

Compiling Learned Lexical Transformations: Semantic Canonicalization and Accepting-Leaf Generalization in Patch-Command Tries

Leo Galambos

Abstract—Lexicon-trained stemmers can derive lexical transformations offline while requiring only a compact immutable decision structure online. The compilation step is therefore part of the algorithm, not merely a serialization detail. This paper formalizes the reduction semantics of Radixor’s patch-command trie compiler from a frozen JDK implementation snapshot. We show that the compiler combines two mathematically distinct transformations. First, mode-specific subtree canonicalization computes a recursive signature from a local output descriptor, accepting-state information, and sorted edge-labelled child signatures. Equal signatures are hash-consed into one canonical node. We prove that signature equality is a substitutive congruence for the observable declared by each public reduction mode and that the ranked and qualified-dominant descriptors remain stable when canonical nodes aggregate local evidence counts. For productive nodes in a contraction-free ranked or unordered run, we further prove the converse: continuation-observable equivalence implies signature equality. The canonicalization-only quotient is therefore state-minimal among quotientings of the fixed node-output trie representation. With accepting-leaf generalization enabled, signature equality remains sound but is not claimed to be the coarsest behavioral equivalence; dominant mode is intentionally conservative as well. We prove that quotienting remains acyclic and that the finite compiler terminates; we also characterize a nontrivial implementation cost: recursively recomputed signature hashes can require quadratic work on a degenerate chain even though the semantic quotient is finite. Second, optional uniform-subtree contraction replaces a uniformly single-valued region by an accepting leaf. It preserves the existing singleton command-selection semantics on the region’s defined domain while extending the selected patch command to every remaining continuation. For stemming, this generalizes the *choice of transformation program*, not a constant stem string: each unseen form receives $\text{Exec}(c, w)$ under the same selected command c . In backward suffix tries, the operation becomes a suffix-conditioned command-selection rule over previously unseen prefixes. Existing unit and property tests corroborate the frozen implementation; linguistic correctness of newly defined conflations remains an empirical question. The result separates quotient canonicalization, evidence aggregation, state minimality relative to an API contract, implementation complexity, and rule-like command generalization in compiled lexicon-trained stemming.

Index Terms—stemming, trie, finite-state methods, automata minimization, canonicalization, lexical morphology, compilation, patch commands, Radixor

I. INTRODUCTION

LEXICON-TRAINED stemming changes where linguistic knowledge lives. Instead of executing a hand-written language program at query time, the system can derive word-to-stem transformations from lexical evidence offline and compile

those transformations into a compact runtime structure. The historical Egothor work introduced this direction for Java-based information retrieval by learning edit commands from lexical examples rather than requiring a complete grammar specification [1], [2]. Radixor continues that lineage with immutable compiled tries, explicit ambiguity handling, deterministic binary persistence, and several reduction contracts [19].

The resulting systems question is deceptively simple: when two regions of a learned trie can be represented by one physical subgraph, what exactly must be equal for that replacement to be correct? In a Boolean dictionary automaton, state equivalence is usually defined by accepted-language equivalence. In a patch-command trie, nodes can carry several ranked values, local evidence counts, preferred outputs, and an accepting state that may terminate traversal before the input is exhausted. A correct reduction therefore depends on which runtime observable an application promises to preserve.

A second issue is even more important. Radixor’s patch-command compiler contains an optional uniform-subtree contraction. When every currently reachable decision in a local region has one value, the region can be replaced by an accepting leaf. Because lookup returns immediately from such a leaf without consuming the rest of the key, the transformation does more than deduplicate representation: it generalizes a uniform observed decision to unseen continuations. For example, in a forward trie containing only $aa \mapsto x$ and $ab \mapsto x$, exact lookup returns no value for az ; after contraction of the a region to an accepting leaf, $az \mapsto x$ [19]. This is intentional behavior and cannot be described honestly as exact-language preservation over all strings.

The distinction motivates the central thesis of this paper:

Radixor compilation combines semantic subtree canonicalization with accepting-leaf command generalization. The first constructs a quotient DAG under a mode-specific output congruence. The second is a monotone extension of the domain on which one transformation program is selected over a uniform decision region.

The paper is theorem-driven rather than benchmark-driven. Its source of truth is the frozen Radixor implementation at commit 0c3b13f, especially `FrequencyTrie`, `ReductionSignature`, `LocalValueSummary`, `ReductionContext`, and the reduction-mode descriptors [19]. Existing tests are used as implementation corroboration, not as substitutes for proof.

The research questions are:

- RQ1: What observable semantics are preserved by ranked, unordered, and dominant subtree canonicalization?
- RQ2: Does aggregation of local evidence counts after canonical merging preserve the descriptor on which the equivalence proof relies?
- RQ3: What semantics does uniform-subtree contraction preserve, and what new behavior does it introduce?
- RQ4: When is the compiler’s signature relation complete, rather than merely sound, for continuation-observable equivalence?
- RQ5: What termination, acyclicity, and complexity properties follow from the concrete bottom-up implementation?
- RQ6: How does the formal model correspond to the actual immutable Java representation, patch-command execution stage, persistence format, and test suite?

The contributions are:

- 1) a recursive, implementation-matched definition of mode-specific subtree signatures and continuation-observable equivalence;
- 2) aggregation-stability lemmas for ranked and qualified-dominant local semantics;
- 3) a global congruence theorem for signature-based canonicalization;
- 4) a completeness theorem for contraction-free ranked and unordered runs, yielding relative state minimality of the canonicalization-only quotient and an explicit boundary once accepting leaves are admitted;
- 5) proofs of quotient acyclicity, finite termination, and decidability, plus an implementation-level complexity bound that exposes repeated recursive signature hashing;
- 6) a value-semantic domain-extension theorem for accepting-leaf contraction and a separate command-selection/output interpretation for stemming;
- 7) a precise partial order of preserved and non-preserved value APIs, including the explicit non-preservation of original count history; and
- 8) an implementation correspondence and verification audit that distinguishes mathematical proof from software regression evidence.

The novelty claim is deliberately narrow. Continuation equivalence and finite-automaton minimization, trie sharing, acyclic lexicon automata and transducers, sequential/subsequential transducer minimization, and finite-state morphology are established topics [4], [8]–[16]. The contribution here is not a new general minimization algorithm or a new general theory of output-machine minimization. It is a formalization of a lexicon-trained stemming compiler whose equivalence relation is selected by an API output contract, whose nodes carry ranked count-derived evidence, whose ranked/unordered signatures can be shown complete in the contraction-free exact-trie phase within a fixed node-output representation, and whose separate accepting-leaf transformation intentionally changes the domain of command selection.

II. BACKGROUND AND NOVELTY BOUNDARY

A. From Lexical Examples to Patch Commands

Classical stemmers illustrate two relevant ways to obtain generalization. Lovins and Porter encode hand-designed affix transformations that apply beyond a memorized lexicon [3], [5]. Krovetz later framed morphological normalization as inference supported by lexical evidence rather than unrestricted suffix deletion [6], [7]. Egothor instead learns transformations from lexical pairs and associates observed word patterns with patch commands [1]. Radixor retains that separation between offline lexical induction and online command execution. A runtime trie value is a serialized patch command, later specialized to an immutable executable command, that deterministically transforms the input token when selected [19].

Three terms must remain distinct. A *canonical target* is the resource-defined target used to induce a patch; it is not assumed to be a linguistic root or context-sensitive lemma. A *patch command* is the executable edit program selected by the trie. A *stem* is the runtime output after that program is applied to a token. This paper starts after command induction. It does not decide whether a lexical assignment is linguistically correct, whether a target policy is desirable, or which command should be preferred for a language. Its object is the compiler that receives key–value evidence and produces an immutable trie/DAG with a declared lookup contract.

B. Classical Automata Minimization

Deterministic automata minimization identifies states that cannot be distinguished by any future input; continuation-based equivalence is the classical organizing idea behind Nerode-style state identification [4]. Hopcroft’s algorithm minimizes finite automata efficiently under accepted-language equivalence [8]. For acyclic automata, Revuz gives a linear-time minimization algorithm particularly relevant to lexicons [9]. Daciuk *et al.* address both minimal acyclic automata and transducers in dictionary construction [10], and later give the well-known incremental construction of minimal acyclic finite-state automata [13]. More generally, minimization of sequential and subsequential transducers establishes canonical or minimal representations for deterministic machines with outputs [11], [12]. In NLP, finite-state transducers provide a broader framework for language processing [14]; two-level morphology and later finite-state morphology systems model lexical–surface relations for morphological analysis and generation [15], [16].

Radixor’s canonicalization therefore sits inside an established output-automata tradition, but the exact equivalence object and preservation contract differ from classical minimal sequential-transducer constructions. A node is not simply final or non-final: it may contain a deterministic ordering of alternative values, a count distribution that determines a winner, and an accepting flag that changes traversal. Furthermore, the dominant reduction mode intentionally uses a sufficient, sometimes stricter-than-necessary descriptor rather than claiming globally minimal equivalence under `get()`. The compiler therefore should not be presented as a generic minimal DFA/FST construction. Its

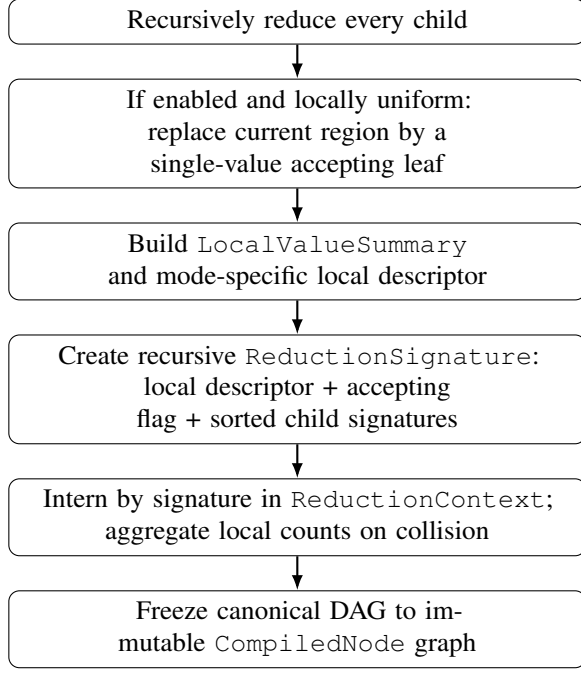


Fig. 1. Implementation-matched compilation pipeline. Contraction is attempted after descendants have already been reduced, but before the current node’s signature is interned.

narrower contribution is an implementation-matched, mode-selectable output congruence whose local evidence counts are aggregated on collisions, plus a separate accepting-prefix transformation that deliberately extends the input domain.

C. Two Distinct Compiler Transformations

The frozen compiler’s recursive procedure is summarized in Fig. 1. Each mutable child is reduced first. The current node may then be contracted if uniform-subtree contraction is enabled. Finally, a mode-specific signature is computed and interned in a canonical-node map. Thus canonicalization and contraction are mathematically separable, but operationally interleaved bottom-up.

III. TRIE AND LOOKUP SEMANTICS

A. Traversal Space

Let Σ be the runtime alphabet of Java UTF-16 `char` values after any external key normalization; concretely, $|\Sigma| = 2^{16}$. Let a logical key be $w \in \Sigma^*$. Radixor supports forward and backward traversal. Define

$$\tau_d(w) = \begin{cases} w, & d = \text{FORWARD}, \\ \text{rev}(w), & d = \text{BACKWARD}. \end{cases} \quad (1)$$

The reversal in Eq. (1) is over UTF-16 code units, matching Java `char`-wise traversal; it is not Unicode scalar-value reversal. All proofs operate in this traversal space, so the same statements apply to both directions. For production suffix models, $d = \text{BACKWARD}$; a traversal prefix therefore corresponds to a logical-word suffix.

Before reduction, the mutable trie is a rooted tree $T = (V, E, r)$ with at most one outgoing edge per character. Each node v stores a finite map of positive local counts

$$c_v : X \rightarrow \mathbb{N}_{>0}, \quad (2)$$

where X is the value domain. For stemming, X is a patch-command domain.

B. Deterministic Local Ranking

The implementation converts c_v into an ordered local sequence. Values are sorted by descending count; ties are broken by shorter textual representation, then lexicographically lower textual representation, then stable first-seen insertion order [19]. Let

$$\mathcal{R}(v) = \langle x_1, \dots, x_k \rangle \quad (3)$$

denote that sequence, and let

$$\mathcal{S}(v) = \{x_1, \dots, x_k\}. \quad (4)$$

If $k > 0$, the preferred value is $\mathcal{P}(v) = x_1$; otherwise it is undefined.

For dominant reduction, the implementation also uses thresholds $p \in [1, 100]$ and $q \geq 1$. Let $C_v = \sum_x c_v(x)$ and let s_v be the second-highest local count, or zero if no competitor exists. The preferred value x_1 is *qualified dominant* when

$$100c_v(x_1) \geq pC_v, \quad (5)$$

$$c_v(x_1) \geq qs_v, \quad (6)$$

with the ratio condition vacuously true when $s_v = 0$. The production defaults are 75 percent and a 3:1 winner-over-second ratio, although the proof treats p and q symbolically [19].

C. Compiled Lookup and Accepting Leaves

A compiled node stores sorted child edges, ordered local values and counts, and a Boolean $A(v)$ corresponding to `acceptsRemainingInput`. Exact traversal follows $\tau_d(w)$ character by character. Before consuming each remaining character, lookup checks $A(v)$. If $A(v) = 1$, lookup returns v immediately and ignores the rest of the key [19].

For a compiled node v , define $F_v(s)$ as the resolved node when lookup starts at v with remaining traversal suffix s . If no path is defined, $F_v(s) = \perp$. Define the ranked and set-valued lookups by

$$R_v(s) = \begin{cases} \mathcal{R}(F_v(s)), & F_v(s) \neq \perp, \\ \langle \rangle, & F_v(s) = \perp, \end{cases} \quad (7)$$

$$S_v(s) = \{x : x \in R_v(s)\}. \quad (8)$$

Because a resolved internal node may have no local values, define preferred lookup from the ranked sequence rather than from path existence alone:

$$G_v(s) = \begin{cases} R_v(s)_1, & R_v(s) \neq \langle \rangle, \\ \perp, & \text{otherwise.} \end{cases} \quad (9)$$

This matches the implementation’s distinction between a resolved value-less node and a resolved node with a preferred value.

Definition 1 (Productive compiler node). *A node v is productive when there exists at least one continuation $s \in \Sigma^*$ with $R_v(s) \neq \langle \rangle$. Every non-root node reachable in a nonempty builder trie is productive because it lies on a path to an inserted value-bearing terminal. Canonicalization preserves productivity, and every accepting node created by uniform contraction is productive because it stores one value. The empty-trie root is the only trivial exception needed below.*

An accepting node defines a continuation cylinder: if traversal reaches v on path label a , then every continuation az , $z \in \Sigma^*$, resolves at v . Under backward traversal this corresponds to every logical word ending in $\text{rev}(a)$.

IV. MODE-SPECIFIC SEMANTIC CANONICALIZATION

A. Local Descriptors

The public reduction modes define different local equality contracts. Let $L_m(v)$ be the local descriptor under mode m :

$$L_R(v) = \mathcal{R}(v), \quad (10)$$

$$L_U(v) = \mathcal{S}(v), \quad (11)$$

$$L_D(v) = \begin{cases} \text{dom}(\mathcal{P}(v)), & \text{if } \mathcal{P}(v) \text{ is qualified,} \\ \text{rank}(\mathcal{R}(v)), & \text{otherwise.} \end{cases} \quad (12)$$

This matches the concrete `RankedLocalDescriptor`, `UnorderedLocalDescriptor`, and `DominantLocalDescriptor`. In dominant mode, a non-qualified node falls back to the ranked descriptor. The fallback is intentionally stricter than preferred-value equality and therefore prevents some merges that would be possible under a theoretical minimal `get()` equivalence.

Table I states the observable contract and its limits.

B. Recursive Signatures

The implementation reduces children before their parent and constructs an immutable semantic signature from the current local descriptor, the accepting flag, and child signatures sorted by edge label.

Definition 2 (Reduction signature). *For a fixed mode m , define*

$$\sigma_m(v) = \left(L_m(v), A(v), \text{sort}\{(a, \sigma_m(u)) : (v \xrightarrow{a} u) \in E\} \right). \quad (13)$$

Define $v \equiv_m u$ iff $\sigma_m(v) = \sigma_m(u)$.

The signature is structural and semantic: equal local behavior is insufficient when descendants differ, and equal descendants are insufficient when local values differ. Sorting child descriptors makes identity independent of incidental map iteration order. The accepting flag participates explicitly, preventing a normal leaf from being identified with a continuation-accepting leaf merely because their local values match [19].

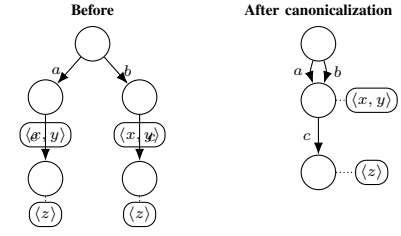


Fig. 2. Canonicalization as quotient-DAG sharing. The two symbolic pre-reduction subtrees have equal local outputs and recursively equal labelled descendants, hence the same signature. Their incoming contexts may therefore share one physical representative. The figure shows ranked outputs only for readability; it is not a contraction example.

C. Why Count Aggregation Requires a Proof

When a signature already has a canonical representative, the implementation does not discard the new node’s local evidence. It executes `mergeLocalCounts()`, summing equal-value counts, and later recomputes the local ranking when freezing the immutable node [19]. Because counts are deliberately omitted from several descriptors, the equivalence argument would be incomplete unless aggregation were shown not to invalidate the descriptor.

Let c_i be the local count map of the i th node in one canonical equivalence class, and let

$$c_\Sigma(x) = \sum_i c_i(x) \quad (14)$$

be the count map stored by the merged representative.

Lemma 1 (Rank-order aggregation stability). *Suppose every c_i induces the same ordered value sequence $\langle x_1, \dots, x_k \rangle$ under the implementation’s deterministic ranking. Then c_Σ induces the same ordered sequence.*

Proof. Fix $a < b$. In every component i , $c_i(x_a) \geq c_i(x_b)$ because x_a precedes x_b in the common ranked sequence. Summing yields $c_\Sigma(x_a) \geq c_\Sigma(x_b)$. If at least one component inequality is strict, the aggregate inequality is strict and the order is fixed by count.

Suppose instead that the aggregate counts are equal. Then every component inequality on this pair must have been an equality. If the textual length or lexical tie-break distinguishes the pair, that intrinsic comparison is unchanged after aggregation. The only remaining case is equal textual representations of distinct values, where the implementation uses first-seen insertion order. Ranked-signature equality implies that every colliding node presents the pair in the same order. Moreover, merging equal-signature nodes adds counts to existing keys rather than introducing a new value key, because the ranked descriptors contain the same complete value sequence. The canonical representative therefore retains its already-correct key order, which is the first-seen order used when the aggregate summary is frozen. Thus x_a remains before x_b in every possible tie case. The argument holds for every ordered pair, so the complete sequence is preserved. $\square$

The lemma covers ranked mode and dominant-mode fallback nodes. It also implies that recomputing

TABLE I
REDUCTION CONTRACTS AND INTENTIONALLY NON-PRESERVED INFORMATION.

Mode / operation	Local identity used by compiler	Guaranteed value observable	Not guaranteed by that contract
Ranked canonicalization	Full ordered local value sequence	Ordered <code>getAll()</code> result for every key	Original per-key counts; original node identity
Unordered canonicalization	Distinct local value set	<code>getAll()</code> membership set for every key	Ordering, preferred <code>get()</code> , original counts
Dominant canonicalization	Qualified winner; ranked fallback otherwise	Preferred <code>get()</code> result for every key	Alternative membership/order at qualified nodes; original counts; global minimality
Uniform accepting-leaf contraction	One uniform value in the current reduced region	Existing ranked singleton, membership, and preferred value on the pre-contraction lookup domain	Arbitrary-input exact-language equivalence; original count history

`LocalValueSummary` during `freeze()` does not invalidate a ranked signature after canonical collisions.

Lemma 2 (Qualified-dominant aggregation stability). *Assume the same value x is qualified dominant in every count map c_i under thresholds (p, q) . Then x remains qualified dominant in c_Σ under the same thresholds.*

Proof. Let $C_i = \sum_y c_i(y)$. From the percentage condition, $100c_i(x) \geq pC_i$ for every i . Summation gives $100c_\Sigma(x) \geq p \sum_i C_i = pC_\Sigma$.

For the ratio condition, let $s_i = \max_{y \neq x} c_i(y)$. For every competitor $y \neq x$, $c_i(y) \leq s_i$, while qualification gives $c_i(x) \geq qs_i$. Hence

$$c_\Sigma(x) = \sum_i c_i(x) \geq q \sum_i s_i \geq q \sum_i c_i(y) = qc_\Sigma(y). \quad (15)$$

For $q > 1$, this makes x strictly more frequent than every present competitor. For the allowed boundary case $q = 1$, suppose $c_\Sigma(x) = c_\Sigma(y)$ for some competitor y . Since $c_i(x) \geq c_i(y)$ in every component, equality of the sums implies $c_i(x) = c_i(y)$ for every i . Because $c_i(x) > 0$, y must be present in every component, including the first canonical representative; if y were absent anywhere, the aggregate inequality would be strict. In every component the deterministic tie-break therefore placed x before y . The representative already contains both keys in that order, and later merges only increment their counts. Hence the same tie-break remains available after aggregation and x remains preferred. Together with the percentage inequality, x remains qualified dominant for all allowed $q \geq 1$. $\square$

This result is specific to the implementation's monotone percentage and winner-ratio predicates. It explains why a dominant descriptor can omit losing alternatives while the merged representative still preserves the preferred output.

For unordered mode the argument is simpler: each colliding node has the same distinct value set, and summing counts preserves exactly that set, although it may change the ranking and thus `get()`.

D. Canonicalization Congruence

Define the declared observable $\mathcal{O}_m(v, s)$ as

$$\mathcal{O}_m(v, s) = \begin{cases} R_v(s), & m = R, \\ S_v(s), & m = U, \\ G_v(s), & m = D. \end{cases} \quad (16)$$

Definition 3 (Continuation-observable equivalence). *For a fixed mode m , define*

$$v \approx_m u \iff \mathcal{O}_m(v, s) = \mathcal{O}_m(u, s) \text{ for every } s \in \Sigma^*. \quad (17)$$

Thus $\approx_m$ is the semantic relation induced by all future continuations, while $\equiv_m$ is the compiler's computable signature relation.

Theorem 1 (Subtree-signature congruence). *For a fixed reduction mode m , consider two pre-interning reduced nodes v and u produced by the recursive compiler, possibly after local or descendant uniform contraction. If $v \equiv_m u$, then for every remaining traversal string $s \in \Sigma^*$,*

$$\mathcal{O}_m(v, s) = \mathcal{O}_m(u, s). \quad (18)$$

Replacing any subtree by another representative of the same signature class therefore preserves the declared observable at every ancestor.

Proof. Proceed by induction on the maximum remaining reachable depth below the compared nodes.

Signature equality gives equal local descriptors and equal accepting state. At $s = \varepsilon$, ranked descriptors give equal ordered sequences, unordered descriptors equal membership sets, and dominant descriptors either the same qualified winner or the same ranked fallback sequence.

For non-empty s , first consider $A(v) = A(u) = 1$. Lookup returns the current node before consuming the next code unit, so the same local descriptor gives the same declared observable for every remaining continuation. Otherwise $A(v) = A(u) = 0$. Equal signatures then give the same sorted outgoing edge labels paired with equal child signatures. For $s = az$ with first traversal code unit a , either both nodes lack edge a , or both contain it and the reached children have equal signatures. The induction hypothesis yields equal observables on z . Therefore the nodes are indistinguishable by the declared mode observable for every continuation, regardless of whether accepting descendants were created by earlier contraction.

Equivalently, $\equiv_m$ is substitutive in every edge-labelled trie context: replacing a child by an $\equiv_m$ -equivalent child cannot change the parent's declared observable.

Canonicalization replaces nodes only by equal-signature representatives. The aggregation-stability lemmas ensure that count summation does not change the representative's relevant local descriptor before freezing. Equal parent signatures also

contain the same edge-labelled child signatures. Because one shared `ReductionContext` interns descendants before their parents, equal child signatures resolve to the same canonical child instances; this is the implementation invariant required by `mergeChildren(...)`, whose identity check therefore remains compatible with signature equality. Repeated bottom-up replacement preserves the selected observable globally on the already-produced reduced graph. This theorem concerns the intern/canonicalization step itself; it does not assert that prior accepting-leaf contraction preserved the unreduced exact lookup function over all strings. $\square$

Theorem 1 establishes soundness of the compiler relation:

$$\equiv_m \subseteq \approx_m \quad \text{for every public reduction mode } m. \quad (19)$$

For two modes the finite trie structure supports the converse as well.

Theorem 2 (Completeness for contraction-free ranked and unordered modes). *Consider a compiler run with uniform-subtree contraction disabled, so $A(v) = 0$ for every node. Let $m \in \{R, U\}$ and let v, u be productive finite compiler nodes. Then*

$$v \approx_m u \iff \sigma_m(v) = \sigma_m(u). \quad (20)$$

Proof. The reverse implication is Theorem 1. For the forward implication, proceed by induction on maximum finite subtree height. Equality at $s = \varepsilon$ gives equal local ranked sequences when $m = R$ and equal local membership sets when $m = U$; all accepting flags are false by hypothesis.

The outgoing edge-label sets must agree. If v had an edge a absent from u , the reached child of v is productive; choose a continuation t on which that child has a non-empty observable. Then at distinguishes v from u . Finally, for every common edge a , the child nodes must themselves be observationally equivalent, otherwise at for a distinguishing child continuation t would distinguish the parents. The induction hypothesis gives equal child signatures. Equal local descriptor, equal non-accepting state, and equal edge-labelled child signatures are exactly Eq. (13); hence the parent signatures are equal. $\square$

The contraction-free hypothesis is substantive. After accepting-leaf generalization, an accepting singleton node can in principle be observationally equivalent to a larger finite non-accepting decision subtree that explicitly covers every continuation with the same singleton value. Their accepting flags and signatures differ even though their continuation observables coincide. Thus production signature equality remains a sound congruence after contraction, but the paper does not claim it is the coarsest behavioral equivalence on contraction-enabled graphs.

Corollary 1 (Ranked mode). *Ranked canonicalization preserves the ordered `getAll()` value sequence for every key, but not original per-key occurrence counts.*

Corollary 2 (Unordered mode). *Unordered canonicalization preserves candidate membership for every key. It need not preserve candidate order or the preferred `get()` value.*

Corollary 3 (Dominant mode). *Dominant canonicalization preserves the preferred `get()` value for every key. At qualified nodes it may merge different losing alternatives; at non-qualified nodes ranked fallback preserves a stronger local distinction than the theorem requires.*

For the serialized patch-command strings used during dictionary reduction, equality of the preferred value means equality of the serialized patch. With the same traversal direction, deterministic command specialization therefore yields the same transformed token for the same input. Thus dominant and ranked preferred-value preservation induce a stemming-output corollary without requiring the reduction proof to model patch execution internally.

E. Quotient Acyclicity and Relative State Minimality

Canonicalization has structural consequences that follow from the recursive relation itself. Let $h(v)$ be the maximum number of child edges on a path from v to a descendant leaf, with $h(v) = 0$ at a leaf.

Lemma 3 (Signature classes preserve height). *If $\sigma_m(v) = \sigma_m(u)$, then $h(v) = h(u)$. Consequently, no signature class contains a node and one of its proper descendants, and canonical quotienting cannot create a directed cycle.*

Proof. The result follows by structural induction. Equal leaf signatures have no child descriptors and therefore equal height zero. At an internal node, equal signatures contain the same edge-labelled child signatures; the induction hypothesis gives equal child heights, so the maxima defining the parent heights are equal. A proper descendant of a finite node has strictly smaller height, hence cannot have the same signature as its ancestor. Identifying equal-signature nodes therefore preserves acyclicity. $\square$

Let V be the mutable nodes and $\mathcal{C}_m$ the encountered signature classes for a run. The reduced graph contains exactly one physical representative for each class:

$$N_m = |\mathcal{C}_m| \leq |V|. \quad (21)$$

The quotient is canonical at the level of equivalence classes even though the Java object chosen as the first representative is an implementation detail.

For contraction-free ranked and unordered runs, Theorem 2 strengthens Eq. (21). Consider only quotientings of the same productive node-output trie that preserve traversal direction, accepting semantics, labelled transitions, and the selected node-output observable; no output pushing or alternative transducer encoding is allowed. Any two source nodes mapped to one quotient state must be continuation-observationally equivalent. Theorem 2 says those classes are exactly the signature classes. Therefore the Radixor canonicalization-only ranked/unordered quotient uses the minimum possible number of states among such quotientings. This is *relative state minimality under the fixed representation contract*, not a claim of globally minimal sequential-transducer size. Dominant mode does not receive this corollary because ranked fallback intentionally refines preferred-output equivalence.

F. Termination, Decidability, and Reference-Implementation Cost

For a finite mutable trie, the abstract compiler is a total deterministic computation under deterministic terminating value primitives. The recursive reducer visits descendants before their parent, every recursive branch is finite, Lemma 3 rules out cycles in the quotient, and freezing follows the finite quotient DAG. Signature equality is decidable by finite structural equality. In contraction-free ranked and unordered runs, Theorem 2 therefore also gives a decision procedure for continuation-observable equivalence on productive compiler nodes: compare their signatures. This is a mathematical termination/decidability statement under unbounded resources; a concrete recursive JVM execution can still fail on a finite but sufficiently deep or large instance through stack or heap exhaustion.

The semantic result should not be confused with the complexity of the current Java implementation. Let N be the mutable node count, H the maximum trie height, k_v the number of local values at node v , $K = \sum_v k_v$, and d_v its out-degree. Building each local summary sorts its values, while signature construction sorts child descriptors. More subtly, the frozen `ReductionSignature.hashCode()` recursively hashes its child signatures through `ChildDescriptor`; hashes are not cached [19]. Under constant-time value hashing/equality and the ordinary expected-cost hash-table model, the reference implementation therefore has the descriptive bound

$$O\left(\sum_v k_v \log k_v + \sum_v d_v \log d_v + (N + K)H\right), \quad (22)$$

where the last term bounds repeated recursive signature hashing/equality across ancestor contexts. In a degenerate chain with $K = O(N)$ and $H = \Theta(N)$, the recursive-hash component admits $\Theta(N^2)$ work even without adversarial hash collisions. This differs materially from specialized linear-time acyclic-automaton minimization algorithms such as Revuz’s [9]; the algorithms solve related but not identical representation problems, and the present reference implementation is optimized for semantic clarity and deterministic behavior rather than a proven asymptotic optimum.

Equation (22) is not a wall-clock benchmark and makes no claim about JVM constants. It also exposes a concrete engineering boundary: memoizing immutable signature hashes or representing children by already-interned canonical IDs could avoid repeated descendant re-hashing, but any performance consequence would require measurement. The semantic proofs in this paper do not depend on that implementation choice.

V. ACCEPTING-LEAF GENERALIZATION

A. Implementation Predicate

Uniform-subtree contraction is controlled independently by `contractUniformSubtrees`. Generic `ReductionSettings.withDefaults(...)` leaves it disabled; patch-command compilation enables it through `StemmerPatchTrieLoader` [19].

At the current node, after every child has already been reduced, the implementation contracts only when:

- 1) the node has at least one reduced child;
- 2) the current local value map is empty or contains exactly one value;
- 3) every reduced child is a leaf containing exactly one value; and
- 4) all present local/child values are equal to one common value x .

The compiler then discards the children, stores a single local count for x , and sets $A(v) = 1$ [19]. The stored count is reset to one; count history is intentionally outside the contraction contract.

B. Local Lookup Domain

For a node v , define its value-bearing lookup domain

$$\mathcal{D}(v) = \{s \in \Sigma^* : R_v(s) \neq \langle \rangle\}. \quad (23)$$

On this domain, $S_v(s)$ is non-empty and $G_v(s)$ is defined as its first ranked value. A normal single-value leaf has $\mathcal{D}(v) = \{\varepsilon\}$. An accepting single-value leaf has $\mathcal{D}(v) = \Sigma^*$ because lookup stops before consuming any remaining code unit.

Immediately before a contraction at v , every defined lookup in $\mathcal{D}(v)$ returns the same singleton value semantics: $R_v(s) = \langle x \rangle$, $S_v(s) = \{x\}$, and $G_v(s) = x$. The local terminal, if present, is x , and each child is a single-value leaf carrying x . A child may itself be accepting because contraction is recursive; even then, all value-level observables in its already-generalized local domain are the same singleton x .

Proposition 1 (Value-semantic domain extension). *Let a node v satisfy the uniform-contraction predicate with common value x , and let v' be the accepting leaf that replaces it. Then*

$$\mathcal{D}(v) \subseteq \mathcal{D}(v') = \Sigma^*, \quad (24)$$

$$R_{v'}(s) = R_v(s) = \langle x \rangle, \quad (25)$$

$$S_{v'}(s) = S_v(s) = \{x\}, \quad (26)$$

$$G_{v'}(s) = G_v(s) = x \quad \forall s \in \mathcal{D}(v). \quad (27)$$

Thus contraction preserves every already-defined value-level result in the replaced region and can only add singleton value results for previously undefined continuations. Original occurrence counts are not preserved.

Proof. By the predicate, all defined local and child results have exactly the singleton value x . Therefore $R_v(s) = \langle x \rangle$, $S_v(s) = \{x\}$, and $G_v(s) = x$ for every $s \in \mathcal{D}(v)$. The replacement node v' stores only x and has $A(v') = 1$. By lookup semantics, $F_{v'}(s) = v'$ for every $s \in \Sigma^*$, so the same singleton sequence, set, and preferred value are returned everywhere. The stated inclusion and equalities follow immediately. $\square$

Corollary 4 (Monotonicity of contraction steps). *Along the subsequence of valid contraction operations in a bottom-up compilation, each contraction preserves all value-level results already defined in its local region and the set of traversal strings receiving a non-empty value result is monotone non-decreasing. Canonicalization steps between contractions obey their own mode-specific contract from Theorem 1.*

This corollary is stronger and more compositional than restricting the argument to the original training keys. If a child

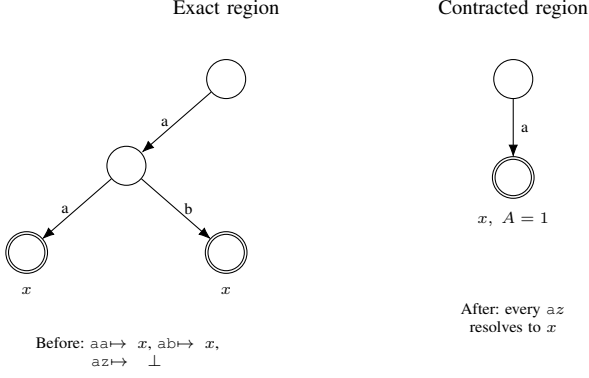


Fig. 3. Uniform contraction is domain extension, not all-string exact equivalence. The illustration uses forward traversal for readability; backward production tries generalize over remaining logical prefixes after a suffix has been consumed.

was already generalized by an earlier contraction, the parent’s contraction preserves that generalized child domain as well. It does not attribute preferred-value preservation to unordered canonicalization; that operation retains only its membership contract.

C. Counterexample to All-String Equivalence

The same proposition proves why the pre-contraction node and its accepting-leaf replacement cannot be identified as an equivalent pair under Theorem 1 when the comparison domain is all of Σ^* . The frozen unit tests construct a forward trie with

$$aa \mapsto x, \quad ab \mapsto x. \quad (28)$$

Without contraction, az follows a missing edge and returns $\perp$. With contraction, the a region becomes an accepting leaf and $az \mapsto x$ [19]. Hence, for the exact lookup function over Σ^* ,

$$G_{\text{exact}} \neq G_{\text{contracted}}. \quad (29)$$

The difference is the defining semantic effect of the transformation: a region that has become single-valued is reinterpreted as a rule-like decision frontier. Any runtime benefit is a separate empirical question.

D. From Selected Value to Stem Output

The domain-extension theorem is a theorem about trie values. In a Radixor stemming model that value is a serialized patch command c , not a constant output stem. Let $C(w)$ denote the command selected for logical input word w when a trie value is defined, and let

$$H(w) = \text{Exec}(C(w), w) \quad (30)$$

be the resulting stemmer output on that selected-command domain. Here Exec denotes the deterministic patch-command semantics for the fixed traversal direction. Compiled commands also contain applicability logic such as minimum source-length requirements, so selecting the same command need not imply the same edit trace, and certainly does not imply a constant output string [19].

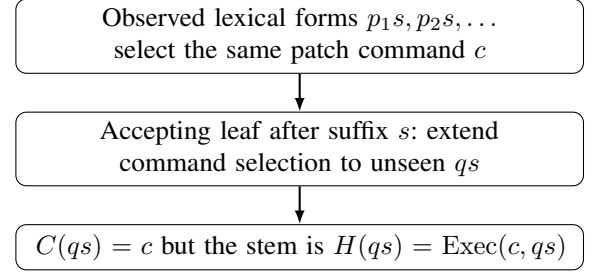


Fig. 4. Accepting-leaf generalization extends a decision region for a transformation program. It does not assign one constant stem to all words in the region. Prefixes p_i, q and suffix s are symbolic; the figure is explanatory, not an empirical language example.

Consequently, if contraction at a traversal region selects command c , the semantic extension is

$$\begin{aligned} C(w) &= c & \text{for every newly covered } w, \\ H(w) &= \text{Exec}(c, w). \end{aligned} \quad (31)$$

This is best described as *command-selection generalization*. Whether $H(w)$ is linguistically desirable for an unseen form is a separate stemming-quality question.

E. Backward Tries as Suffix-Conditioned Rules

For $d = \text{BACKWARD}$, a traversal prefix a corresponds to logical suffix $\text{rev}(a)$. An accepting leaf reached after consuming a therefore defines the logical-word region

$$\Sigma^* \text{rev}(a). \quad (32)$$

In other words, once the compiler establishes a uniform patch command c for a suffix region, the accepting leaf selects c regardless of unseen material to the left, and the resulting stem is $\text{Exec}(c, w)$. The compiled trie therefore ceases to behave like a finite exact-path table for that region and behaves like an induced suffix-conditioned *command rule*. This is analogous in spirit to the way classical stemming rules apply beyond memorized forms [3], [5], but the rule here is induced from uniform compiled lexical evidence rather than authored directly.

This interpretation also clarifies why contraction and canonicalization should remain separate in the scientific vocabulary. Canonicalization says that two already-defined decision subgraphs are observationally interchangeable. Contraction says that a uniform decision subgraph justifies closing the same command-selection decision over additional continuations. Neither statement establishes a linguistic morpheme boundary or validates the resulting conflation.

VI. COMPOSITION OF CANONICALIZATION AND CONTRACTION

The implementation does not run two global passes. Instead, `reduce()` recursively reduces children, attempts current-node contraction, creates the resulting signature, and interns that signature. This ordering has three consequences.

First, a child may already be an accepting leaf when the parent is tested for uniformity. The parent’s contraction can therefore absorb a previously generalized child region, and

Proposition 1 remains compositional because all defined child outputs are the same x .

Second, the accepting flag is included in `ReductionSignature`. After contraction, two accepting leaves are canonicalized only when the selected local descriptor and accepting state match. This prevents accidental merging of a normal exact leaf with a generalized accepting leaf.

Third, canonicalization can aggregate local evidence before an ancestor is considered for contraction. Under dominant reduction, for example, two equal-signature children may carry different losing alternatives but the same qualified winner. They canonicalize safely by Lemma 2, yet their merged child can contain more than one stored value and therefore fail the later `isSingleValueLeaf` predicate. Thus contraction opportunities depend on the actual ordered bottom-up compiler, not merely on a post-hoc scan of the unreduced trie. This is another reason not to treat contraction as a generic “compress after minimization” phase.

The formal results nevertheless compose cleanly at the level of the selected contract. Each canonicalization step preserves the observable declared by its reduction mode, including when the compared nodes are already accepting. Each contraction step preserves the ranked singleton sequence, singleton membership, and preferred singleton value at every point of its pre-contraction value domain, while possibly extending that domain. Consequently ranked compilation preserves ranked value behavior on previously defined inputs, unordered compilation preserves membership on previously defined inputs, and dominant compilation preserves preferred output on previously defined inputs. With contraction enabled, none of these statements implies observational identity with the unreduced exact trie over all strings.

VII. IMPLEMENTATION CORRESPONDENCE

Table II maps the formal objects to the frozen Java implementation. This correspondence is part of the evidence boundary: the theorems are intended to describe the actual compiler rather than an idealized trie algorithm.

A. Patch-Command Value Staging

For dictionary-built stemming models, the reduction relation is evaluated on serialized patch-command `String` values. The `loadCompiled(...)` path first builds and reduces that `String`-valued trie, then `compilePatchTrie(...)` maps each serialized patch deterministically to a `CompiledPatchCommand` while preserving the already-compiled graph, local ordering/count arrays, shared-node topology, and accepting flags through `FrequencyTrieBuilders.mapValues(...)` [19].

This staging matters to the stemming-output corollary. Equality of preferred values at reduction time means equality of the serialized patch command. With a fixed traversal direction, the subsequent compiler maps the same serialized patch to the same deterministic transformation semantics. The theorem therefore does not depend on object-identity equality between independently instantiated runtime command objects.

B. Canonical DAG and Persistence

Canonical representatives are frozen with an identity cache, so a shared `ReducedNode` becomes one shared `CompiledNode` instance. The resulting structure is a DAG even though the mutable source is a tree. The public `size()` method explicitly counts unique compiled node identities rather than logical paths [19].

The binary format persists the accepting flag and canonical child references. Accepting nodes were introduced in format version 6; current format version 7 additionally stores distinct values in a stream-local value table [19]. The fingerprint includes metadata, accepting state, edge labels, child identities, values, and counts, so content identity reflects the compiled structure rather than process-local object addresses.

C. Counts Are Evidence, Not the Reduction Contract

The implementation exposes `getEntries()` with local counts, but its documentation explicitly permits those counts to be aggregated across build-time nodes that canonicalize to the same compiled node [19]. The information relation is therefore a partial order rather than a linear hierarchy:

$$\begin{aligned} \text{ordered count entries} &\implies \text{ranked values,} \\ \text{ranked values} &\implies \text{membership,} \\ \text{ranked values} &\implies \text{preferred value.} \end{aligned} \tag{33}$$

Membership and preferred value are incomparable summaries: a set of candidates does not identify which candidate ranks first, while the preferred value does not identify the remaining candidates. Different reduction modes choose different preservation points in this partial order. Uniform contraction adds an orthogonal dimension: extension of the input domain on which those value observables are defined.

VIII. IMPLEMENTATION VERIFICATION EVIDENCE

Proof establishes the abstract correctness claim. Tests establish that the frozen implementation behaves consistently with the source-level model on generated and hand-crafted cases. The two evidence types should not be confused.

A. Mode-Specific Unit Cases

The frozen `FrequencyTrieTest` exercises all three public modes. Ranked mode merges subtrees with the same ordered `getAll()` results and keeps differently ordered nodes separate. Unordered mode permits merging when membership matches despite different local order. Dominant mode merges nodes with the same qualified winner and falls back when the winner is not strong enough [19].

A synthetic repeated-suffix fixture records both mutable `buildTimeSize()` and compiled `size()` and requires at least a 50 percent node reduction while checking representative lookups. This is a non-vacuity test of structural sharing, not a multilingual effect estimate.

TABLE II
CORRESPONDENCE BETWEEN FORMAL OBJECTS AND THE FROZEN RADIXOR IMPLEMENTATION.

Formal concept	Implementation element	Relevant behavior
Mutable trie node v	<code>MutableNode<V></code>	Child map and local value-count map
Deterministic ranking $\mathcal{R}(v)$	<code>LocalValueSummary.of(...)</code>	Count order plus text-length, lexical, and first-seen tie-breaking
Mode descriptor $L_m(v)$	<code>RankedLocalDescriptor</code> , <code>UnorderedLocalDescriptor</code> , <code>DominantLocalDescriptor</code>	Ranked sequence, distinct set, or qualified winner with ranked fallback
Recursive signature $\sigma_m(v)$	<code>ReductionSignature</code>	Local descriptor, sorted child descriptors, and accepting flag
Canonical equivalence class	<code>ReductionContext</code>	Map from signature to canonical <code>ReducedNode</code>
Count aggregation	<code>mergeLocalCounts(...)</code>	Sums evidence from colliding signature-equivalent nodes
Uniform contraction	<code>uniform-contraction helper</code>	Replaces uniform non-leaf region by one single-value accepting leaf
Immutable runtime graph	<code>CompiledNode<V></code>	Sorted edges, shared child references, ordered values/counts, accepting flag
Early-accept lookup	<code>findNode(...)</code>	Returns accepting node before consuming remaining key characters
Patch-command specialization	<code>loadCompiled(...)</code>	Enables uniform contraction for patch-command compilation and maps textual patches to compiled commands

B. Accepting-Leaf Cases

The same test class distinguishes exact lookup from uniform contraction. With contraction disabled, $aa \mapsto x$ and $ab \mapsto x$ do not imply $az \mapsto x$. With contraction enabled, the a region becomes accepting and the unseen continuation is defined. Additional tests verify that accepting-leaf semantics survive binary round trip and value-type mapping [19].

C. Property-Based Tests

The property suite uses generated trie scenarios across all `ReductionMode` values. It executes 80 trials for deterministic recompilation, 80 for alignment among `get()`, `getAll()`, and `getEntries()`, 40 for binary round-trip equivalence, and 60 for builder reconstruction from compiled state [19]. These properties protect deterministic compilation and persistence behavior. They do not prove Theorem 1 for all possible tries; the theorem derives from the recursive signature construction and aggregation lemmas.

D. Verification Boundary

The implementation evidence is intentionally not described as “exhaustive proof over Unicode.” The mutable and compiled algorithms operate on UTF-16 `char` transitions; normalization modes can transform lookup keys before traversal; arbitrary user value types can define equality and textual tie-breaking in application-specific ways. The theorem is parameterized by the deterministic value equality/ranking behavior actually supplied to the trie.

IX. DISCUSSION

A. Canonicalization as Output-Aware Hash-Consing

The compiler can be understood as output-aware hash-consing of acyclic decision subgraphs. A recursively computed signature names the behavior that a mode promises to preserve, and a map reuses the first canonical representative of that signature. This resembles state merging in acyclic automata and transducers [9]–[11], [13], but the local observation and preservation policy are specific to Radixor’s ranked/count-derived API contracts.

The distinction is especially clear across the three modes. Ranked canonicalization treats alternative order as semantic. Unordered canonicalization declares order irrelevant and can therefore merge more aggressively. For both modes, Theorem 2 shows that in the contraction-free exact-trie phase the recursive signature is not merely safe but complete for the chosen continuation observable; that quotient is relatively state-minimal within the fixed node-output representation. Once accepting leaves are introduced, the signature remains safe but is not claimed to be the coarsest possible behavioral equivalence. Dominant canonicalization instead declares only a qualified preferred result important at certain nodes but preserves ranked distinctions when the evidence is insufficiently decisive. Its signature is therefore deliberately a refinement of preferred-output equivalence. The compiler exposes an explicit continuum between representation sharing and retained ambiguity rather than one universal notion of minimality.

B. Why Aggregation Stability Matters

Ignoring count aggregation would leave a subtle correctness gap. The canonical representative is not frozen with one node’s original counts; it accumulates evidence from every colliding node. The ranked and dominant lemmas show that the descriptors are closed under precisely this aggregation operation. This is stronger than observing that unit tests happen to pass: it explains why the implementation may safely retain evidence counts without undermining the output contract used to decide equivalence.

The same reasoning also clarifies what is *not* safe. Unordered mode does not constrain ranking, so aggregated counts can legitimately select a different preferred value from the one an individual unreduced key had. An application that calls `get()` should therefore not choose unordered mode merely because its candidate set is unchanged.

C. Accepting Leaves as Induced Command Rules

Uniform contraction is more than common-subexpression elimination. Once a region is accepting, the remaining input is irrelevant to the *selected trie value*. For a stemming model that value is a patch program c , so traversal space is extended

to a constant *command-selection* function while the final stem remains input-dependent through $\text{Exec}(c, w)$.

This gives a precise compiler interpretation of stemming generalization. Classical stemmers deliberately apply affix rules beyond memorized lexical items [3], [5]; lexicon-sensitive approaches such as Krovetz emphasize that morphological normalization is an inference problem constrained by lexical evidence and ambiguity [6], [7]. Radixor’s accepting leaf provides a different mechanism: lexical examples first induce patch commands, and a local uniformity condition then promotes one compiled command decision to unseen continuations. Early acceptance therefore embodies an induced decision rule, not a claim that all covered words share one literal stem.

This mechanism can define transformations for forms unrepresented by an exact training-trie path, including forms that are out of vocabulary relative to the inducing lexical resource. Its linguistic quality is not entailed by Proposition 1. An extended command may conflate forms that should remain distinct or fail to make a desired conflation; classical stemming evaluation describes such behavior in terms of over- and under-stemming, while retrieval-oriented evaluation asks whether the resulting conflations help a task [17], [18]. The present theorem isolates where the new decision comes from so that those empirical questions can be tested separately.

D. Relationship to Finite-State Morphology

Finite-state morphology traditionally emphasizes automata/transducers that encode lexical–surface relations for morphological analysis and generation [14]–[16]. Radixor’s patch-command trie is narrower: it compiles a unidirectional normalization decision over lexically induced transformations rather than a bidirectional morphological grammar. Its canonical target is resource-defined and need not be a linguistic root or lemma. The canonicalization theorem nonetheless belongs naturally in the finite-state tradition because it identifies subgraphs by future observable behavior. Accepting-leaf contraction adds a command-selection generalization step that ordinary language-preserving DFA minimization does not perform.

E. No Runtime Causality Claim

A smaller or shallower compiled structure may improve cache behavior or lookup latency, but this paper does not infer a causal speedup from the formal reductions. Runtime depends on edge-search strategy, dense lookup tables, command materialization, workload distribution, JVM behavior, and model shape. Those effects require controlled benchmarking. The present contribution is stronger in a different sense: the semantic consequences of canonicalization and contraction follow from the algorithm itself and do not depend on hardware.

X. THREATS TO VALIDITY

A. Implementation Snapshot

The proof is matched to Radixor commit 0c3b13f. Future implementations could change local ranking, dominance thresholds, contraction predicates, or persistence semantics. Such changes would require re-checking the correspondence, although the abstract proof pattern remains reusable.

B. Value Equality and Textual Tie-Breaking

The generic trie accepts arbitrary value types. Equality, hashing, and `toString()` influence descriptor identity and deterministic tie-breaking. The results assume these operations are deterministic and consistent for the values used in one compilation. The serialized `String` patch values used during dictionary reduction have deterministic equality and textual ordering; runtime command specialization occurs only afterwards. The generic trie API nevertheless permits pathological user-defined objects outside the paper’s practical scope.

C. Counts Versus Values

The theorem concerns the observable selected by the reduction mode, not original insertion history. Canonical merges aggregate counts, and uniform contraction resets its stored single value to count one. Any application whose semantics depend on unreduced per-key counts must not infer their preservation from ranked or dominant output equivalence.

D. Dominant Mode Is Conservative, Not Minimal

Dominant fallback uses a ranked descriptor whenever the local winner does not meet configured thresholds. Consequently, two nodes can be behaviorally identical under preferred `get()` yet remain unmerged because the compiler intentionally preserves more information. We prove safety of performed merges, not maximality of compression.

E. Contraction Changes the Function Domain

Uniform contraction is not exact-language preservation over Σ^* . It preserves the already-defined ranked singleton sequence, membership, and preferred value in the contracted region while extending the local lookup domain. Any evaluation that compares contraction on arbitrary inputs must therefore treat newly defined outputs as generalization behavior, not as a failure of canonicalization equivalence.

F. Compiler Complexity Scope

Equation (22) characterizes the reference implementation only under explicit expected-hash-table and constant-time value-operation assumptions. Adversarial hash collisions, expensive user-defined value equality/hashing, allocation costs, or JVM behavior can worsen concrete execution. Conversely, the quadratic chain example is a structural worst-case witness for repeated recursive hashing, not a claim that production language models exhibit quadratic wall-clock scaling. The semantic quotient and the complexity of this particular Java realization must remain separate.

G. Linguistic Generalization Scope

Accepting-leaf contraction proves conservative extension of command-selection semantics on an already uniform value region. It does not prove that the extended command corresponds to a valid morpheme, lemma relation, or retrieval-relevant conflation. Stemming errors can manifest as over- or under-stemming [17], and retrieval effects require task-level evaluation

[18]. Moreover, a compiled patch command can contain source-length applicability behavior, so a common selected command does not imply an identical edit trace for all covered words. The formal claim is therefore intentionally about selection and deterministic execution, not linguistic correctness.

H. Normalization and Traversal Direction

The formal analysis works after key normalization. Case conversion and diacritic handling occur outside the reduction theorem, and backward traversal maps logical prefixes/suffixes differently from forward examples. Traversal is over UTF-16 code units, so supplementary Unicode characters are represented by surrogate pairs whose code units are visited separately and in reverse code-unit order under backward traversal. Equation (1) makes this machine-level direction explicit, but application-level semantics still depend on the chosen normalization metadata and character repertoire.

XI. REPRODUCIBILITY

The article is grounded in the public Radixor repository at commit `0c3b13f485a9ad0b460c0931e4497ea95bed66a1` [19]. The primary source files are:

- `src/main/java/org/egothor/stemmer/FrequencyTrie.java`;
- `src/main/java/org/egothor/stemmer/ReductionMode.java`;
- `src/main/java/org/egothor/stemmer/ReductionSettings.java`;
- `src/main/java/org/egothor/stemmer/trie/LocalValueSummary.java`;
- `src/main/java/org/egothor/stemmer/trie/ReductionSignature.java`;
- `src/main/java/org/egothor/stemmer/trie/ReductionContext.java`;
- `src/main/java/org/egothor/stemmer/trie/ReducedNode.java`;
- `src/main/java/org/egothor/stemmer/trie/ChildDescriptor.java`;
- `src/main/java/org/egothor/stemmer/trie/CompiledNode.java`;
- the three local descriptor classes;
- `src/main/java/org/egothor/stemmer/StemmerPatchTrieLoader.java`;
- `src/main/java/org/egothor/stemmer/FrequencyTrieBuilders.java`; and
- `src/main/java/org/egothor/stemmer/CompiledPatchCommand.java`.

Implementation corroborations are provided by `FrequencyTrieTest.java` and `FrequencyTrieProperties.java`. The publication package records source-file blob identities and a theorem-to-source correspondence so that a future revision can detect semantic drift.

No multilingual structural benchmark is required to regenerate the article’s principal results because the main claims are mathematical consequences of the frozen algorithm. The synthetic size-reduction fixture and property tests are cited

only as regression evidence. The article deliberately does not report an unreduced-versus-reduced multilingual percentage or a runtime speedup attributable to reduction.

XII. CONCLUSION

A learned stemming model is not fully characterized by the transformations it induces; the compiler determines which distinctions survive into runtime representation and where observed behavior becomes a generalized rule. Radixor makes that boundary explicit.

Mode-specific subtree canonicalization computes a recursive semantic signature and shares equal-signature subgraphs. We proved that signature equality is a congruence for the declared ranked, unordered, or preferred-value observable and that the count aggregation performed by the real compiler cannot invalidate the ranked or qualified-dominant descriptors used to authorize a merge. For productive nodes in contraction-free ranked and unordered runs, signatures are also complete for their continuation observables, giving a state-minimal canonicalization-only quotient among quotientings of the fixed node-output representation. Accepting-leaf generalization deliberately changes that setting, so the production interleaved graph is not claimed to be globally behavior-minimal. Dominant mode is intentionally more conservative. Equal signatures preserve subtree height, so quotienting cannot introduce cycles; finite compilation terminates and its equivalence tests are decidable. The reference Java implementation is semantically correct but not asymptotically identical to classical linear acyclic minimizers: recursive uncached signature hashing admits quadratic work on a degenerate chain.

Uniform-subtree contraction is a separate operation. Replacing a uniformly single-valued region by an accepting leaf preserves the ranked singleton sequence, candidate membership, and preferred command everywhere those value observables were already defined, but extends that command-selection result to all remaining continuations. In a backward suffix trie, the accepting leaf is a suffix-conditioned command rule: previously unseen words in the covered cylinder select the same transformation program c , while their stems remain $\text{Exec}(c, w)$ and therefore depend on the full input. This is controlled extension of a compiler decision, not all-string exact equivalence and not proof of linguistic correctness.

The separation resolves an ambiguity that is easy to miss when both mechanisms are called “reduction.” Canonicalization computes a quotient under a selected semantic contract; accepting-leaf contraction enlarges the domain over which one uniform command decision is applied. Treating them separately yields a stronger theory of compiled patch-command stemming: it identifies exactly when state sharing is sound and relatively minimal, where the concrete implementation pays additional algorithmic cost, and how finite lexical evidence becomes a reusable transformation decision whose linguistic quality can be evaluated independently.

REFERENCES

- [1] L. Galambos, “Lemmatizer for document information retrieval systems in JAVA,” in *SOFSEM 2001: Theory and Practice of Informatics*, LNCS 2234. Springer, 2001, pp. 243–252, doi: 10.1007/3-540-45627-9_21.

- [2] L. Galambos, "Semi-automatic stemmer evaluation," in *Intelligent Information Processing and Web Mining*. Springer, 2004, pp. 209–218, doi: 10.1007/978-3-540-39985-8_22.
- [3] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980, doi: 10.1108/eb046814.
- [4] A. Nerode, "Linear automaton transformations," *Proceedings of the American Mathematical Society*, vol. 9, no. 4, pp. 541–544, 1958, doi: 10.1090/S0002-9939-1958-0135681-9.
- [5] J. B. Lovins, "Development of a stemming algorithm," *Mechanical Translation and Computational Linguistics*, vol. 11, nos. 1–2, pp. 22–31, 1968.
- [6] R. Krovetz, "Viewing morphology as an inference process," in *Proc. 16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 1993, pp. 191–202, doi: 10.1145/160688.160718.
- [7] R. Krovetz, "Viewing morphology as an inference process," *Artificial Intelligence*, vol. 118, nos. 1–2, pp. 277–294, 2000, doi: 10.1016/S0004-3702(99)00101-0.
- [8] J. E. Hopcroft, "An $n \log n$ algorithm for minimizing states in a finite automaton," in *Theory of Machines and Computations*. Academic Press, 1971, pp. 189–196, doi: 10.1016/B978-0-12-417750-5.50022-1.
- [9] D. Revuz, "Minimisation of acyclic deterministic automata in linear time," *Theoretical Computer Science*, vol. 92, no. 1, pp. 181–189, 1992, doi: 10.1016/0304-3975(92)90142-3.
- [10] J. Daciuk, B. W. Watson, and R. E. Watson, "Incremental construction of minimal acyclic finite state automata and transducers," in *Finite State Methods in Natural Language Processing*, 1998, pp. 48–56, doi: 10.3115/1611533.1611538.
- [11] M. Mohri, "Minimization algorithms for sequential transducers," *Theoretical Computer Science*, vol. 234, nos. 1–2, pp. 177–201, 2000, doi: 10.1016/S0304-3975(98)00115-7.
- [12] C. Choffrut, "Minimizing subsequential transducers: A survey," *Theoretical Computer Science*, vol. 292, no. 1, pp. 131–143, 2003, doi: 10.1016/S0304-3975(01)00219-5.
- [13] J. Daciuk, S. Mihov, B. W. Watson, and R. E. Watson, "Incremental construction of minimal acyclic finite-state automata," *Computational Linguistics*, vol. 26, no. 1, pp. 3–16, 2000, doi: 10.1162/089120100561601.
- [14] M. Mohri, "Finite-state transducers in language and speech processing," *Computational Linguistics*, vol. 23, no. 2, pp. 269–311, 1997.
- [15] K. Koskenniemi, "Two-level model for morphological analysis," in *Proc. 8th Int. Joint Conf. Artificial Intelligence*, 1983.
- [16] K. R. Beesley and L. Karttunen, *Finite-State Morphology*. Stanford, CA, USA: CSLI Publications, 2003, ISBN 978-1-57586-434-1.
- [17] C. D. Paice, "Method for evaluation of stemming algorithms based on error counting," *Journal of the American Society for Information Science*, vol. 47, no. 8, pp. 632–649, 1996, doi: 10.1002/(SICI)1097-4571(199608)47:8<632::AID-ASI8>3.0.CO;2-U.
- [18] D. A. Hull, "Stemming algorithms: A case study for detailed evaluation," *Journal of the American Society for Information Science*, vol. 47, no. 1, pp. 70–84, 1996, doi: 10.1002/(SICI)1097-4571(199601)47:1<70::AID-ASI7>3.0.CO;2-#.
- [19] L. Galambos, "Radixor: Multilingual patch-command stemmer," GitHub repository, commit 0c3b13f485a9ad0b460c0931e4497ea95bed66a1, 2026. [Online]. Available: <https://github.com/leogalambos/Radixor/tree/0c3b13f485a9ad0b460c0931e4497ea95bed66a1>