NATP: Native Automated Theorem Proving in Rust

William Kirkman-Moller

https://github.com/NATPProject/natp

Abstract

NATP brings automated proofs into ordinary Rust development through a typed API for mathematical sequence properties. Programmers state claims in Rust and check them alongside their tests. For identities involving concatenation and reversal of integer sequences, NATP automatically produces a proof or a concrete counterexample. The procedure uses signed-word normalisation, with soundness, completeness and a tight counterexample bound of two elements per input sequence formalised in Lean. A separate Lean checker validates exported evidence. The API distinguishes these proofs from bounded verification and SMT-assisted results. We measure decision, evidence construction and checking separately. Compact evidence reduces construction costs for valid identities by omitting the general backend's diagnostic traces.

1 Introduction

NATP lets Rust developers state mathematical properties and request automated proofs within their tests. Its typed API builds propositions over sequences and passes them to a prover through `prove()`. Developers write the property; the prover constructs the evidence. A test can require a checked proof or inspect concrete inputs that refute the claim. Bounded and solver-assisted answers have distinct result types.

An event log stores integer identifiers in older and newer batches. Reading newest first can reverse their concatenation or reverse each batch and visit the newer one first. Listing 1 proves these traversals equivalent for all finite input lengths.

```
use natp::{claim, theorem, Seq};

#[test]
fn newest_first_batches() -> Result<(), natp::Error> {
    theorem()
        .forall::<Seq<i32>>()
        .forall::<Seq<i32>>()
        .claim(claim!(|older, newer| {
            reverse(append(older, newer))
                == append(reverse(newer), reverse(older))
        }))
        .prove()?
        .assert_proven();
    Ok(())
}
```

Listing 1: Checking a newest-first traversal of two event batches in `cargo test`.

The `claim!` macro constructs the proposition, and `assert_proven()` requires an unbounded native proof. NATP proves this identity and accepts its generated certificate.

A plausible mistake keeps the original batch order on the right: `append(reverse(older), reverse(newer))`. For this variant, NATP returns `older = [0, 1]` and `newer = [2, 3]`. Joining and reversing gives [3, 2, 1, 0], while the faulty expression gives [1, 0, 3, 2]. The counterexample is replayed by the evaluator before it is reported.

QuickCheck expresses properties as host-language functions and tests them on generated inputs [1]. Creusot [2] and Verus [3] connect specifications and proofs to Rust programs. NATP embeds a language of mathematical sequence properties in Rust tests and provides automated proof procedures for that language.

Unconditional identities built from empty integer sequences, variables, concatenation and reversal normalise to words of signed variables: each letter names an input sequence and records whether it is reversed. This normal-form construction follows the laws of a monoid with involution, an established algebraic setting [6]. NATP uses that construction as the basis of its decision procedure.

We implement this procedure in Rust and connect it to independently checked evidence. We provide a typed Rust API, formalise soundness and completeness in Lean, and prove an optimal counterexample bound of two elements per input sequence. The bound gives a concrete refutation whenever an identity fails, although the identity itself quantifies over sequences of every finite length. A separate Lean checker validates exported decisions against the intended claim; its soundness and completeness are also proved. We then compare compact evidence with the general backend's certificates, measuring decision, evidence construction and checking separately. Producing this evidence can cost considerably more than deciding the equation.

The broader proposition language includes Boolean sequences, lengths, membership, element predicates and scalar equality. A native kernel checks certificates for these deductions. Bounded enumeration and SMT-assisted induction handle further claims; their result types record which guarantee was obtained.

2 Language and result semantics

2.1 Domains and expressions

A theorem consists of an ordered variable context Γ, a scope, and a proposition φ. Each variable has one of four domains: `Seq<i32>`, `Seq<bool>`, `Elem<i32>`, or `Elem<bool>`. Let I be the set of 2^{32} signed 32-bit integer values and $B = \{\text{false}, \text{true}\}$. A sequence of sort E denotes a member of E*, the set of finite lists over E. Integer elements admit equality and literals. Natural-number expressions describe lengths using constants and addition.

For a type-compatible sequence variable x, element expression a, and natural constant k, the core syntax is:

$$
\begin{aligned}
s &::= x \mid [] \mid \text{append}(s, s) \mid \text{reverse}(s) \\
a &::= \text{scalar variable} \mid \text{typed literal} \mid \text{bound element} \\
n &::= k \mid \text{length}(s) \mid n + n \\
\varphi &::= \text{true} \mid \text{false} \mid s = s \mid a = a \mid n = n \\
&\quad \mid \text{is_empty}(s) \mid \text{contains}(s, a) \mid \text{all}(s, e.\, \varphi) \\
&\quad \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \Rightarrow \varphi
\end{aligned}
$$

Inequality and proposition equivalence are derived forms. The surface syntax uses `==`, `!=`, `&&`, `||`, `!`, and `=>`. Implication has the lowest precedence and associates to the right. The predicate `all(s, e. φ)` holds when φ holds for each element occurring in s, and is true on

the empty list. Its body may mention outer variables. The current representation supports one lexical element binder, so nested `all` expressions are rejected.

The `claim!` macro translates these operators into the proposition IR. The library API can also construct propositions directly. Both interfaces supply the same representation to the reasoning procedures.

2.2 Quantification and outcomes

An environment ρ assigns each variable a value of its declared domain. In unbounded scope, Γ satisfies φ when φ evaluates to true under every such environment. In scope bounded by n, the quantification is restricted to environments whose input sequences each have length at most n. Scalar domains are unchanged. `.max_length(n)` applies to every sequence quantifier, including quantifiers added later; the bound places no restriction on lengths of intermediate results.

Table 1 lists the logical outcomes. Construction, I/O and backend errors use Rust's `Result` error channel; a successful call returns a report containing one of the outcomes below.

Outcome	Acceptance condition
`Proven`	Unbounded scope and a native certificate accepted by the kernel checker.
`Verified { max_length }`	Every required bounded case completed successfully.
`SmtValidated`	Every generated unbounded obligation was discharged by the external solver.
`Refuted { inputs }`	A concrete typed assignment evaluates the proposition to false.
`Inconclusive { reason }`	The procedure established neither validity nor a concrete refutation.

Table 1: Logical outcomes and their acceptance conditions.

Reports contain the theorem, scope, method and evidence. `assert_proven()` requires an unbounded proof; separate assertions accept bounded and solver-dependent results. A certificate received from another process must be checked against the intended theorem.

3 Proof construction and checking

3.1 Sequence and predicate normalisation

The normal form of a sequence expression is a word of signed variables. A letter $(x, 0)$ denotes the whole sequence assigned to x; $(x, 1)$ denotes its reversal. Write ε for the empty word, $\cdot$ for word concatenation, and $\mathrm{inv}(w)$ for reversal of word order with every orientation bit toggled. The normaliser N obeys:

$$\begin{aligned} N([]) = \varepsilon, \qquad N(x) &= [(x, 0)], \\ N(\mathrm{append}(s, t)) &= N(s) \cdot N(t), \\ N(\mathrm{reverse}(s)) &= \mathrm{inv}(N(s)). \end{aligned}$$

For Listing 1, both sides normalise to [(newer, 1), (older, 1)]. Append associativity and double reversal also reduce to equality of words. Each letter denotes an entire input list, of arbitrary length.

The trusted sequence rules include append identity and associativity, reverse involution and distribution over append, equality symmetry, and reverse injectivity. Equal normal words yield

true. Residual equalities are canonicalised by exchanging their sides or reversing both sides. Equality with the empty word becomes a conjunction requiring each participating sequence variable to be empty. For a residual sequence equality, the normaliser also adds the implied equality of lengths as a conjunct. Boolean reasoning can then use this consequence without changing the formula's truth value.

Lengths normalise to expressions $c + \sum_i a_i \operatorname{length}(x_i)$, with nonnegative coefficients. Common terms are cancelled across an equation. A positive sum equals zero precisely when every contributing length is zero and the constant is zero. Element equality uses reflexivity, symmetry, and equality of typed literals.

Membership in an append becomes a disjunction. Universal element predicates over an append become conjunctions. Both predicates are invariant under reversal; membership in the empty list is false, and universal predicates on it are true. An `all` predicate with constant false body reduces to emptiness. These rules remain valid when the predicate body refers to other quantified inputs, which stay fixed while elements of the selected list are considered.

Lemma 1 (normalisation)**.** For every well-typed proposition φ and environment ρ, interpreting the normalised formula under ρ gives the same truth value as interpreting φ.

Proof sketch. Induction on sequence-expression structure establishes preservation of list denotation by N. The required append and reversal laws follow by structural induction on finite lists. The length rules follow from additivity, reversal invariance, cancellation, and nonnegativity in $\mathbb{N}$. Membership and `all` distribute over concatenation by their existential and universal element semantics. Their reversal laws follow because reversal preserves element occurrences. Induction on proposition structure then establishes the result, including each predicate body under an arbitrary value of its bound element. Boolean simplifications preserve the classical interpretation. □

3.2 A complete fragment of sequence identities

Consider an unconditional equation $s = t$ whose expressions use only empty, sequence variables, append and reverse. All quantified variables are integer sequences, and there are at most eight. Write $N(s)$ for the signed-variable word defined in Section 3.1.

Theorem 1 (complete normalisation)**.** $N(s) = N(t)$ if and only if s and t evaluate to the same sequence under every environment of finite i32 sequences.

Proof. Soundness follows by structural induction: each normalisation clause preserves the interpreted list. For the converse, assign variable i the list $[2i, 2i + 1]$. A forward occurrence contributes the block $[2i, 2i + 1]$, and a reversed occurrence contributes $[2i + 1, 2i]$. The first value determines the variable and orientation, and every block has length two. Consequently, equal output lists have equal first signed letters and equal remaining encodings. Induction proves that this encoding of words is injective. Unequal normal words therefore give unequal outputs under the stated assignment. With at most eight variables, all values lie between zero and fifteen and belong to i32. □

Corollary 1 (tight length cutoff)**.** For this fragment, validity on all admissible environments with input lengths at most two is equivalent to validity on all finite input lengths. The uniform bound cannot be reduced to one.

Proof. The separating environment in Theorem 1 assigns every variable length two. Therefore every false identity has a counterexample within that bound. Conversely, every universally valid identity holds on the shorter inputs. For necessity, reverse(xs) = xs holds when xs

is empty or a singleton, while xs $= [0, 1]$ refutes it. Lean checks both directions and the lower-bound example as `two_elements_suffice` and `length_one_is_insufficient`. □

The Lean development proves normalisation soundness and injectivity of the encoding, then specialises the result to i32 sequences. Its theorem `one_assignment_decides` proves that equality under the separating assignment suffices for validity under every environment. These proofs use Lean's standard library.

The cutoff applies to unconditional equations in this grammar. The broader proposition $\text{length}(xs) = 0 \vee \text{length}(xs) = 1 \vee \text{length}(xs) = 2$, for example, holds on inputs through length two and fails at length three.

Normal words contain one letter per variable occurrence. Let $\text{occ}(s)$ count these occurrences in s. Normalisation produces exactly $\text{occ}(s)$ letters, and evaluating s under the separating environment produces exactly 2 $\text{occ}(s)$ elements. The theorems `normalise_length` and `separating_output_length` establish these equalities. The proof also defines a suffix-accumulator normaliser A and establishes $A(s, r, w) = N(s, r) \cdot w$ for every orientation r and suffix w. Its clauses construct one letter per variable occurrence without concatenating intermediate words.

The Rust implementation constructs these words with an explicit stack. It visits each syntax node once and emits each letter once, giving $O(|s|)$ time for word construction. For an expression of depth d, the stack uses $O(d)$ space and the output uses $O(\text{occ}(s))$. Input validation, counterexample evaluation and certificate generation incur separate costs. The general certificate normaliser also records a recursive trace of its steps.

After validating the theorem, the backend compares the words. If they differ, it constructs the separating assignment and evaluates the equation before returning `Refuted`. Equal words proceed to certificate generation. For example, $xs +\!\!+ ys = ys +\!\!+ xs$ is refuted by xs $= [0, 1]$ and ys $= [2, 3]$. The Rust implementation follows the proved algorithm and is tested against the separately implemented Lean checker in Section 6.

3.3 Certificates and scalar equality

After normalisation, unresolved sequence equalities, length equalities, element equalities, emptiness conditions, memberships, and universal element predicates form Boolean atoms. Case analysis assigns truth values to these atoms. For scalar equalities, true assumptions join terms into equivalence classes. A branch closes if it separates terms in the same class or equates distinct typed literals. Other relationships between residual atoms remain unexamined, so the procedure may leave valid claims unproved.

A certificate contains the exact theorem, normalised formula, normalisation trace, sorted atom table, and a deduction tree. A `True` leaf requires the entire formula to evaluate to true under its partial valuation. An `EqualityConflict` leaf requires the checker to reconstruct a scalar equality contradiction from the branch assumptions. A `Cases { atom, on_false, on_true }` node splits on the next atom in the recomputed table, and both children must pass. Unassigned atoms evaluate as unknown until the enclosing formula can be decided.

Proposition 1 (certificate soundness)**.** If a well-typed, unbounded theorem has a certificate accepted by the specified checker, its proposition holds under every environment of its declared domains.

Proof sketch. Induction on the deduction tree establishes the normalised formula under every environment consistent with its branch assumptions. A `True` leaf follows from Boolean evaluation. An `EqualityConflict` leaf has no such environment: transitivity forces terms in one class to be equal, and distinct literals denote different values. A case node covers both

values of its next atom. At the root there are no assumptions, so Lemma 1 shows that the original proposition holds under every environment. □

3.4 Trusted code and implementation limits

The certificate checker validates the theorem and requires unbounded scope. It compares the certificate's theorem with the requested theorem, recomputes the normal form and atom table, and checks the deduction tree. It also regenerates the rule trace for comparison with the supplied trace. Proof search runs in a separate module, and saved certificates can be checked in a fresh process.

The trusted code consists of IR validation in `natp-core` and normalisation, Boolean evaluation, scalar equality consistency and deduction checking in `natp-kernel`. Proposition 1 is a mathematical soundness argument for these procedures. Their Rust implementation and runtime are trusted. The mechanised results concern the equation algorithm and Lean checker described in Sections 3.2 and 3.5.

The implementation admits at most eight quantified variables, 2,048 input syntax nodes and 12 residual atoms, with certificates limited to 8 MiB. The limits apply to the proposition and its evidence; unbounded theorems quantify over sequences of every finite length.

3.5 Checking sequence identities in Lean

For the complete equation fragment, evidence consists of the equation and its two normal words, together with counterexample inputs when they differ. General Boolean deductions retain the certificate format from Section 3.3.

The Lean checker receives the expected equation separately from the evidence. It requires the embedded equation and its name to match, then recomputes both normal words. It accepts positive evidence when the words are equal and negative evidence when the supplied i32 inputs make the two sides evaluate to different lists.

Theorem 2 (independent acceptance)**.** If the typed checker accepts positive evidence for an expected equation $s = t$, that equation holds under every environment of finite i32 sequences. If it accepts negative evidence, the supplied inputs are an admissible counterexample to that equation. Accepted evidence is bound to the expected equation. Every equation in the admitted fragment has acceptable evidence.

Proof. If the checker accepts positive evidence, the recomputed normal forms are equal, so Theorem 1 establishes validity. For negative evidence, it verifies that the inputs are admissible and that the two expressions evaluate differently. The checker also requires the evidence to match the expected equation. To construct acceptable evidence, compute the two normal forms and choose positive evidence if they agree; otherwise use the separating environment. Theorem 1 guarantees that this construction succeeds, and the separating inputs satisfy the element bounds. Lean checks these statements as `accepted_valid`, `accepted_refutation`, `accepted_binding` and `canonical_accepted`. □

The Rust API can export this evidence from a `Proven` or `Refuted` report through `identity_evidence`, or construct it directly through `identity_candidate`. The direct route avoids diagnostic traces and replays counterexamples before export. Candidates have a separate type from checked `Proven` reports; both routes use the same Lean checker. `identity_normal_forms` exposes the decision words alone.

The `identity-check` executable reads equations and evidence from JSON. It compares the embedded theorem with the caller's expectation, converts the data to the types used by the proved checker, and rejects unsupported sorts, scopes and excessive depth. Theorem 2

covers the checking function on typed data. The JSON parser, Rust exporter, Lean compiler and runtime are trusted when running this executable.

4 Bounded verification

4.1 Equality-pattern enumeration

For a bound n and q sequence variables, the general bounded procedure visits the Cartesian product of the q length ranges 0 through n. At each length vector it collects the input positions and scalar variables. Boolean cells are enumerated over their two actual values. Integer cells are enumerated by equality patterns, with integer literals in the proposition fixed as distinguished classes.

For each integer cell, enumeration chooses a named literal, a previously introduced fresh class, or one new fresh class. A fresh class is represented by an unused concrete `i32` value. Restricted-growth ordering avoids enumerating different names for the same partition. The general procedure permits at most 64 input cells. The i32 domain therefore has enough values for every fresh class, even after excluding all literals admitted by the syntax limit.

Lemma 2 (representative coverage)**.** At a fixed admissible length vector, every concrete assignment has the same proposition truth value as one enumerated representative.

Proof sketch. Preserve the Boolean values. The integer assignment determines which cells equal named literals and a partition of its remaining cells by equality. Enumeration includes that partition. Map its fresh representatives to the corresponding concrete values and extend this finite bijection to a permutation of I fixing the literals. Applying the permutation elementwise commutes with append and reverse and preserves length, equality, membership, and emptiness. For `all`, apply the same permutation to the bound element and use induction on its predicate body. Induction on the proposition then gives the result. □

Proposition 2 (bounded acceptance)**.** If every required representative at every permitted length vector evaluates to true, the bounded theorem holds.

Proof. Choose any assignment within the bound. The procedure visited its length vector and, by Lemma 2, checked a representative with the same truth value. □

Repeated elements are essential. The implication $xs = ys \Rightarrow \text{is_empty}(xs)$ fails for $xs = ys = [a]$. Equality-pattern enumeration includes this assignment; restricting the search to pairwise distinct elements would miss it.

4.2 Specialised paths and resource limits

An unconditional sequence equality can be checked more cheaply. Its two sides are evaluated as words of symbolic input positions. Equal output words establish equality under every value assignment. A mismatch in output positions can be separated with two actual element values; a length mismatch already distinguishes the outputs. The candidate counterexample is evaluated concretely before reporting a refutation. Claims depending only on lengths, emptiness, and Boolean constants require one assignment per length vector.

General propositions use equality-pattern enumeration. With m integer cells, no named integer literals and b Boolean cells, there are $\text{Bell}(m) \cdot 2^b$ representatives at a fixed length vector, where Bell(m) counts set partitions. The default budgets are 100,000 cases and 10,000,000 traversal or evaluation steps. Exhausting either budget produces `Inconclusive`. A successful bounded result records the input-length bound.

5 SMT-assisted induction

The optional backend translates the same IR into SMT-LIB obligations for Z3 [4]. It declares distinct recursive list datatypes for Boolean and integer elements, with recursive definitions of append, reverse, length, and membership. Integer elements use 32-bit bitvectors without arithmetic; lengths use integers. The predicate `all(s, e. φ)` becomes universal quantification over e with membership in s as its antecedent.

The first sequence variable is selected for structural induction. With p denoting all remaining parameters, the generated queries negate the following obligations:

$$\begin{aligned} &\text{Base:} && \forall p.\ P([], p) \\ &\text{Step:} && \forall p, h, t.\ P(t, p) \Rightarrow P(h :: t, p) \end{aligned}$$

Unsatisfiability of both queries establishes the theorem by structural induction for each fixed p. Other sequence parameters remain arbitrary lists. A scalar-only theorem uses a direct query for the negated proposition.

Both induction obligations must return `unsat`. Unknown answers, timeouts and protocol failures produce an inconclusive result. A satisfiable step is also inconclusive, since a true proposition may require a stronger induction hypothesis. Successful runs return `SmtValidated`, relying on the translation, induction scheme and external solver.

6 Evaluation

6.1 Implementation and experimental setup

The Rust implementation separates the public API and syntax capture from the intermediate representation and reasoning procedures. A command-line interface exposes theorem checking and certificate export.

The experiments used NATP 0.1.0 on Windows 11 with Rust 1.97.1 and Z3 5.1.0. A set of 27 cases covers normalisation, predicates, sort distinctions, refutation and resource limits. We used release builds, the default search budgets, and a three-second timeout for each SMT obligation. A separate budget test allowed only one bounded case. We recorded outcomes, certificate sizes, enumerated cases and traversal or evaluation steps.

6.2 Proofs and counterexamples

Table 2 gives nine propositions proved by the Rust backend. Each certificate was serialised, reloaded and checked against its theorem. Certificates range from 880 to 10,331 bytes, including formatted IR and explanatory traces. Substituting another theorem name caused rejection.

K9 is proved by closing the branch in which $x = y$ and $y = z$ hold but $x = z$ fails. K8 uses the length equality implied by sequence equality.

The bounded backend refuted $xs = ys \Rightarrow \text{is_empty}(xs)$ at length one, the claim that some pair among three integer scalars must be equal, and the claim that a sequence cannot contain the literal -2,147,483,648. Every reported assignment was replayed through the concrete evaluator. The three-scalar claim is true over Booleans and was verified using all eight assignments. Double reversal with bound 100 was verified in 101 symbolic length cases and retained `Verified { max_length: 100 }`.

6.3 Enumeration growth and solver outcomes

For the claim $\text{all}(xs, e.e \neq x) \Rightarrow \neg\,\text{mem}(xs, x)$, a sequence of length l and one scalar contribute $l + 1$ integer cells and no named literals. Complete enumeration through bound n therefore requires $\sum_{l=0}^{n} \text{Bell}(l + 1)$ cases. The measured counts agree with the Bell-number calculation.

ID	Proposition	Result	Certificate bytes
K1	$r(r(xs)) = xs$	Proven	1,677
K2	$r(xs +\!\!+ ys) = r(ys) +\!\!+ r(xs)$	Proven	3,545
K3	$(xs +\!\!+ ys) +\!\!+ zs = xs +\!\!+ (ys +\!\!+ zs)$	Proven	4,100
K4	$\text{len}(r(xs)) = \text{len}(xs)$	Proven	880
K5	$xs = [] \Rightarrow r(xs) = []$	Proven	2,837
K6	$\text{mem}(r(xs), x) \Leftrightarrow \text{mem}(xs, x)$	Proven	5,770
K7	$\text{all}(r(xs), e.e \neq x) \Leftrightarrow \text{all}(xs, e.e \neq x)$	Proven	10,331
K8	$xs = ys \Rightarrow \text{len}(xs) = \text{len}(ys)$	Proven	4,637
K9	$(x = y \wedge y = z) \Rightarrow x = z$	Proven	4,515

Table 2: Propositions proved by the Rust backend. Here r is reverse, ++ is append, len is length, and mem is membership. Sequence variables have integer elements except in K3, which uses Boolean elements; x, y, z are integer scalars. All free variables are universally quantified.

Bound	Cases examined	Work units	Result
0	1	11	Verified
1	3	33	Verified
2	8	95	Verified
3	23	305	Verified
4	75	1,124	Verified
5	278	4,689	Verified
6	1,155	21,687	Verified
7	5,295	109,415	Verified
8	26,442	595,393	Verified
9	100,000	2,447,097	Inconclusive

Table 3: Bounded enumeration of the all/membership implication. Work units count instrumented traversal and evaluation steps. The final row stops at the default case budget.

At bound nine, complete enumeration requires 142,417 cases, exceeding the default budget of 100,000. The result is therefore inconclusive. The bound-four claim also returns inconclusive when limited to one case.

Z3 discharged both obligations for append$(xs, []) = xs$, producing `SmtValidated`. For $xs = []$, it returned unsatisfiable on the negated base and satisfiable on the step; the overall result was inconclusive. Across the 27 invocations, the outcomes were nine native proofs, eleven bounded verifications, three refutations, one SMT validation, and three inconclusive results.

6.4 Agreement with independent checks

We compared 225 ordered equalities formed from 15 Boolean sequence expressions with concrete enumeration through length two, and checked 52 formulas built from four predicate atoms. Small scalar assignments were enumerated to check that reported equality conflicts have no model. Regression tests also cover malformed propositions and certificates, incorrect scope and sorts, counterexample replay and solver failures.

For the complete fragment, we generated 51 expressions over two integer-sequence variables and considered every ordered equation between them. Of the 2,601 equations, 179 yield certificates and 2,422 yield counterexamples. The separate Lean executable accepts all these answers, both as exports from the general backend and as compact candidates. Tests with eight variables cover the implementation limit. Further tests ensure that the complete decision procedure is selected only for its supported sort, syntax and scope. The stack-based normaliser agrees with the trace-producing normaliser on 131 expressions in both orientations, giving 262 comparisons.

We also modified the exported evidence to test rejection. The checker rejects all 48 altered exports. They include incorrect theorem names or normal words, forged proofs, invalid counterexamples, and changes to the schema, scope, sort, arity or outcome.

Bounded verification and concrete evaluation share an evaluator; certificate checking and proof search share a normaliser. The Lean checker supplies a separate implementation of both operations for sequence identities. The timing corpus uses generated equations; properties drawn from Rust projects would help assess how representative these costs are.

6.5 Decision and proof costs

Theorem 1 permits a decision by comparing signed words or by evaluating both sides under the separating assignment. We time these two algorithms, compact evidence construction and the general certificate-producing backend separately. The decision algorithms return no evidence. Compact candidates include the equation and supporting evidence, while the general backend also constructs and checks a full certificate.

The scaling corpus contains 192 equations, half valid and half invalid. It varies the number of variables over one, two, four and eight, with three patterns of variables and orientations. Balanced expressions contain 8, 32, 128 or 256 variable occurrences per side; left- and right-skewed expressions contain 8 or 24. The right side is reconstructed from an independently generated signed word, with one orientation reversed to obtain each invalid case. All inputs fit the depth and node limits. The four Rust procedures agree on every equation, and Lean accepts both the backend exports and compact candidates. A further 48 modified compact candidates are rejected; these alter the theorem, words, decision label, result fields or counterexample.

We used release Rust 1.97.1 and Lean 4.35.0 on Windows 11 with an AMD Ryzen 9 9950X3D. Each case and method has a calibration call, a warm-up batch and seven measured batches. Calibration targets two milliseconds per batch, with at most 32,768 repetitions. Method order rotates across cases and samples. The Rust timings include input validation and

exclude syntax construction, serialisation, file access and process startup. Compact evidence construction copies the theorem and evaluates counterexamples, while the general backend constructs and checks certificates for valid identities. Table 4 reports the median of the twelve case medians in each balanced valid family.

Occurrences per side	Words	Separator	Compact evidence	Checked certificate
8	0.49	1.10	1.41	39.80
32	1.56	5.06	5.38	196.67
128	6.27	28.78	27.02	1,272.35
256	12.03	58.24	53.30	2,707.40

Table 4: Decision and evidence costs within NATP, in microseconds per valid identity. Rows aggregate four variable counts and three orientation patterns. Words and Separator time only decisions; Compact evidence adds construction; Checked certificate also includes Rust checking. Lean checking is timed separately.

Comparing normal words is the fastest decision method because it avoids constructing and moving concrete sequence elements. Compact candidates omit the diagnostic traces that the general backend constructs and rechecks for valid equations. At 256 occurrences per side, constructing compact evidence takes 53.30 microseconds, compared with 2,707.40 microseconds to construct and check a general certificate. For invalid equations at the same size, compact evidence construction takes 110.53 microseconds and the general refutation procedure takes 111.45 microseconds. Both evaluate the counterexample before returning it.

We measured the Lean checker both within a running process and as a fresh process. For batches containing 24 balanced equations, the resident checks take 8.06, 27.11, 130.92 and 317.03 microseconds per decision at 8, 32, 128 and 256 occurrences respectively. These seven-sample measurements include theorem matching, conversion to typed evidence, normalisation and counterexample evaluation. File access and generic JSON parsing occur before the timer.

Nine fresh-process measurements include file access, JSON parsing and output serialisation. Median batch times range from 38.80 to 67.35 milliseconds, compared with 37.78 milliseconds for an empty batch. Filesystem caches were warmed. Startup dominates smaller batches; recomputation and evidence checking account for more time as expressions grow.

7 Related work

Concatenation and reversal form a monoid with involution: reversal changes the product order and is its own inverse. Diekert, Jeż and Plandowski use recompression to describe all solutions of equations in free monoids with involution, including rational constraints [6]. NATP asks whether a constant-free identity holds under every substitution. Signed-word normalisation provides the algebraic basis for deciding this universal question. We implement and formalise this procedure, prove the optimal input-length cutoff and measure the cost of producing independently checked evidence from Rust proof requests.

QuickCheck expresses properties as host-language functions and tests them on generated inputs, with user-defined generators controlling the test distribution [1]. NATP also lets developers state properties within the host language. Its properties have a fixed mathematical interpretation over sequences. For bounded claims, it checks one representative of each equality pattern among the inputs.

Creusot uses prophecies to specify mutable borrows and traits to support abstraction [2]. Verus combines SMT solving with linear ghost types for reasoning about permissions and concurrent resources [3]. Both verify Rust programs against specifications. NATP uses Rust

as the host for mathematical sequence propositions, with specialised decision procedures and certificate checking for that language.

LCF established the design in which proof search is separated from a small trusted kernel; Gordon traces this approach from Edinburgh LCF to HOL [5]. NATP represents certificates as serialised data and checks them against the requested theorem. Its trusted Rust code is described in Section 3.4. Lean proves soundness and completeness of the separate checker for sequence identities.

8 Conclusion

NATP uses signed-word normalisation to prove or refute sequence identities from Rust tests, with evidence that can be checked separately. For identities built from empty integer sequences, variables, concatenation and reversal, the Lean development establishes a complete procedure, an optimal two-element counterexample bound and a sound and complete checker for exported decisions. The Rust API exposes the procedure through ordinary tests, alongside the tool's broader reasoning methods.

Evidence construction accounts for much of the measured cost of valid equations. The compact representation avoids the general backend's diagnostic traces, while counterexamples still require evaluation. Beyond sequence identities, the implementation supports predicate reasoning, bounded enumeration and SMT induction. Mechanising predicate normalisation and the general certificate checker would extend the formal guarantees to those deductions.

References

[1] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 268-279. https://doi.org/10.1145/351240.351266

[2] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A Foundry for the Deductive Verification of Rust Programs. In *Formal Methods and Software Engineering (ICFEM)*, LNCS 13478, 90-105. https://doi.org/10.1007/978-3-031-17244-1_6

[3] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1, Article 85, 30 pages. https://doi.org/10.1145/3586037

[4] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 4963, 337-340. https://doi.org/10.1007/978-3-540-78800-3_24

[5] Michael J. C. Gordon. 2000. From LCF to HOL: A Short History. In *Proof, Language, and Interaction*, Gordon Plotkin, Colin Stirling, and Mads Tofte (eds.). MIT Press. https://www.cl.cam.ac.uk/archive/mjcg/papers/HolHistory.pdf

[6] Volker Diekert, Artur Jeż, and Wojciech Plandowski. 2016. Finding all solutions of equations in free groups and monoids with involution. *Information and Computation* 251, 263-286. https://doi.org/10.1016/j.ic.2016.09.009