d
```

Note that `include_str! "Quine.lean"` is resolved at compile time. At runtime, `file` is simply a string constant baked into the binary. The function computes the prefix (everything up to and including the opening quote of d), prints it, then prints `d.quote` (the escaped representation of d), then prints d itself (the raw suffix). The concatenation equals the original source.

3.4 Bootstrapping

The data string d must be computed before the file can compile, since the suffix—which d encodes—includes d's own definition line (or rather, the text *after* it). This is resolved by a bootstrapping script that:

1. Writes the file with a placeholder for d.
2. Extracts the suffix (everything after the placeholder's closing quote).
3. Lean-escapes the suffix.
4. Rewrites the file with the escaped suffix as d's value.

The result is a fixed point: the suffix that d encodes is exactly the suffix that follows d in the file.

4 Proofs

4.1 Quine Correctness

The central theorem states that the quine formula equals the source:

```
theorem quine_correct :
    quine_formula! = include_str! "Quine.lean"
    := rfl
```

Both sides are elaborated to string literals before the kernel sees them. `quine_formula!` computes the quine reconstruction from d; `include_str!` reads the actual file. Since the bootstrapping establishes the fixed-point property, both literals are identical, and the kernel accepts `rfl`.

This is a genuine kernel-level proof. The elaborators perform the computation, but the kernel independently verifies that the two terms are definitionally equal. No axioms are invoked.

4.2 Why Not `decide`?

One might expect to prove string equality using `decide`, which asks the kernel to evaluate a decidable proposition. For our quine, this fails for two reasons:

1. **Kernel limitations on `partial` functions.** Lean's `String.splitOn` is marked `partial`, making it opaque to the kernel. Any proof strategy that requires the kernel to reduce `splitOn` will fail.

2. **Stack overflow on large terms.** Converting a ∼7,000-character string to `List Char` and recursing over it exhausts the process stack, even with generous recursion limits.

The elaboration-time approach sidesteps both issues: compiled code (not the kernel) performs the string operations, and the kernel only sees the final string literals.

4.3 Self-Reference

The quine output contains the text of its own correctness theorem:

```
theorem self_referential :
    (file_contains! "Quine.lean",
      "theorem quine_correct") = true := rfl
```

And the quine output contains its own blueprint annotations:

```
theorem annotations_self_referential :
    (file_contains! "Quine.lean",
      "@[blueprint") = true := rfl
```

Both proofs use the same trust model as `include_str!`: the elaborator performs the substring check using compiled code, returns `Bool.true`, and the kernel verifies `true = true` via `rfl`.

4.4 Post-Elaboration Verification

As a defense-in-depth measure, an `initialize` block runs after all declarations are elaborated and performs the full quine reconstruction check at load time:

```
initialize do
  let fileContent <- IO.FS.readFile "Quine.lean"
  let marker := "def d := \""
  let pfx := (fileContent.take
    ((findSubstr fileContent marker).get!
      + marker.length - 1)).toString
  let reconstructed := pfx ++ d.quote ++ d
  unless reconstructed == fileContent do
    throw <| IO.userError
```

```
10        "Post-elaboration quine verification failed"
```

This catches any scenario where the file was modified after bootstrapping but before the proofs were checked—a consistency safeguard.

5 The SBS Loop

The Side-by-Side Blueprint annotations in this file serve a dual purpose. Pragmatically, they generate the dependency graph and documentation pages. Logically, they are part of the quined output, creating a loop:

1. The **source code** contains `@[blueprint]` attributes.
2. The **quine output** reproduces the source, including those attributes.
3. The **blueprint toolchain** reads those attributes and generates a dependency graph and this paper.
4. The **paper** (this document) describes the quine, including its blueprint annotations.
5. The source **proves** that its output contains those annotations (`annotations_self_referential`).

This is not circular in a logical sense—the proof is checked by the kernel independently of the documentation pipeline. But it is self-referential in a meaningful way: the artifact that the quine produces includes the metadata that describes the artifact.

The dependency graph captures this structure precisely:

$$\texttt{find\_substr} \to \texttt{quine\_data} \to \texttt{quine\_main}$$

$$\texttt{quine\_data} \to \texttt{quine\_correct} \to \texttt{self\_ref} \to \texttt{annotations\_ref}$$

All six nodes are *fully proven* (no `sorry`, no unverified dependencies), and the graph is connected with no cycles.

6 Trust Model

The proof rests on the following trust assumptions:

- **Lean's kernel** correctly checks definitional equality of string literals. This is the core trust assumption of any Lean formalization.
- **Elaboration-time file reads** return the actual file content. The elaborators use `IO.FS.readFile`, which is trusted in the same way that `#eval` trusts `IO` actions. This is the same trust model as Lean's built-in `include_str` (which we reimplemented for pedagogical clarity).
- **The bootstrapping script** correctly computes the fixed point. This is a one-time offline computation, and its correctness is independently verified by the `initialize` block at load time.

Notably absent from the trust base:

- `native_decide`: We do not use native code evaluation for proofs. All kernel-level reasoning uses `rfl`.

- `sorry`: There are no admitted propositions.
- `Decidable` instances for string operations: We bypass the kernel's inability to reduce `partial` functions by performing computation at elaboration time instead.

7 Conclusion

We have shown that Lean 4's elaboration pipeline is expressive enough to construct a quine with a kernel-verified correctness proof, where the proof itself is part of the quined output. The key technique—reducing string computation to elaboration-time evaluation and presenting the kernel with pre-computed literals—is broadly applicable to any scenario where the kernel must reason about large concrete data.

The interaction with the Side-by-Side Blueprint adds a layer of self-reference that, while not logically necessary, demonstrates the composability of Lean's metaprogramming facilities: the same file can be an executable program, a formal proof, and a documented blueprint, with all three aspects entangled through the quine property.

References

[1] S. C. Kleene, "On notation for ordinal numbers," *Journal of Symbolic Logic*, vol. 3, no. 4, pp. 150–155, 1938.

[2] L. de Moura and S. Ullrich, "The Lean 4 theorem prover and programming language," in *CADE-28*, 2021, pp. 625–635.

[3] W. V. O. Quine, "The ways of paradox," in *The Ways of Paradox and Other Essays*, Random House, 1966, pp. 1–21.