# LEAN Proofs Small Enough to Read: Kernel Contracts from LaTeX Theorems to Compiler Decisions

Brent S. Hartshorn brenthartshorn@proton.me

September 9, 2026

**Abstract**

In September 2026 a computer-checked proof of Fermat's Last Theorem and a Lean-formalized proof of blowup in the forced Navier–Stokes equations were announced within days of each other, and both are believed because a kernel checked them. This paper applies that discipline at the other end of the scale. A Calculus of Constructions micro-kernel with no axioms, is given an imperative language—assignment, branches, bounded loops, and `while` with an explicit variant—compiled by symbolic execution into kernel terms. Contracts are decidable propositions, `Holds` $b :=$ `Eq Bool` $b$ `true`, about the very expression a runtime check would evaluate. A `while` loop raises four obligations, one of them the soundness of its own lowering, and termination is proved for every starting state. Records and a state invariant give a model of an seL4-style OS whose routing layer carries contracts that hold for every input. A bridge to Crust, a self-contained C/C++/Rust toolchain, settles its source-level contracts in the kernel; building it found two of Crust's passes reading one contract two ways. Finally a certificate changes generated code: a SIMD kernel's scalar remainder is omitted on it, and a contract proven at every call site downgrades 80% of the memory-safety checks in a leaf initializer while the check it leaves still catches an overflow.

## 1 Introduction

Anthropic (2026) reported that Claude, working largely autonomously over eleven days, had produced the first end-to-end computer-checked proof of Fermat's Last Theorem: thirteen million lines of Lean, 29,500 intermediate theorems, and—in Kevin Buzzard's words—"no assumptions other than the axioms of mathematics" [6, 7, 8]. Four days later OpenAI reported a proof that the Navier–Stokes equations, under a smooth applied force, can develop a singularity in finite time, released with a Lean formalization whose verification took a further seventeen hours [4]; the unforced problem remains open [5].

What the two share is not their subject but their epistemology. Nobody has read thirteen million lines of Lean. The proof is believed because a kernel checked it, and the kernel is trusted because it is small—the move the Simons Foundation called "from trust to verification" [11], and that Buzzard, watching AI systems settle Erdős problems faster than they could be posed, called being "outcounterexampled" [10].

This paper is about that shift at the other end of the scale. Follwing from a LaTeX subset to Python with a dependent-type micro-kernel that checks proofs about Python functions [1], and a C/C++/Rust toolchain small enough to read, whose memory-safety instrumentation omits what the compiler can prove away [2, 3]. The kernel could check a proof term; the compiler could decide, by hand-written dataflow, that a check was unnecessary; neither talked to the other. Now they do (Figure 1), and the contribution is the chain between them: an imperative fragment compiled into ordinary kernel terms (§5); a `while` rule that states the soundness of its own lowering as an obligation, and termination proved for every state (§6); records, a state invariant, and composition by chaining proofs (§7); a model of an seL4-style OS's routing layer, all four functions of which carry contracts that hold for every input (§8);

a bridge to Crust that found two passes reading one contract two ways (§9); and generated code that depends on a proof (§10).

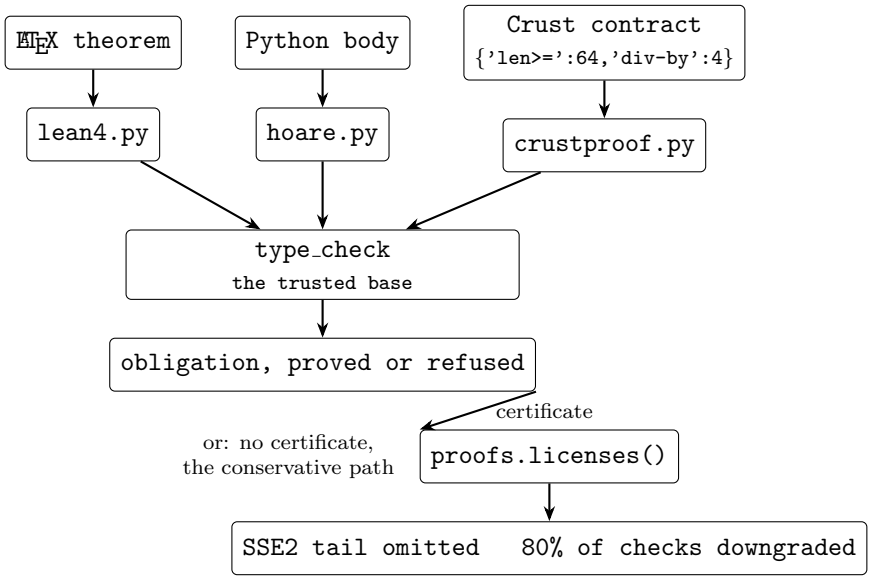


Figure 1: Three front ends, one checker, and the decision a compiler makes on the strength of what it checked.

Two disclaimers. The micro-kernel is named `lean4.py` for the discipline it follows and the statement language it reads, not because it is Lean; it is a Calculus of Constructions checker with de Bruijn indices and an untrusted elaborator. And the OS is a model: the scheme layer of `crustos` and the shape of a scheduler are modelled and proved about; nothing boots.

## 2 Background

### 2.1 RosettaMath and the micro-kernel

RosettaMath [1] is a self-hosting translator from a subset of LaTeX to Python, built for the gap between the notation mathematics is written in and the language a student can run. Alongside it is `lean4.py`, a Calculus of Constructions [16] checker in which the theorem statements are written in that same LaTeX subset:

```
@theorem(r'\forall x \in \text{Nat}, x = x')
def reflexivity(x: 'Nat'):
    return refl(x)
```

The decorator compiles the function to a kernel term, reads the statement as a type, and requires the two to agree definitionally. This inverts the Xena project's bridge, which rendered Lean proofs *into* LaTeX [12, 13]: here LaTeX is the input and the thing proved is a Python function. It is not Lean [15]; it is a kernel of the same design, small enough to be a teaching object.

The front end accepted exactly one `return`—enough for a proof term, not for a program. The previous paper closed by naming the next step: make the statement language the contract language of Crust, so that a contract is a Π-type and the obligation a theorem the kernel checks. "This is future work, and we do not claim more than the shape of it here." This paper is that work.

### 2.2 Crust and its contracts

Crust [3] is a C, C++ and Rust toolchain that owns its whole stack: every source language lowers to plain C, and one self-contained backend takes C to machine code, with no external compiler. Its memory-safety paper [2] argues that uniform instrumentation asks questions the compiler already knows the answers to, so a whole-program view should prove most checks away before they are emitted.

Contracts already exist in Crust's C. A source pre-pass lifts `assert` clauses out of the region between a function's parameter list and its body, blanks them space-for-space so byte offsets are preserved, and hands the result downstream as structured metadata:

```
int calc_sum(int *ptr, unsigned int len)
assert len(ptr) >= 64
assert not len(ptr) % 4
{ ... }
```

becomes {'ptr': {'len>=': 64, 'div-by': 4}}, which four whole-program passes consume: call-site violations become compile errors, proven alignment lets a reduction drop its scalar remainder, a declared core partitions the register file, and the memory-safety instrumentation omits what it can prove. The passes share a discipline—a proof that fails degrades to the conservative path—but, as §9 reports, not a definition.

## 3 The kernel, and three measured fixes

A term is a universe, a free or bound variable, an application, a metavariable, a Π or a λ, over de Bruijn indices. Equality is structural on the de Bruijn form, so it is $\alpha$-equivalence for free. Reduction is $\beta$, $\delta$ (unfolding a definition) and $\iota$ (a recursor meeting a constructor). `type_check` is the whole trusted base; the elaborator, which fills implicit arguments by Miller pattern unification, is untrusted, and the kernel re-checks its output from scratch. `inductive()` generates two recursors per family, `T.rec` into `Type 0` and `T.ind` into `Prop`, because there is no universe polymorphism. One restriction is stated rather than hidden: a recursive argument is refused in a family with indices, since the induction hypothesis would have to name that occurrence's indices.

The imperative fragment was blocked before it began. `leb 20 30` did not terminate. The cause took three passes, and each was measured before it was fixed (Table 1).

| symptom | cause | fix |
|---|---|---|
| cost $2^n$; normal forms $14n+3$ | pure recomputation | memoise `normalize` per environment generation |
| still exponential | substitution *shares* the inserted term, so a body using its argument three times builds a DAG with $3^n$ paths and $O(n)$ nodes; hashing a nested tuple walks paths | structural keys become interned integers |
| 4.5M `shift` and 2.7M `instantiate` calls | the DAG re-expanded into a tree on every substitution | memoise the de Bruijn rewrites, so a hit returns the same object |

Table 1: Three fixes to `normalize`. `modb 12`: 134 s to 0.003 s. `modb 32`: never, to 0.046 s. Each was gated on byte-identical selftest and demo output.

Keying the caches on the structural key silently reprinted `symm`'s type, turning {`a : A`} into `a : A`: the structural key is blind to binder hints and implicitness—that is what makes it $\alpha$-equivalence—but the rewrites carry both through. A second key, `fullkey()`, draws the line where the rewrites do.

### 3.1 The statement language, read backwards

The LaTeX subset was an input language only. `type2latex` makes it an output language as well, and the property that makes that worth having is `latex2type`(`type2latex`($t$)) $= t$ for every term the kernel checks. It is written against the parser rather than against taste: every shape it emits is emitted because the parser accepts it, and anything the parser could not read back is refused with the reason—a hole, a loose index, `Type 1`, a name like `N` that the reader would alias away—rather than approximated. The existing pretty-printer is not this function and cannot become it: it prints `Type 1`, `?A7`, $\lambda$ and `x'`, none of which the reader takes.

Two places where the term does not determine the text. A binder's name is free to change, since the structural key ignores name hints, but only within what the lexer can read: `Nat` lexes as three tokens and a binder name is taken as one, so a bound name must be a single letter, and the letter chosen must also avoid the free names of the body or the reader would abstract two variables into one. And `a = b` is written only when the parser's recovery of `Eq`'s carrier—from the binder that introduced an operand—would return this very carrier; otherwise the application is written out, which always reads back.

**The printer found what the language was missing.** Of the 91 declarations the environment holds, five had types it refused, and they are most of Table 4: `snoc_le`, `stuck`, `fold_preserves`, `fold_terminates`, `loop_preserves`. Every one for the same reason, a $\lambda$ inside a *type*: an induction motive in the first, the `Nat.rec` of §6 in the rest. Normalising removes neither. So the subset gained $\lambda x : A,\ b$ on both sides at once, which is what keeps the two inverse; every type and every value in that environment, 141 terms, now round-trips, and Appendix A is written in LaTeX on the strength of it. A latent bug came out with it: the parser stopped an application at a `\forall` but an atom did not, so an arrow whose right side began with one read the binder as a variable *named* `forall`, and failed later, somewhere confusing.

## 4 Contracts as decidable propositions

The natural encoding of $\leq$, an inductive family, is refused by `inductive()`, and the refusal is deliberate. Rather than route around it:

$$\texttt{Holds}\ b\ :=\ \texttt{Eq}\ \texttt{Bool}\ b\ \texttt{true}.$$

A contract is a `Bool`-valued expression, and the proposition is that it computes to `true`. Every bound Crust's `extensions.py` parses is already decidable, because a contract a compiler cannot *check* is no use to it. So the proposition is about the same expression the runtime check would evaluate, and the proof is what licenses deleting the check. Closed instances need no proof at all: they compute, and `refl` is the proof.

## 5 The imperative fragment

`PythonToLean` accepts exactly one `return`. `ImpToLean` accepts a body. It is symbolic execution, not a semantics: there is no state type, no variable type, no big-step relation. The store is a dictionary from a variable to the term it currently holds, so assignment is a substitution performed in the compiler rather than a `let` the kernel would have to understand (Table 2). Nothing here is trusted; whatever it produces is handed to `type_check`, which knows nothing of Python.

### 5.1 The `while` rule

A while loop terminates for a reason the text does not state, so lowering one unannotated means inventing a variant—a guess where a guess is an unproved theorem. An annotation supplies it:

| Python | becomes |
|---|---|
| `v = e` | substitution into the store |
| `if c: ...else: ...` | `ite` per assigned variable |
| `for i in range(n)` | `Nat.rec`—the fold it always was |
| `while c` with a variant | iterating a guarded step; fuel is the variant on entry |
| several loop-carried variables | packed into a right-nested `Prod` |
| `c.field = e` | `c = Record.with_field c e` |
| `out: 'Array' = []` | `nil` at the annotation's element type |

Table 2: The lowering.

```
while c.current < c.nthreads:
    assert invariant(c.current <= c.nthreads)
    assert variant(c.nthreads - c.current)
    c.current = c.current + 1
```

The condition, invariant, variant and one pass are each named as functions of the state, and the fold is built from those names, which is what lets a general lemma about folds apply to a particular loop. Four obligations are raised (Table 3), and the soundness of the lowering is one of them: there is no metatheorem in a comment saying the fold equals the loop.

| obligation | says |
|---|---|
| `progress` | the condition really is false when the fold is done |
| `invariant holds on entry` | `Holds` $(inv\ s_0)$ |
| `invariant is preserved` | $\forall s$, `Holds` $(inv\ s) \to$ `Holds` $(cond\ s) \to$ `Holds` $(inv\ (pass\ s))$ |
| `variant decreases` | $\forall s$, $\ldots \to$ `Holds` $(\texttt{ltb}\ (rank\ (pass\ s))\ (rank\ s))$ |

Table 3: What a `while` loop owes.

Two hypotheses, not one `andb`: a case split on the condition then has `Holds true` to hand, which `refl` proves, whereas `Holds` $(\texttt{andb}\ I\ \texttt{true})$ with symbolic $I$ has nothing to reduce. Only one form composes; the fact was found by failing to prove `guarded` with the other.

### 5.2 What is refused

Roughly a third of the fragment's 157 checks are refusals, each pinned to its reason: `while` without a variant; `return` inside a loop (a fold cannot stop early) or a branch (the join would have to know which ran); an iterable that is not `range(n)`; an unannotated parameter; a variable assigned on one branch only, or changing type; division; a postcondition that is not `Bool`; an empty list with no annotation; a missing field. Once the state invariant exists: a syscall that can break it, a suffix that breaks it, a variant that does not come down, and a field name never bound—which raises at once rather than falling through, because a mistake in the caller's own lemma is theirs to see. The refusals are the deliverable as much as the acceptances: a fragment that accepted `while` without a variant would be lowering a guess.

# 6 Loops, proved

## 6.1 Two definitions that computed the right answer

`iter` ($n$+1) $s$ is `iter` $n$ (`step` $s$), not `step` (`iter` $n$ $s$). Both compute the same thing, but the variant comes down on the *first* pass, so that is where an induction on termination has to be able to look. Likewise `sub` recurses on its first argument, so `sub` ($m$+1) ($n$+1) is `sub` $m$ $n$ by definition; iterating `pred` on the second argument gives the same numbers and no such equation, and leaves a proof about an $n - i$ variant with nothing to induct on. A definition that computes the right answer can still be the wrong definition to reason about.

## 6.2 The lemmas

| lemma | proved by |
|---|---|
| `absurd` : $\forall C$:`Prop`, `Holds false` $\to C$ | `Eq.ind` into a motive reading `true` as the goal |
| `leb_refl`, `leb_succ` | induction |
| `leb_trans` | induction on the first argument, case split on the other two |
| `sub_le`, `sub_lt` | induction |
| `snoc_le` | induction on the list, case split on the bound |
| `andb_both`, `andb_left`, `andb_right` | `Bool.ind` |
| `guarded` | a case split on the condition |
| `fold_preserves` | induction on the number of passes |
| `loop_preserves` | the two composed |
| `stuck` | once the condition is false, iterating changes nothing |
| `fold_terminates` | the variant bounds the number of passes |

Table 4: Everything a `while` loop appeals to. None is an axiom.

`fold_terminates` is the one that matters most. `progress` used to be discharged only at concrete values, which checks the examples and says nothing about the rest. It is now proved for every starting state, and the environment in which it is proved contains 91 declarations—7 inductive families, 10 constructors, 14 recursors, 59 definitions—and no axioms.

# 7 State: records, an invariant, composition

A single-constructor inductive is a record; what `inductive()` does not give is names. `record(env, name, fields)` generates, per field, `Name.field` : `Name` $\to T$ and `Name.with_field` : `Name` $\to T \to$ `Name`, both by $\iota$ on the one constructor, so both compute. In the fragment, `c.ticks = c.ticks + 1` lowers to `Context.with_ticks` $c$ (`add` (`Context.ticks` $c$) 1). Nothing is mutated: a syscall takes a context and returns one, which is what makes state threadable through a sequence of them.

`state_invariant(env, 'Context', 'c.current <= c.nthreads')` defines `Context.invariant` : `Context` $\to$ `Bool`, a property of the state rather than of any operation, which is the seL4 shape [17]. `@procedure(preserves='Context')` then lets a body assume it and obliges the syscall to hand it on:

$$\forall c,\ \texttt{Holds}\,(I\ c) \to \texttt{Holds}\,(I\ (\texttt{tick}\ c)).$$

**Composition chains proofs rather than re-proving.** If $f$ hands the invariant on and $g$ hands it on, then $g \circ f$ does, and the term that says so is the two proofs applied in turn:

$$\lambda c\ h.\ \mathit{tick}_{\mathrm{pf}}\ (\texttt{enqueue}\ (\texttt{tick}\ c))\ (\mathit{enqueue}_{\mathrm{pf}}\ (\texttt{tick}\ c)\ (\mathit{tick}_{\mathrm{pf}}\ c\ h)).$$

The kernel checks the chain.

**The sequence rule.** A procedure with a loop is three steps—before, the loop, after—and each must hand the invariant on. The suffix is recovered as a function of the loop's result by `replace_subterm`, safe because the target is always an applied loop name with no bound variables of its own. Segments are named and skipped when definitionally the identity. A correct loop followed by one increment too many is refused at exactly the right place:

```
what runs after the loop in bad_suffix changes what the Context invariant reads, so the
hypothesis going in is not a proof of the conclusion coming out.
```

A whole-procedure failure would have been much less useful.

### 7.1 Why every proof goes through a case split

`Context.current` (`Context.with_ticks` $c$ $v$) does not reduce on a symbolic $c$: until the record is known to be built by its constructor, $\iota$ has nothing to fire on. So even an operation that plainly leaves a field alone has nothing to compute with. One split on the one constructor unsticks every projection at once. The subject need not be a record: a loop carrying two variables carries a `Prod`, and the split that unsticks a record's projections unsticks `fst` and `snd` for the same reason.

## 8 The OS model

Crust's `crustos` is a Redox-shaped OS model in which every resource is a URL and an `open` is routed to whichever *scheme* claims the prefix. The routing lives in `crustos/schemes.py`, in RPython, because it is text handling and table lookup—which is also what the fragment had to learn to read.

`Context` has six fields: `frames`, `queue`, `schemes`, `current`, `nthreads`, `ticks`. `List` is polymorphic, so a byte string is `List Nat` and a list of them is `List (List Nat)`; on that, `find`, `take`, `drop`, `eqs`, `split`, `append`, `snoc` and `rev`. `find` returning the length when the separator is absent is how `str.find`'s $-1$ is expressed without a negative number `Nat` does not have. All four public functions of `schemes.py` are modelled from the real source.

The contracts are the safety-relevant ones, and all four are proved: `scheme_of`'s returned index never leaves the scheme table; `path_of`'s result never grows; `route_all` returns one id per URL, in order; and `accepted`'s result is never longer than its input, *for every input*.

### 8.1 `accepted`, end to end

```
@procedure(ensures=['len(result) <= len(split(urls, 44))'])
def accepted(names: 'Strs', urls: 'Bytes') -> 'Array':
    parts = split(urls, 44)
    out: 'Array' = []
    i = 0
    while i < len(parts):
        assert invariant(len(out) <= i and i <= len(parts))
        assert variant(len(parts) - i)
        if scheme_of(names, parts[i]) < len(names):
            out = snoc(out, i)
        i = i + 1
    return out
```

The invariant is a conjunction because `i <= len(parts)` alone gives termination and not the result. Entry computes, whatever the input. Preservation splits on the `Prod` the loop carries and then on the `ite` the `if` left behind: an accepted URL grows the list by one as $i$ grows (`snoc_le`); a rejected one moves only $i$ (`leb_trans`, `leb_succ`); `andb_both` rejoins the halves. The variant is `sub_lt`. `progress_by_loop` gives termination for every input, and `invariant_at_exit` gives the invariant at the value the loop produced, from which the postcondition follows:

$$\forall\, \mathit{names}\ \mathit{urls},\ \texttt{Holds}\,(\texttt{leb}\ (\texttt{len}\ (\texttt{accepted}\ \mathit{names}\ \mathit{urls}))\ (\texttt{len}\ (\texttt{split}\ \mathit{urls}\ 44))).$$

One utility was load-bearing: `normalize` is all or nothing, and asked to look inside the invariant it also unfolds `andb` and `ite` into their recursors, leaving no conjunction to take apart. `unfold` expands only the definitions it is told to.

# 9 Integration with Crust

`crustproof.py` turns a Crust contract into a term of type `Nat` → `Bool`: {`'len>=': 64, 'div-by': 4`} becomes `andb` (`dvdb` 4 $n$) (`leb` 64 $n$). At a known length it reduces, and when it reduces to `true` a proof term is built and type-checked.

## 9.1 What the differential test found

Crust's two consumers read one contract two ways. `simd_contracts._satisfies` conjoined every clause; `contracts._violates` returned on the first clause present. For the contract in Crust's own SIMD documentation, a length of 70 cleared `len>=` so the `div-by` was never looked at: the call compiled, while `simd_contracts` correctly refused to prove it and kept the scalar tail. The pass that reports errors was the lenient one. Over a grid of six contracts and 130 lengths, 110 of 780 cases differed, all of them multi-clause. Crust now has one reading, in `shivyc/proofs.py`, which both passes call and which raises on a clause it cannot read rather than skipping it; the diagnostic names the clause the length actually breaks, which the first-clause reading could not do. `crustproof.py` asserts the kernel agrees with both passes on every case and pins the old reading as a regression.

## 9.2 Literals

Certification is on by default. It was not, because settling a contract at length 64 took 1.35 s and at 1024 took 43 s: the kernel's numerals are unary, and the cost grew with the length rather than the size of the number. A computation rule is now attached to nine definitions that fires only when the arguments have already reduced to numerals; anything else falls through to ordinary $\delta$, so every proof by induction is untouched. This is an extension of the trusted base—Lean does the same for `Nat`, for the same reason—and what keeps it honest is a differential test of each accelerator against its definition. That test earned its place at once: `modb` $n$ 0 is $n$, because the definition counts up and never meets a zero divisor to reset at, and the first accelerator said 0.

| length | before | after |
|---|---|---|
| 64 | 1.35 s | 0.001 s |
| 1024 | 43 s | 0.005 s |
| 65536 | — | 0.38 s |

Table 5: Settling {`'len>=': 64, 'div-by': 4`} at a length.

Making the rule fire took two attempts. `normalize` unfolds a definition's value the moment it meets the bare name, so a rule on a definition never sees its arguments; the declaration has to stop offering a value and drive $\delta$ from the rule. Then a partial application still unfolded to a lambda, which the caller $\beta$-reduced, so the full application—the only place the arguments are all visible—was never reached. Under-application now stays stuck on purpose.

# 10 A proof changes generated code

## 10.1 A scalar loop, emitted or not

`simd_contracts` drops a SIMD kernel's scalar remainder loop when the contract holds at every call site. Satisfying it is no longer on its own the licence: `proofs.licenses()` also wants the certificate, and without one the tail stays.

```
int calc_sum(int *ptr, unsigned int len)
assert len(ptr) >= 64
assert not len(ptr) % 4
{ int v = 0; unsigned int i;
  for (i = 0; i < len; i = i + 1) v = v + ptr[i];
  return v; }
```

```
default              proven at all 1 call site(s) (kernel-checked);
                     scalar fallback omitted
CRUST_PROOF_MAX=8    not proven (aligned but unproved (no certificate));
                     keeping scalar code
```

`calc_sum` comes out as 22 instructions using `paddd` and `movdqu` in the first case and 40 scalar instructions in the second. Three tests pin it, including that withdrawing the certificate withdraws the SSE2—otherwise the certificate would be decoration.

## 10.2 A runtime check, removed

`memsafe_elide` proves away what it can of the checks `--mem-safe` emits. Its rule 2 needs a live allocation of known size, and a parameter has none—the static pass does not follow a pointer into a callee, which was checked rather than assumed. So this is a new rule.

> **Rule 4 — a parameter with a proven contract.** `assert len(p) >= 64` on a parameter is a statement about every caller, and Crust can check it against every caller. Where it holds at all of them, the callee may treat `p` as an allocation of at least that size, and a constant offset into it is in bounds by the same arithmetic rule 2 uses.

`len` counts elements, so the byte extent is `len` × `sizeof(*p)`, converted once next to the element size it needs. Two conditions, both about not being wrong: the contract must be established—every visible call site traced to an allocation large enough, plus a certificate—and the callee must make no calls at all, since nothing supplies liveness for a parameter and the only safe substitute is a body in which nothing could have freed the buffer.

```
void fill(int *p) assert len(p) >= 4
{ p[0] = 1; p[1] = 2; p[2] = 3; p[3] = 4; }
```

goes from 5 checks emitted and none avoided to 1 emitted, 4 downgraded to a shadow update, 80% avoided:

```
before   5 check(s) emitted, 0 removed, 0 downgraded to a shadow update (0% avoided)
after    1 check(s) emitted, 0 removed, 4 downgraded to a shadow update (80% avoided)
```

A proved write becomes a bare shadow update rather than nothing, for the reason rule 2 already gives: the check is also what records which bytes are now defined.

The soundness tests are the point. No contract, a caller that allocates less, a callee that calls anything, a withdrawn certificate: all refuse, 0%. And the checks left behind still work. Add `p[4] = 5`: four writes are downgraded, the fifth check stays, and at run time:

```
crust --mem-safe: heap buffer overflow
  at f_over.c:4 in fill
    write of 4 bytes at 0x40af62b0
    object: 16 bytes, malloc (heap)
    offset 16 into a 16-byte object: the write begins at the first byte past the end
```

Let the caller pass two elements instead: the contract is not established, all five checks stay, and the runtime reports the overflow at `p[2]`.

One fix was not about proofs. `p[2]` is emitted as $\texttt{Add}(p, \texttt{Mult}(2, 4))$: the offset is constant but not a *literal*, and the origin tracker walked past it, so rule 4 registered its extent and elided nothing on the first attempt. Folding constants through `*`, `+` and copies—for values assigned once, since IL values are not SSA—fixed it, and rule 2 had been missing every `a[3]` into a malloc'd array for the same reason.

## 11 What is trusted, and what is not done

- **The trusted base** is `type_check` and the nine literal accelerators. The accelerators are Python the kernel takes the word of; they are checked against their definitions over a grid, and checked is not proved.
- **The OS is a model.** `crustos/kernel.c` does not boot, has no MMU and no interrupts. The scheme layer and a scheduler's shape are modelled, not a system.
- **Rule 4 is leaf-only and constant-offset-only.** Crust sees the whole call graph, so "no callee transitively frees this" is knowable and would widen the rule; "no calls at all" is what is implemented. `calc_sum` itself, a loop over a variable index, does not benefit.
- **`route_all`'s contract is checked at a batch.** Its `for` loop carries no invariant annotation, so there is nothing to carry.
- **The kernel is not Lean.** The statement language and the discipline are borrowed.

## 12 Discussion

Thirteen million lines of Lean at one end; 2,793 lines of Python at the other. The two are not in competition; they are the halves of one argument. A proof is believed because a kernel checked it, and a kernel is believed because it can be read. The first half is what the FLT and Navier–Stokes results demonstrate at a scale nobody expected this year—the FLT formalization was checked by a second, independent kernel precisely because the kernel is the only thing anyone must believe. The second half is what a checker of 2,793 lines is for, and it is the half that does not get easier as proofs get larger.

The most useful thing this integration produced was a disagreement rather than a proof. Two passes in a toolchain that had passed 495 tests read one contract two ways, and the difference showed only when a third reading—a term in the calculus of constructions—was set beside both. Buzzard writes that human mathematicians are being outcounterexampled [10]; here the counterexample was a length of 65, found by a differential test that took an afternoon. A specification that lives in two places is two specifications.

On attribution: the FLT formalization rested on Buzzard's project, Mathlib and Prove2Me [9]; this work rests on ShivyC, the two prior papers, and a design—kernel-checked, axiom-free, elaborator untrusted—that is Lean's. The micro-kernel is named for that debt. And on what it means for a learner: Wegerif asks what education becomes when the reason to believe something is no longer that a trusted person said so [14]. A student who can read `type_check` can see what "the kernel checked

it" means, and read the news with that understanding rather than on faith. That the same checker changes what a C compiler emits is the demonstration that verification is not a subject but a discipline, running from a theorem in LaTeX to the instruction a machine executes.

# A The OS model as one proposition

Everything §7 and §8 prove about `crustos` is two theorems, and their conjunction is a proposition the kernel checks. Equation (1) is that conjunction, with the braces expanding each name into what it stands for, down to the fold a `while` lowers to.

It can be written at all because the statement language now reads and writes a $\lambda$ (§3). Every fragment below is `type2latex` output, so every fragment reads back as the term it came from; the equation abbreviates only the names, per Table 6, and curries the fold's three binders into one $\lambda$. Listing 1 gives one of them unabbreviated, exactly as printed.

$$\overbrace{\underbrace{\forall c \in \mathtt{Ctx}}_{\text{every state, not a tested one}}\ ,\ \mathtt{Holds}\left(\ \underbrace{I}_{\lambda c,\ \mathtt{leb}\,(c.\mathtt{cur})\,(c.\mathtt{n})}\ c\right)\ \to\ \mathtt{Holds}\left(I\ (\ \underbrace{\mathtt{tick}\,(\mathtt{sched}\,(\mathtt{tick}\ c))}_{\mathtt{run}\ =\ \text{three syscalls, proofs chained}}\ )\right)}^{\textbf{the state layer}\ (\S7)}$$

$$\text{where}\quad \mathtt{sched}\ c\ =\ \underbrace{\mathtt{Nat.rec}\ (\lambda n,\ \mathtt{Ctx}\to\mathtt{Ctx})\ (\lambda s,\ s)\ \mathtt{step}\quad \underbrace{(\mathtt{rank}\ c)}_{\mathtt{sub}\,(c.\mathtt{n})\,(c.\mathtt{cur})}\quad c}_{\mathtt{sched.loop1}:\ \text{the variant bounds the passes, so the fold is the loop}} \tag{1}$$

$$\mathtt{step}\ =\ \lambda k\ ih\ s,\ ih\left(\mathtt{ite}\ \mathtt{Ctx}\quad \underbrace{(\mathtt{cond}\ s)}_{\mathtt{ltb}\,(s.\mathtt{cur})\,(s.\mathtt{n})}\quad (\mathtt{pass}\ s)\ s\right)$$

$$\overbrace{\wedge\ \forall\, ns\ us,\ \mathtt{Holds}\left(\mathtt{leb}\left(\mathtt{len}\ \underbrace{(\mathtt{accepted}\ ns\ us)}_{\text{a while carrying a Prod}}\right)\left(\mathtt{len}\ \underbrace{(\mathtt{split}\ us\ 44)}_{44\ \text{is}\ ,}\right)\right)}^{\textbf{the routing layer}\ (\S8)}$$

Read from the outside in. The first conjunct is preservation: assume the invariant of an arbitrary context, and it still holds of the context the three composed syscalls hand back. The `where` line is the only place a $\lambda$ is unavoidable — `Nat.rec`'s motive is one, and so is each minor premise — which is why five of the environment's declarations had no LaTeX spelling before this paper. The second conjunct is `accepted`'s postcondition from §8.1, quantified over every input rather than checked at a batch.

Nothing in the equation is a metatheorem stated in prose. `cond`, `pass` and `rank` are separate definitions in the environment because a general lemma about folds can only apply to a particular loop if that loop's parts are named; `fold_terminates` and `loop_preserves` are the lemmas, and the four obligations of §5.1 are what connect them to this `sched`.

| in (1) | in the environment | is |
|---|---|---|
| `Ctx` | `Context` | a six-field record |
| $I$ | `Context.invariant` | $\lambda c$, `leb` $(c.\mathtt{cur})$ $(c.\mathtt{n})$ |
| $c.\mathtt{cur}$, $c.\mathtt{n}$ | `Context.current`, `Context.nthreads` | projections, by $\iota$ |
| `cond`, `pass`, `rank` | `sched.cond1`, `sched.pass1`, `sched.rank1` | the loop, in parts |
| `tick` (second use) | `tick2` | see below |

Table 6: The equation shortens names and nothing else.

Two honest notes. `run` composes `tick`, `sched` and `tick`, but the environment holds the second under the name `tick2`: `declare` refuses to rebind a name, so a section that has already defined a `tick` gets a fresh one rather than a silent overwrite. And `sched.pass1` is written out in (1) as `pass`; in full

it applies `Context.with_current` and then reads `ticks` back off the result, because an assignment is a substitution in the compiler and two assignments to the same context nest:

Listing 1: `type2latex(value_of(env, 'sched.pass1'))`: line-wrapped to fit, and otherwise character for character what the printer emitted.

```
\lambda x : \text{Context}, \text{Context.with\_ticks}
  (\text{Context.with\_current} x (\text{add} (\text{Context.current} x) 1))
  (\text{add} (\text{Context.ticks}
     (\text{Context.with\_current} x (\text{add} (\text{Context.current} x) 1))) 1)
```

Handed back to `latex2type`, that string is the same term again. The whole environment behind (1) — 141 types and values — has that property, which is what makes an appendix written in LaTeX a claim about the model rather than a description of it.

## Reproducibility - Source Code

https://github.com/brentharts/RosettaMath
https://github.com/brentharts/crust

```
python3 lean4.py --selftest    # 177 checks; the demo output is byte-identical
python3 hoare.py               # 157 checks, about a third of them refusals
python3 crustproof.py          #  21 checks, including the differential test
make test_crust                # 495 tests; plus test_contracts (15),
                               # test_mem_safe_elide (34), test_metamorphic_simd (10)
```

## References


[1] Hartshorn, B. S. (2026). RosettaMath: Reading, Running and Proving Mathematics from LaTeX. Zenodo. https://doi.org/10.5281/zenodo.22646969

[2] Hartshorn, B. S. (2026). Memory Safety Where it is Needed: Proof-guided Runtime Checking in a Toolchain Small Enough to Read. SSRN. https://dx.doi.org/10.2139/ssrn.7396160

[3] Hartshorn, B. S. (2026). A Successor Discipline, Not a Successor Language: Safety by Subtraction in a Self-Contained C++ Toolchain. SSRN. https://dx.doi.org/10.2139/ssrn.7382398

[4] OpenAI (2026). On the Navier–Stokes Millennium Prize Problem. 8 September 2026. https://openai.com/index/navier-stokes-solution/ — Lean proof at https://github.com/openai/NavierStokesAndEuler

[5] McGreivy, N. (2026). How OpenAI found a singularity in Navier–Stokes. 8 September 2026. https://nickmcgreivy.substack.com/p/how-openai-found-a-singularity-in

[6] Anthropic (2026). Formalizing Fermat's Last Theorem. 4 September 2026. https://www.anthropic.com/research/formalizing-fermats-last-theorem

[7] *New Scientist* (2026). Fermat's last theorem formalised by AI agents in just 11 days. https://www.newscientist.com/article/2587839-fermats-last-theorem-formalised-by-ai-agents-in-just-11-days/

[8] *Nature* (2026). Anthropic AI 'formalizes' proof of Fermat's last theorem — a milestone for mathematics. 8 September 2026. https://www.nature.com/articles/d41586-026-02822-9

[9] *TheNextWeb* (2026). Claude formalised Fermat's Last Theorem in 11 days. https://thenextweb.com/news/anthropic-claude-fermat-last-theorem-lean-buzzard

[10] Buzzard, K. (2026). Human mathematicians are being outcounterexampled. *The Xena Project*, 20 July 2026. https://xenaproject.wordpress.com/2026/07/20/human-mathematicians-are-being-outcounterexampled/

[11] Simons Foundation (2026). From trust to verification: Lean's impact on mathematics. 23 June 2026. https://www.simonsfoundation.org/2026/06/23/from-trust-to-verification-leans-impact-on-mathematics/

[12] Massot, P. (2019). Lean in LaTeX. *The Xena Project*, 11 February 2019. https://xenaproject.wordpress.com/2019/02/11/lean-in-latex/

[13] Massot, P. (2024). Teaching mathematics using Lean and controlled natural language. In *15th International Conference on Interactive Theorem Proving (ITP 2024)*, LIPIcs vol. 309, 27:1–27:19. https://doi.org/10.4230/LIPIcs.ITP.2024.27

[14] Wegerif, R. (2026). AI: researching the future of education. *Expanding Dialogic Space*, 17 August 2026. https://rupertwegerif.substack.com/p/ai-researching-the-future-of-education

[15] de Moura, L., & Ullrich, S. (2021). The Lean 4 theorem prover and programming language. In *CADE 28*, LNCS 12699, 625–635.

[16] Coquand, T., & Huet, G. (1988). The Calculus of Constructions. *Information and Computation*, 76(2–3), 95–120.

[17] Klein, G., et al. (2009). seL4: Formal verification of an OS kernel. In *SOSP '09*, 207–220.