

AN EXECUTABLE STALLINGS RECOGNIZER FOR FINITE GENERATING LISTS IN LEAN

ARTHUR FREITAS RAMOS, DAVID BARROS HULAK, AND RUY J. G. B. DE QUEIROZ

ABSTRACT. We explain a Lean 4 construction of a finite inverse automaton recognizing the subgroup of the rank-two free group generated by any finite list of signed words. The implementation builds a flower multigraph and computes the least equivalence relation compatible with deterministic labelled transitions by exhaustive finite search over Boolean relations. Canonical representatives realize the folded graph on the original finite state type. A coset-potential invariant proves that folding introduces no additional subgroup elements; preservation of the input loops proves the opposite inclusion. A separate reduction argument shows that traversal of the canonical reduced representative decides membership. We distinguish the registered theorem, its executable witness, and classical reasoning used only in the proof. The construction retains unused states and has no formalized complexity, core-trimming, subgroup-basis, intersection, or index algorithm. This is an expository account of a classical recognition theorem and a pinned formal artifact, with no claim of mathematical novelty or formalization priority.

1. THE RECOGNITION THEOREM AND ITS SCOPE

A finite labelled graph can encode an infinite subgroup of a free group. The based loops supply group elements, while deterministic transitions make the reduced-word membership question a finite traversal. This is the recognition aspect of Stallings' finite-graph method [4]. Kapovich and Myasnikov give a later combinatorial and computational treatment [1]. The present article explains a deliberately limited formal implementation of that recognition result. It does not formalize the full range of applications of subgroup graphs developed in those works.

All implementation claims below refer to the repository *stallings-folding* at commit 117dde0a8415d3da1787e15ad22a6223a744432d [3]. This is the commit recorded by PALOMAR-2026-09-23-000004, version 1 [2]. The public record selects `Stallings.folded_recognizer_exists` and the closed statement definition `Stallings.completeStatement`. The construction and intermediate correctness results are in the implementation modules; the final interface is in `Solution.lean`.

1.1. Words and the generated subgroup. Write $\mathcal{A} = \{a, a^{-1}, b, b^{-1}\}$ and let $x \mapsto \bar{x}$ exchange a letter with its inverse. A word is a finite list of letters. Its evaluation in $F_2 = F(a, b)$ is denoted by $[w]$. If $w = x_1 \cdots x_k$, its inverse word is $\bar{x}_k \cdots \bar{x}_1$. In Lean these data are `Fin 2 × Bool`, lists of these pairs, and `FreeGroup (Fin 2)`; the Boolean value `true` denotes the positive orientation. The definitions `letterInv`, `wordInv`, and `wordEval` implement inversion and evaluation.

For a finite list $S = (s_0, \dots, s_{m-1})$ of words, set

$$(1) \quad H_S = \langle [s_i] \mid 0 \leq i < m \rangle \leq F_2.$$

Date: October 1, 2026.

2020 Mathematics Subject Classification. 20E05, 68Q45, 03B35.

Key words and phrases. Stallings folding, free groups, finite inverse automata, subgroup membership, Lean 4, Mathlib.

Author ORCID identifiers: Arthur Freitas Ramos, 0009-0003-3568-0325; David Barros Hulak, 0009-0002-8056-1774; Ruy J. G. B. de Queiroz, 0000-0003-1482-0977.

Copyright 2026 the authors. Licensed under Creative Commons Attribution 4.0 International (CC BY 4.0).

The source defines `generatorSubgroup S` using `Subgroup.closure` of the range of evaluation on `Fin S.length`. There is no requirement that the input words be reduced, nonempty, distinct, or a free basis. The empty list and lists containing empty words are allowed.

Theorem 1.1 (Registered finite-recognizer statement). *For every finite list S of signed words, there exist a finite type V , a basepoint $o \in V$, and a partial transition map $\delta : V \times \mathcal{A} \rightarrow V \cup \{\perp\}$ such that*

$$(2) \quad \delta(v, x) = w \implies \delta(w, \bar{x}) = v$$

and, for every $g \in F_2$,

$$(3) \quad \text{run}_\delta(o, \text{red}(g)) = o \iff g \in H_S.$$

Here $\text{red}(g)$ is the canonical reduced word for g and traversal fails if a required transition is absent.

In the registered definition, V has a `Fintype` instance, δ returns `Option V`, and traversal is written directly with `List.foldl` and `Option.bind`. The canonical word is `FreeGroup.toWord g`. A successful return is `some base`; `none` represents failure. Thus the statement uses Mathlib’s free group and list operations without importing the candidate’s automaton semantics. It specifies recognition existentially; the implementation supplies the witness through `foldWords S`.

2. INVERSE AUTOMATA AND CANONICAL-WORD TRAVERSAL

2.1. The subgroup represented by an automaton. The structure `InverseAutomaton V` consists of a basepoint, a partial transition map, and the inverse-edge law (2). Determinism is built into the type of the transition map: an input pair (v, x) has at most one target. The structure itself does not require V to be finite; this allows its semantic lemmas to be used independently of the later finite construction.

The function `InverseAutomaton.run` recurses over a word, propagating failure through `Option.bind`. The inductive predicate `InverseAutomaton.Walk` records the corresponding path. The theorem `run_eq_some_iff_walk` equates successful execution with a walk having the stated endpoint. Walks concatenate, and reversing a walk while inverting its labels gives a walk in the reverse direction.

For an inverse automaton G with basepoint o , define

$$L(G) = \{[w] \mid w \text{ labels a walk from } o \text{ to } o\}.$$

Empty walks, concatenation, and reversal prove that $L(G)$ is a subgroup. This is the carrier of `InverseAutomaton.loopSubgroup`; no additional subgroup closure is needed in this definition.

2.2. Why reduction is part of the specification. A partial automaton need not read every word representing an accepted group element. If a required edge is missing, inserting an inverse pair can turn a successful traversal into a failure. The usable direction is that *deleting* an inverse pair from a successful traversal preserves its endpoint.

Lemma 2.1 (Successful traversal survives reduction). *If $\text{run}_G(v, w) = z$ and w' is obtained from w by free reduction, then $\text{run}_G(v, w') = z$.*

Proof. For one cancellation write $w = p x \bar{x} q$. Successful traversal of p ends at some vertex u , and the next letter x takes u to a vertex t . The inverse-edge law forces the following $\bar{x}$ transition to return to u . Removing these two steps leaves traversal of q unchanged. Induction over a sequence of cancellations proves the claim. $\square$

The source develops this argument in `run_success_cancel`, `run_success_of_red_step`, `run_success_of_red`, and `run_success_of_reduction`. The concatenation identity `run_append` isolates the cancelled pair inside the traversal.

Proposition 2.2 (Canonical-word recognition). *For every inverse automaton G and $g \in F_2$, traversal of $\text{red}(g)$ returns to the basepoint if and only if $g \in L(G)$.*

Proof. A successful canonical-word traversal is itself a based loop, and its word evaluates to g . Conversely, choose a based-loop word w evaluating to g . Lemma 2.1 preserves its successful endpoint while reducing w . Mathlib’s reduced-word soundness identifies this reduction with the canonical reduced representative of g . The result follows. $\square$

This is `InverseAutomaton.accepts_iff_mem_loopSubgroup`. The Boolean function `InverseAutomaton.membershipTest`, available when vertices have decidable equality, tests whether the canonical traversal returns some `base`. Its correctness theorem is `InverseAutomaton.membershipTest_eq_true_iff`. The proof uses Mathlib’s `FreeGroup.mk_toWord` and `FreeGroup.reduce.sound` [5]. It does not assert equality of traversal outcomes for all freely equivalent, unreduced words.

3. A FINITE LEAST-CONGRUENCE FOLDING CONSTRUCTION

3.1. The input multigraph and admissible relations. Before folding, the labelled graph can have several targets for a fixed source and label. The structure `InverseMultigraph V` represents these targets by finite sets $E(v, x)$. Its inverse law is

$$u \in E(v, x) \iff v \in E(u, \bar{x}).$$

Throughout the folding module, V has finite enumeration, decidable equality, and a linear order. The order is used to choose representatives.

Definition 3.1 (Fold congruence). A relation R on V is a fold congruence if it is an equivalence relation and satisfies

$$(4) \quad v R w, \quad u \in E(v, x), \quad t \in E(w, x) \implies u R t.$$

The condition includes $v = w$, so all targets of the same label from one vertex must be identified. It also enforces consistency after source vertices have been identified. The source predicate `IsFoldCongruence` uses a Boolean table $r : V \rightarrow V \rightarrow \text{Bool}$ and expresses reflexivity, symmetry, transitivity, and (4) with finite quantifiers. Its explicit decidability instance follows from the finite types involved.

Define the relation R_* by

$$(5) \quad v R_* w \iff \text{every Boolean fold congruence relates } v \text{ and } w.$$

This is `foldRel`. The universal Boolean relation is a fold congruence, so the family is nonempty. The lemmas `foldRel.refl`, `foldRel.symm`, `foldRel.trans`, and `foldRel.edge_congr` prove that its intersection is itself a fold congruence. By its definition, R_* is contained in every fold congruence. These facts justify describing it as the least deterministic fold congruence.

3.2. Representatives and partial transitions. Let $\text{rep}(v)$ be the least vertex in the R_* class of v . Reflexivity ensures that this class is nonempty. The function `foldRep` takes the minimum of a filtered `Finset.univ`. Its main properties are

$$\text{rep}(v) R_* v, \quad v R_* w \implies \text{rep}(v) = \text{rep}(w), \quad \text{rep}(\text{rep}(v)) = \text{rep}(v).$$

They are proved by `foldRep_rel`, `foldRep_eq_of_rel`, and `foldRep_idem`.

For a class and a label, collect the original targets

$$T(v, x) = \{t \in V \mid \exists u, u R_* v \text{ and } t \in E(u, x)\}.$$

This finite set is `foldTargets G v x`. Any two of its elements are R_* equivalent, by (4); this is `foldTargets_pair`. Hence every target in $T(v, x)$ has the same representative. The transition chooses the least target and then its representative, or fails if the target set is empty.

The actual state type remains V . The function `foldedNext` permits outgoing transitions only when $v = \text{rep}(v)$; other vertices have no outgoing transitions. The basepoint becomes $\text{rep}(o)$, and every successful target is canonical. Thus traversals from the basepoint stay in the canonical part of V , although the returned type still includes noncanonical vertices. The definition `foldAutomaton` proves the inverse law for these transitions using the original inverse edges and equality of target representatives.

3.3. Executability and termination. The computation uses decidable predicates on finite types, finite filtering, and finite minima. In particular, deciding (5) enumerates the finite type of Boolean relations and checks the fold congruence conditions. This is exhaustive reference computation; there is no implemented union-find fold sequence. Traversal is structural recursion on the input word. These definitions do not depend on a search for an unbounded sequence of folds or on an external termination oracle.

The executable functions should be distinguished from a classical predicate used only in their correctness proof. The private definition `cosetBool` in `Folding.lean` is noncomputable: it converts membership in a generally undecidable subgroup coset relation into a Boolean relation under classical reasoning. It supplies a candidate relation to the proof of (5); the folding code never computes `cosetBool` or queries subgroup membership. No complexity bound or performance benchmark is proved or reported here.

4. THE COSET INVARIANT AND PRESERVATION OF SUBGROUP LABELS

Identifying vertices can create new combinatorial walks, so preservation of old paths alone proves only one inclusion of subgroups. The reverse inclusion uses a potential valued in the free group. This avoids an assertion that every folded walk lifts to a walk with the same word in the unfolded graph.

4.1. The orientation of the coset relation. For a subgroup $H \leq F_2$, write

$$(6) \quad g \sim_H h \iff gh^{-1} \in H.$$

Equivalently $Hg = Hh$. Formula (6), rather than terminology about left or right cosets, fixes the convention. The relation is an equivalence relation and is invariant under common multiplication on the right:

$$(7) \quad g \sim_H h \implies gx \sim_H hx.$$

Indeed $(gx)(hx)^{-1} = gh^{-1}$. The source names this relation `CosetEquivalent` and proves the corresponding reflexivity, symmetry, transitivity, and `right_mul` lemmas.

Definition 4.1 (Compatible potential). A potential $p : V \rightarrow F_2$ is compatible with a labelled multigraph modulo H if each edge $v \xrightarrow{x} w$ satisfies

$$(8) \quad p(w) \sim_H p(v)[x].$$

This is `HasCosetPotential`. Induction on walks gives

$$v \xrightarrow{w} z \implies p(z) \sim_H p(v)[w].$$

The same expression holds for successful folded traversal, as shown below.

Lemma 4.2 (A fold respects compatible potentials). *If p is compatible modulo H , then $v R_* w$ implies $p(v) \sim_H p(w)$.*

Proof. Consider the relation $v R_p w$ defined by $p(v) \sim_H p(w)$. It is an equivalence relation. If $v R_p w$ and there are x edges from v, w to u, t , compatibility and (7) give

$$p(u) \sim_H p(v)[x] \sim_H p(w)[x] \sim_H p(t).$$

Thus R_p is a fold congruence. Represent it as a Boolean relation using classical decidability. Definition (5) places R_* inside this relation, proving the assertion. $\square$

The lemma is `foldRel_cosetPotential`. Using it to move between a vertex and its representative, `foldedNext_cosetPotential` proves that each folded transition satisfies (8). Induction on words then gives `run_cosetPotential`:

$$(9) \quad \text{run}_{\text{fold}(G)}(v, w) = t \implies p(t) \sim_H p(v)[w].$$

4.2. Equality of the represented subgroups. For an inverse multigraph G , the source defines its based-loop subgroup as the *subgroup closure* of evaluated labels of walks based at o . Denote this subgroup by H_G . This differs definitionally from the automaton's direct loop carrier, although both have the intended group semantics.

Proposition 4.3 (Folding preserves the loop subgroup). *Suppose that G has a potential compatible modulo H_G and $p(o) = 1$. Then $L(\text{fold}(G)) = H_G$.*

Proof. Each original edge induces a folded transition between representatives, by `foldedNext_of_edge`. Induction gives `foldWalk`, which maps an original walk's endpoints to their representatives without changing its word. Every original based loop therefore belongs to the folded loop subgroup. Closure proves $H_G \subseteq L(\text{fold}(G))$.

For the opposite inclusion, let a folded loop at $\text{rep}(o)$ have label w . Lemma 4.2 and $p(o) = 1$ imply $p(\text{rep}(o)) \in H_G$. Equation (9) gives

$$p(\text{rep}(o)) \sim_{H_G} p(\text{rep}(o))[w].$$

If $g \in H_G$ and $g \sim_{H_G} h$, subgroup closure under products and inverses implies $h \in H_G$. Applying this fact shows $p(\text{rep}(o))[w] \in H_G$. Multiplication by $p(\text{rep}(o))^{-1}$ gives $[w] \in H_G$, as required. $\square$

The proposition is `foldAutomaton_loopSubgroup_eq`. It is stated with a compatible potential as a hypothesis. The end-to-end construction supplies that potential explicitly for its flower graph; it does not assume that an arbitrary finite inverse multigraph comes equipped with one.

5. THE FLOWER CONSTRUCTION AND END-TO-END CORRECTNESS

5.1. A finite state type with unused positions. Let m be the number of input words and M their maximum length, with $M = 0$ when S is empty. The state type is

$$(10) \quad V_S = \{o\} \sqcup (\{0, \dots, m-1\} \times \{0, \dots, M\}),$$

implemented by `FlowerVertex S`, whose definition is the option of that finite product. The basepoint is `none`. Positions 0 and $|s_i|$ on the i th word both map to the basepoint; positions strictly between them use their own state (i, k) . Each letter of s_i gives a forward edge between successive positions and an inverse edge in the reverse direction.

The full product in (10) includes positions unused by these loops. In particular the pair states with $k = 0$ and $k \geq |s_i|$ have no incident flower edges. The construction keeps these positions for a simple uniform finite type. It is not a core-trimming procedure. The function `flowerPos` treats the endpoints, `flowerEdge` specifies the two orientations, and `flowerGraph` filters the finite vertex set to obtain each target set. A lexicographic order with `none` first provides the linear order required for representatives.

5.2. The prefix potential. The potential at an internal position (i, k) is the evaluated prefix $[s_i[0 : k]]$. The basepoint and unused positions receive value 1. For an internal forward edge, evaluation of a prefix extended by one letter is exactly its earlier value times that letter. For the final edge back to the basepoint, the full-word evaluation lies in H_S , so the equality is replaced by coset equivalence modulo H_S . Inverse edges inherit the same compatibility from inversion and common right multiplication.

These steps prove `flower_hasCosetPotential` for `flowerPotential`. The theorem `flowerWalk` establishes a based walk spelling each input word. It is proved by induction on the remaining suffix as the position advances, so it covers empty words as well.

Proposition 5.1 (The flower represents the input subgroup). *The subgroup closure of the flower’s based-loop labels is H_S .*

Proof. Every input word labels a based loop, giving the inclusion of its generated subgroup in the flower’s loop subgroup. For the converse, the prefix potential begins and ends at 1 on a based loop. Compatibility along its walk implies $1 \sim_{H_S} [w]$, or $[w]^{-1} \in H_S$. Taking inverses proves that its label is in H_S . Subgroup closure yields the desired inclusion. $\square$

This is `flowerGraph_loopSubgroup_eq_generatorSubgroup`. Combining it with Proposition 4.3 gives

$$(11) \quad L(\text{foldWords } S) = H_S,$$

proved as `foldWords_loopSubgroup_eq_generatorSubgroup`. Proposition 2.2 then establishes the executable theorem

$$(12) \quad \text{foldedMembershipTest } S \ g = \text{true} \iff g \in H_S,$$

namely `foldedMembershipTest_eq_true_iff`.

Finally, `Solution.lean` witnesses Theorem 1.1 with V_S , its finite enumeration, the basepoint of `foldWords S`, and its transition map. The private lemma `foldl_run_eq` connects the registered statement’s list fold with the implementation’s recursive traversal. The inverse law and (11) then discharge the two specification clauses.

5.3. A small explanatory example. Consider $S = (aa, ab)$, with both letters positive. The active flower vertices are the basepoint o and two intermediate vertices u, v . Both generator loops begin with an a edge from o , so every fold congruence identifies u with v . There is a fold congruence that keeps o distinct from them and leaves each unused position separate; hence no further active vertex identification is forced. Write q for the representative of u, v . The basepoint component of the output has transitions

	a	a^{-1}	b	b^{-1}
o	q	q	$\perp$	q
q	o	o	o	$\perp$

and basepoint o . The full returned state type still has the unused positions from (10).

Reading aa or ab returns to o , while reading a ends at q and reading b fails. The word bb^{-1} also fails as a raw traversal, even though it evaluates to the identity. Its canonical reduced word is empty and is accepted. This illustrates why (3) specifies canonical-word traversal. The example is a direct mathematical reading of the definitions, rather than a reported Lean evaluation or performance measurement.

6. THE REGISTERED ARTIFACT AND VERIFICATION BOUNDARIES

6.1. The statement and proof modules. The Mathlib-only `Challenge.lean` contains the selected closed specification and a deliberate theorem placeholder. The actual proof is in `Solution.lean`, which imports the implementation. The project configures separate `Challenge` and `Solution` modules for comparison. The placeholder in the challenge is not a proof of the result and is not used by the solution.

The four implementation files separate inverse-automaton semantics, finite folding, arbitrary word flowers, and a supplementary one-vertex rose for subsets of the original basis. The rose module proves a basic special case; it does not implement extraction of a free basis for an arbitrary input subgroup. The main proof follows the arbitrary-word flower path described above.

The pinned script `scripts/StatementAudit.lean` inspects the compiled statement’s constant dependencies and prints the theorem’s axioms. Its purpose is to ensure that the selected closed mathematical statement does not reach unselected candidate-defined data. The present source audit checked that the `Challenge` and `Solution` definitions agree and that

the solution supplies the advertised witness. A text scan found the deliberate Challenge placeholder and the proof-only noncomputable `cosetBool`; no `sorry`, `admit`, custom axiom declaration, or `unsafe` declaration was found in the solution or implementation. Such a scan is a limited source check, not a kernel replay.

6.2. Historical mechanical verification. The record pins Lean 4.33.0 and Mathlib commit `db584cd6d46c92f209a44c0f1c829460d327499d`. It reports verification on September 23, 2026, through the recorded Palomar workflow run [2]. The comparator configuration selects the theorem and closed statement definition, enables NanoDa, and permits `propext`, `Quot.sound`, and `Classical.choice`. The record reports a neutral review outcome with no listed warnings and preservation of the exact source commit. These are historical registry facts; they are not a claim of human peer review.

For this article, the immutable source was checked out, the public registry record was re-read, the definitions and proof dependencies were inspected, and the manuscript was compiled and visually checked. No fresh Lean build, compiled statement audit, comparator run, or NanoDa replay was performed for this manuscript. The historical verification record supports the formal checking claim. The present inspection supports the correspondence between the prose and the pinned source, within that explicit boundary.

6.3. Attribution and assistance disclosure. Stallings’ finite-graph method is the mathematical source; the implementation adapts it to an exhaustive least-congruence computation. Mathlib supplies the free-group quotient and reduced-word API. The repository’s provenance states that its implementation was written for the project and does not vendor a prior formalization. This article makes no priority claim about other formalizations and reports no new group-theoretic theorem.

Historical repository metadata identifies Arthur Freitas Ramos as its author and responsible maintainer, and records AI-assisted proof development with *GPT-6 Codex*. That historical statement is separate from the present manuscript’s preparation. This manuscript was drafted primarily with GPT-6.1 assistance, including source cross-checking and typesetting. The submitting author, Arthur Freitas Ramos, reports understanding some parts of the work; this disclosure does not assert full understanding by him or any level of understanding by the other named manuscript authors. Automated source and mathematical review are not independent human peer review. The Lean source is licensed Apache-2.0; this manuscript and its TeX source are licensed CC BY 4.0.

7. CONCLUSION

The artifact gives a finite, executable witness to subgroup recognition for every finite list of words in F_2 . Its correctness separates into three reusable arguments: canonical reduction preserves successful inverse-automaton traversal, least-congruence folding preserves loop labels through a coset potential, and the flower’s prefix potential identifies its loop subgroup with the input-generated subgroup. The final Mathlib-only specification packages exactly these recognition semantics.

The implementation does not trim the graph, compute subgroup bases or rank, compute intersections or index, or certify a complexity bound. Replacing the exhaustive congruence search with an optimized algorithm would require its own implementation and a proof relating its output to the existing correctness interface. The present result supplies that interface without claiming those additional algorithms.

REFERENCES

1. Ilya Kapovich and Alexei Myasnikov, *Stallings foldings and subgroups of free groups*, Journal of Algebra **248** (2002), no. 2, 608–668, DOI: <https://doi.org/10.1006/jabr.2001.9033>. Preprint titled *Stallings foldings and the subgroup structure of free groups*, arXiv:math/0202285.
2. Palomar Registry, *Verified finite automata for finitely generated subgroups of F_2* , <https://palomar-registry.org/entry?id=PALOMAR-2026-09-23-000004&version=1>, 2026, PALOMAR-2026-09-23-000004, version 1; registered September 23, 2026; accessed October 1, 2026.
3. Arthur Freitas Ramos, *Verified finite automata for finitely generated subgroups of F_2* , <https://github.com/Arthur742Ramos/stallings-folding/tree/117dde0a8415d3da1787e15ad22a6223a744432d>, 2026, Immutable source commit 117dde0a8415d3da1787e15ad22a6223a744432d; accessed October 1, 2026.
4. John R. Stallings, *Topology of finite graphs*, Inventiones Mathematicae **71** (1983), no. 3, 551–565, DOI: <https://doi.org/10.1007/BF02095993>.
5. The Mathlib Community, *Mathlib free-group reduction library*, <https://github.com/leanprover-community/mathlib4/blob/db584cd6d46c92f209a44c0f1c829460d327499d/Mathlib/GroupTheory/FreeGroup/Reduce.lean>, 2026, Pinned commit db584cd6d46c92f209a44c0f1c829460d327499d; accessed October 1, 2026.