Comonadic Morphophonology: A Compositional Framework for Context-Dependent Morphological Rules in Finnish

Yongseok Jang[*]
Independent Researcher
yongsk0066@gmail.com

Abstract

Composing finite-state transducers (FSTs) for context-dependent morphophonological rules—consonant gradation, vowel harmony, possessive suffix assimilation—leads to multiplicative state explosion; neural models sidestep the problem but provide no formal account of the rules themselves. We present the first framework where each morphophonological rule is a function from a focused local context to a single output segment—the type of a local rule familiar from cellular automata—and where length-changing rules compose as coKleisli arrows of a comonad. Our central contribution is the *Writer comonad* (DeletionSet $\times$ Zipper), a new algebraic construction that restores strict coKleisli compositionality for such rules: each rule is a coKleisli arrow, extend lifts it to a global transformation, and deletions accumulate as a monoid action rather than requiring intermediate materialization. As supporting evidence, thirteen coKleisli arrows provide an alternative formulation expressing the same morphophonological behaviors that Omorfi encodes via 874 continuation classes (67:1 reduction at the rule-representation level), and the same abstraction enables bidirectional morphology—a MorphGenerator reuses the analysis arrows for generation. On UD Finnish-TDT, the system achieves 83.92% UPOS accuracy with rule-only disambiguation (94.66% with an external suffix tagger), validating the framework as a practical morphological engine.

1 Introduction

A standard FST treats a rule as a string-to-string relation. We instead treat each rule as a function from a focused local context to a single output segment—precisely the type of a local rule in cellular automata. Composing such arrows is not function composition: the output of one rule has type Σ, not $\mathsf{Zipper}\langle\Sigma\rangle$. The coKleisli construction over a Zipper comonad fills exactly this typing gap.

We show that morphophonological rules—including length-changing operations such as consonant deletion—are coKleisli arrows of a Writer comonad, yielding the first framework that handles length-changing morphophonological rules within a compositional category-theoretic structure. Thirteen coKleisli arrows provide an alternative formulation expressing the same morphophonological behaviors that Omorfi encodes via 874 continuation classes (67:1 reduction at the rule-representation level via orthogonal composition), and the same abstraction unifies character-level, morpheme-level, and sentence-level processing.

Finnish morphophonology motivates the framework. *Consonant gradation* alternates stem-final consonants between strong and weak grades depending on syllable structure; *vowel harmony* requires suffix vowels to agree in backness with stem vowels; and *possessive suffix vowel copying* inserts a vowel identical to the preceding segment at morphological boundaries. Rule-based systems such as Voikko (Pitkänen, 2006) and Omorfi (Pirinen, 2015) encode these alternations as FST cascades (Koskenniemi, 1983; Beesley and Karttunen, 2003), achieving excellent coverage but limited compositionality: composing two FSTs yields a product automaton whose state space grows multiplicatively. Neural approaches (Kanerva et al., 2018; Nguyen et al., 2021) achieve high accuracy but provide no formal account of the underlying processes. No existing framework treats morphophonological rules as first-class compositional objects with formal combination laws.

Our goal is not to re-derive known rules, but to provide a *composition algebra* for rules that FST cascades handle only through state-space multiplication. The Writer comonad—which tracks deletions as a monoid action, deferring materialization to pipeline end—is a new algebraic construction

[*]Claude (Anthropic) was used for LaTeX polishing and code-writing assistance; the author takes full responsibility for the content.

Proceedings of the Society for Computation in Linguistics (SCiL) 2026, pages 100–112
July 3-4, 2026 ©2026 Association for Computational Linguistics

with no prior analogue in computational morphology. In contrast to FST ε-transitions, which require recomposition after each length-changing rule, the Writer comonad accumulates deletions within the coKleisli pipeline, preserving strict associativity without intermediate re-materialization.

Morphophonological rules are naturally *coKleisli arrows*: functions $W\,a \to b$ where W is a comonad providing a context window around a focused element. The comonad operation extend lifts such a local rule into a global transformation, and coKleisli composition ($\gg\!=\!\gg$) chains rules with associativity and identity guarantees. This connection is not accidental: Capobianco and Uustalu (2010) established that cellular automaton local behavior is precisely a coKleisli map of the exponent comonad, and morphophonological rule application instantiates exactly this pattern.

Our primary contribution is the Writer comonad (DeletionSet $\times$ Zipper), which restores strict coKleisli compositionality for length-changing morphophonological rules—a problem that FST cascades leave unresolved without multiplicative state explosion (§3, §3.6). As supporting demonstrations:

- The same comonadic abstraction operates at multiple linguistic levels—character (gradation), morpheme (harmony), and sentence (Constraint Grammar (CG) lite disambiguation)—via the uniform Zipper/extend mechanism (§3.5).
- The coKleisli pipeline enables bidirectional morphology: a MorphGenerator reuses the same arrows for both analysis and generation (§3.4).

We evaluate on UD Finnish-TDT, reporting 83.92% UPOS with rule-only disambiguation (94.66% with a suffix tagger), per-rule latency microbenchmarks, and a 67:1 reduction at the rule-representation level (§5).

2 Background

2.1 Finnish Morphophonology

Consonant gradation (*astevaihtelu*) alternates stem-final consonants between a *strong grade* and a *weak grade* (Hakulinen et al., 2004). The grade is determined by syllable structure: open syllables trigger the strong grade, closed syllables the weak grade. The alternation affects stops and certain consonant clusters according to the patterns in Table 1.

#	Strong	Weak	Type	Example
1	pp	p	Quant.	kaappi → kaapi
2	tt	t	Quant.	matto → mato
3	kk	k	Quant.	kukka → kuka
4	p	v	Qual. (s)	tupa → tuva
5	t	d	Qual. (s)	katu → kadu
6	k	∅	Qual. (s)	puku → puu
7	mp	mm	Qual. (c)	kampa → kamma
8	lt	ll	Qual. (c)	kulta → kulla
9	nt	nn	Qual. (c)	ranta → ranna
10	rt	rr	Qual. (c)	parta → parra
11	nk	ng	Qual. (c)	kenkä → kengä

Table 1: The 11 Finnish consonant gradation patterns (KOTUS types). Quant. = quantitative (geminate reduction), Qual. = qualitative (s = single, c = cluster). CoKleisli arrows examine only the immediate left neighbor, consistent with input strictly local (ISL) functions (Chandlee, 2014).

Vowel harmony (*vokaalisointu*) divides Finnish vowels into back (*a, o, u*), front (*ä, ö, y*), and neutral (*e, i*). Suffixes with archiphonemes A, O, U are realized as back or front variants depending on the stem. Vowel harmony exhibits TSL-2 locality (Heinz, 2018): the harmony arrow scans leftward through transparent neutral vowels, depending only on the nearest non-neutral vowel on the vowel tier (a window of size 2 on that tier). The Zipper's full left context naturally supports this tier projection: the harmony arrow scans leftward past neutral vowels until a non-neutral vowel is found, implementing this tier-based locality within the standard coKleisli mechanism.

Possessive suffix vowel copying. The archiphoneme *V* is realized as a copy of the immediately preceding vowel: the third-person possessive suffix *-Vn* becomes *-an* after *a*, *-en* after *e*, and analogously for the remaining vowels.

2.2 Comonads in Computer Science

A *comonad* is dual to a monad: where monads abstract over computations that produce effects, comonads abstract over computations that consume context (Orchard and Mycroft, 2012). We use the extract/extend formulation:

Definition 1 (Comonad)**.** *A comonad* $(W, \mathsf{extract}, \mathsf{extend})$ *consists of an endofunctor* W *with* $\mathsf{extract} : W\,a \to a$ *and* $\mathsf{extend} : (W\,a \to b) \to W\,a \to W\,b$*, satisfying:*

$$\mathsf{extend}\ \mathsf{extract} = \mathrm{id} \tag{L1′}$$

$$\mathsf{extract} \circ \mathsf{extend}\ f = f \tag{L2′}$$

$$\mathsf{extend}\ g \circ \mathsf{extend}\ f = \mathsf{extend}\ (g \circ \mathsf{extend}\ f) \tag{L3′}$$

Definition 2 (CoKleisli arrow)**.** *A function* $f : W\ a \to b$ *is a* coKleisli arrow. *CoKleisli arrows compose via* $(f \gg\!= g)(w) = g(\mathsf{extend}\ f\ w)$*, forming the* coKleisli category *of* W *(associative, with* extract *as identity).*

The ListZipper comonad. The ListZipper (Huet, 1997) represents a non-empty sequence with a distinguished *focus* and left/right context (Figure 1). extract returns the focused element; extend f applies f at every position, each seeing the full context from that position's perspective. Uustalu and Vene (2005) showed that this captures the essence of applying a local update rule globally—precisely a cellular automaton. Our insight is that Finnish morphophonological rules instantiate the same pattern.

3 Formalization

3.1 Consonant Gradation as CoKleisli Arrows

We encode each consonant gradation pattern as a two-character window (c_0, c_1), where c_0 is the left context and c_1 the focused character. The coKleisli arrow has the signature:

$$\mathsf{apply\_gradation} : \mathsf{Zipper}(\mathit{char}) \times \mathit{Grade} \to \mathit{char} \tag{1}$$

where $\mathit{Grade} \in \{\mathsf{Strong}, \mathsf{Weak}\}$. Partially applying the grade yields a coKleisli arrow $\mathsf{Zipper}(\mathit{char}) \to \mathit{char}$ suitable for extend.

Design principle: only position 1 is transformed. The coKleisli arrow modifies only the focused character (position 1 of the pattern window) and treats the left neighbor (position 0) as read-only context. This is essential because extend applies the function at *every* position: if both characters were eligible, geminate $pp \to p$ would produce double-counting. By transforming only position 1, each position produces exactly one output, consistent with comonadic extend semantics. This mirrors cellular automata, where each cell reads neighbors but writes only its own state.

The deletion marker. Pattern 6 (intervocalic *k* deletion) presents a type-theoretic challenge: the coKleisli arrow has type $\mathsf{Zipper}(\mathit{char}) \to \mathit{char}$, requiring each position to produce exactly one character. A naïve approach inserts a null character `'\0'` as a deletion marker, but this breaks strict coKleisli associativity because filtering between steps requires materializing and re-creating zippers. We resolve this algebraically via the Writer comonad (§3.6).

Pattern priority and suppression. Priority is determined by specificity: geminates match first, then clusters, then single consonants. A suppression predicate $\mathsf{is\_pos0}(c, r)$ returns true when the focused character c and its right neighbor r form position 0 of a higher-priority pattern (e.g., $\mathsf{is\_pos0}(\mathtt{p}, \mathtt{p})$ = true for the geminate pp), preventing single-consonant rules from firing erroneously. Formally:

$$\mathsf{grad}(z, W) = \begin{cases} c & \mathsf{is\_pos0}(c, r) \\ \mathsf{tgt}(p, W)[1] & \mathsf{find}(l, c, r, W) = \mathsf{Some}(p) \\ c & \text{otherwise} \end{cases} \tag{2}$$

3.2 Worked Example: Weakening "kaappi"

We trace $\mathsf{extend}(\mathsf{apply\_gradation}(\cdot, \mathsf{Weak}))$ on *kaappi* in Figure 3. The key interactions are: (1) at position 3, the first p is *suppressed* because $\mathsf{is\_pos0}(\mathtt{p}, \mathtt{p})$ recognizes it as position 0 of the geminate pattern—without suppression, the single-consonant rule would fire, yielding incorrect **kaavi*; (2) at position 4, the geminate pattern matches and the Writer comonad arrow returns $(\{4\}, \mathtt{p})$, marking the position for deletion; (3) materialize applies the accumulated $\mathit{DeletionSet} = \{4\}$, yielding *kaapi* (weak grade).

3.3 Vowel Harmony as a CoKleisli Arrow

Vowel harmony is a coKleisli arrow $\mathsf{harmony} : \mathsf{Zipper}(\mathit{char}) \to \mathit{char}$ resolving archiphonemes by harmony class:

$$\mathsf{detect}(z) = \begin{cases} \mathsf{Back} & \text{back vowel left} \\ \mathsf{Front} & \text{front vowel left} \\ \mathsf{Front} & \text{neutral only} \end{cases} \tag{3}$$

$$\mathsf{harmony}(z) = \begin{cases} a/\text{ä} & \mathsf{extract}(z) = \mathtt{A} \\ o/\text{ö} & \mathsf{extract}(z) = \mathtt{O} \\ u/y & \mathsf{extract}(z) = \mathtt{U} \\ c & \text{otherwise} \end{cases} \tag{4}$$

Example 1. *talo* + *-ssA* → t-a-l-o-s-s-a (back vowel *a* found left). Conversely, *kynä* + *-ssA* → k-y-n-ä-s-s-ä (front vowels *y, ä*).

3.4 Pipeline Composition and Bidirectional Morphology

The morphophonological pipeline chains three coKleisli arrows (Figure 2):

$$\mathsf{pipe} = \mathsf{grad} \gg\!= \mathsf{harmony} \gg\!= \mathsf{poss} \tag{5}$$

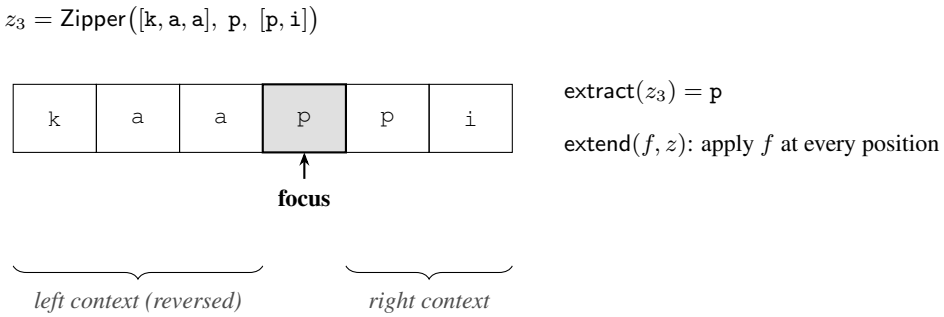

Figure 1: The ListZipper data structure for the word *kaappi* focused on position 3. The left context is stored in reversed order (nearest neighbor last) for $O(1)$ access. extract returns the focused element; $\mathsf{extend}(f, z)$ applies a coKleisli arrow f at every position, each seeing the full context from that position's perspective.

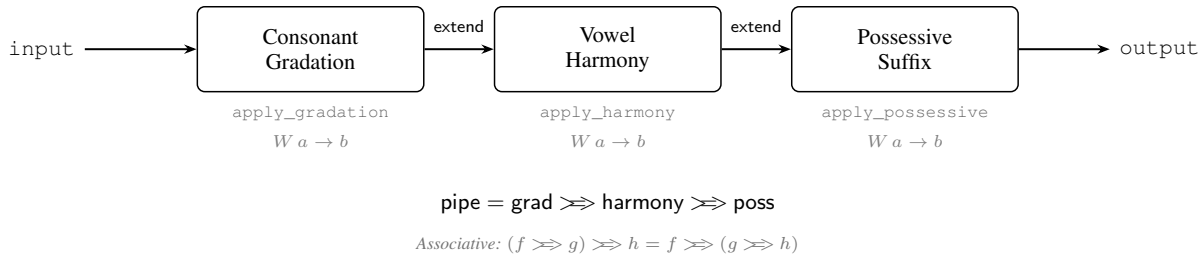

Figure 2: Morphophonological pipeline as coKleisli composition. Each box is a coKleisli arrow ($W\ a \to b$) lifted globally by extend. CoKleisli composition is associative with extract as identity.

A naïve implementation inserts a `'\0'`-filtering step between gradation and harmony, which breaks strict associativity. The Writer comonad (§3.6) resolves this: each arrow returns $(DeletionSet, char)$, accumulated deletions are merged via monoid union during extend, and a single materialize step applies all deletions at pipeline end. This restores strict coKleisli associativity across the full pipeline, including deletion.

Example 2 (Pipeline: "kampAstAVn" (weak grade)). (1) Gradation: `mp` → `mm` ⇒ `kammAstAVn`. (2) Harmony: `A` → `a` ⇒ `kammastaVn`. (3) Possessive: `V` → `a` ⇒ `kammastaan` ('from his/her comb').

MorphGenerator: bidirectional morphology. The coKleisli pipeline enables bidirectional morphology: analysis (via Voikko FST, hereafter VFST) and generation (via coKleisli arrows) share the same rules. The MorphGenerator produces inflected forms from baseform + features by applying coKleisli arrows in the generative direction (currently 11 noun cases). Since each rule is an explicit function rather than a compiled FST state, the same logic drives both analysis and generation with minimal additional code.

Rule ordering. Gradation precedes harmony because gradation may delete characters, changing the vowel context. The coKleisli framework does not eliminate order-dependence—it formalizes it: each rule remains individually addressable and testable, and the composition order is explicit rather than implicit in merged automaton states.

3.5 CG Rules as Comonadic Extension

The comonadic framework extends to sentence-level disambiguation: only the Zipper's element type changes (from character to ReadingSet), while the coKleisli composition structure is identical (the deletion-handling Writer comonad needed for character-level rules is formalized in §3.6). In CG-lite, each rule is a coKleisli arrow over *reading sets*:

$$rule : \mathsf{Zipper}(ReadingSet) \to ReadingSet \quad (6)$$

Twenty-four rule types are implemented, covering removal, selection, and structural patterns parameterized by part-of-speech (POS) tags, baseforms, and feature constraints. The current Finnish configuration contains 62 active rules (85 total; 23 disabled for accuracy after tuning). All rules enforce a safety invariant: a rule never removes the last reading at any position, ensuring well-definedness

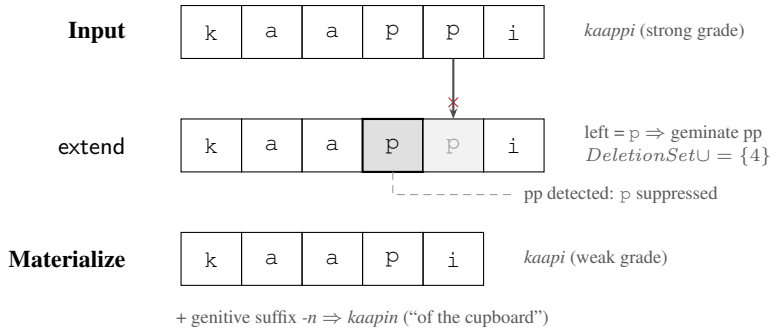

Figure 3: Step-by-step trace of consonant gradation on *kaappi* using the Writer comonad (positions are 0-indexed). The extend pass applies the gradation arrow at each position. At position 3, the first p is recognized as position 0 of the geminate pp pattern and **suppressed** (not transformed to v). At position 4, the second p matches position 1 of the geminate pattern; the arrow returns $(\{4\}, \mathrm{p})$, marking it for deletion via the $DeletionSet$ while preserving the character in the zipper. Materialize applies accumulated deletions once, yielding *kaapi* (weak grade stem).

Level	Zipper	Arrow	Out
Char	$\mathsf{Zip}\langle c\rangle$	Gradation	c
Morph	$\mathsf{Zip}\langle c\rangle$	Harmony	c
Sent	$\mathsf{Zip}\langle RS\rangle$	CG rule	RS

Table 2: Unified comonadic abstraction across three linguistic levels (c = char, RS = ReadingSet). Zipper provides context, the coKleisli arrow computes a local transformation, extend lifts it globally. These three levels are operationalized in MCE at: char/morph in the MorphGenerator (§3.4), sent in CG-lite disambiguation (§3.5).

of subsequent extend passes. The same coKleisli abstraction that governs character-level morphophonology applies at this sentence level—each CG rule is a coKleisli arrow over reading sets, composed via extend (details in Appendix B.3).

Multiple CG rules are applied by iterating extend:

$$\mathsf{cg}(s, [r_1, \ldots, r_k]) = \mathsf{vec}(\mathsf{ext}(r_k, \ldots \mathsf{ext}(r_1, \mathsf{Zip}(s)) \ldots)) \quad (7)$$

Note that sequential extension is equivalent to coKleisli composition: $\mathsf{extend}(r_2) \circ \mathsf{extend}(r_1) = \mathsf{extend}(r_1 \gg\!= r_2)$, so the CG pipeline uses the same composition mechanism as the morphophonological pipeline ((5)).

Unified comonadic abstraction. The same algebraic structure operates at three linguistic levels (Table 2):

To our knowledge, this is the first application of the comonad/cellular-automaton correspondence (Capobianco and Uustalu, 2010) simultaneously at multiple linguistic levels in computational morphology.

3.6 Writer Comonad for Deletion

The deletion marker approach (§3.1) would break strict coKleisli associativity: filtering `'\0'` characters between extend passes requires materializing intermediate sequences and reconstructing zippers, violating the comonad law $\mathsf{extend}\ g \circ \mathsf{extend}\ f = \mathsf{extend}\ (g \circ \mathsf{extend}\ f)$.

The key idea: instead of immediately deleting characters, we *accumulate* deletion positions alongside the computation, applying all deletions once at pipeline end. We formalize this with the *Writer comonad*[1] $DS \times \mathsf{Zipper}$, where $(DS, \cup, \emptyset)$ is the *deletion monoid*—a set of positions with union as the monoid operation and the empty set as identity.

Definition 3 (Writer Comonad)**.** *Let $(DS, \cup, \emptyset)$ be the deletion monoid. The* Writer comonad *is the product $DS \times \mathsf{Zipper}$ with:*

$$\mathsf{extract}_W(w, z) = \mathsf{extract}(z) \quad (8)$$
$$\mathsf{extend}_W(f)(w, z) = \Big(w \cup \textstyle\bigcup_i \pi_1(f(w, z_i)), \mathsf{fmap}(\pi_2 \circ f)(w, z)\Big)$$

where z_i ranges over all positions of z, π_1/π_2 are the product projections onto DS and the output character, and fmap *denotes the underlying Zipper's* extend *applied to the character component: concretely,* $\mathsf{fmap}(\pi_2 \circ f)(w, z) = \mathsf{Zipper.extend}(\lambda z_i.\ \pi_2(f(w, z_i)))(z)$.

The construction factorizes: the Zipper satisfies standard comonad laws on characters, while DS

[1]The name "Writer comonad" is by analogy with the Writer monad—both accumulate a monoid value across a computation—but the construction is not the categorical dual of Writer. Standard categorical taxonomy does not provide a canonical name for this structure; we retain "Writer comonad" for its descriptive value with this caveat.

satisfies the monoid laws independently; extend_W preserves both.

Concretely, each coKleisli arrow returns (DS, A) instead of A: gradation returns $(\{pos\}, c)$ when deleting a consonant and $(\emptyset, c')$ otherwise; harmony and possessive always return $(\emptyset, c')$ since they are non-deleting. The pipeline begins with $w = \emptyset$ (no prior deletions); subsequent arrows in the coKleisli composition receive the accumulated DS from all preceding arrows via extend_W, propagating deletion information through the pipeline without intermediate materialization. A single materialize step at pipeline end applies all accumulated deletions.

Proposition 1. *The Writer comonad construction* $(DS \times \mathsf{Zipper}, \mathsf{extract}_W, \mathsf{extend}_W)$ *satisfies the comonad laws on the character component and the monoid laws on DS, jointly ensuring compositional correctness of the pipeline.*

Proof sketch. **L1′** ($\mathsf{extend}_W\ \mathsf{extract}_W = \mathrm{id}$): For the deletion component, $\pi_1(\mathsf{extract}_W(w, z_i)) = \emptyset$ at every position, so $w \cup \bigcup_i \emptyset = w$ by monoid identity. For the character component, $\mathsf{Zipper.extend}(\lambda z_i.\ \pi_2(\mathsf{extract}_W(w, z_i)))(z) = \mathsf{Zipper.extend}(\mathsf{extract})(z) = z$ by the underlying Zipper's L1′. Thus $\mathsf{extend}_W(\mathsf{extract}_W)(w, z) = (w, z)$. **L2′** ($\mathsf{extract}_W \circ \mathsf{extend}_W\ f = \pi_2 \circ f$): $\mathsf{extract}_W(\mathsf{extend}_W(f)(w, z)) = \mathsf{extract}(\mathsf{Zipper.extend}(\lambda z_i.\ \pi_2(f(w, z_i)))(z)) = \pi_2(f(w, z))$ by the Zipper's L2′; the DS component is accumulated by extend_W and consumed by materialize. **L3′**: see Appendix A.6 for the full argument; the key step is that extend_W distributes over coKleisli composition because the DS components associate via the monoid law $(DS, \cup, \emptyset)$ and the character components associate via the underlying Zipper's L3′. All laws are verified by 44 unit tests covering the Zipper comonad laws (character component) and the DS monoid laws independently. □

The equivalence between the Writer pipeline and the old hybrid pipeline is verified on all 11 gradation patterns, confirming that the algebraic treatment of deletion introduces no behavioral difference.

4 Implementation

We implement the framework in the Morphological Computation Engine (MCE), a Rust (2024 edition, stable 1.86+) library targeting WebAssembly (∼380 KB). Since Rust lacks higher-kinded types, the comonad operations are implemented directly on the concrete `Zipper<T>` type rather than via a generic `Comonad` trait. Each morphophonological rule is a plain function `fn(&Zipper<char>) -> char`; CG rules use trait-based dispatch with the same coKleisli signature. The current extend allocates $O(n^2)$ due to cloning at each position, which is negligible for Finnish word lengths ($n \leq 15$). The implementation is verified through 317 tests (comonadic pipeline; 1,619 total across the system) covering Zipper comonad laws, *DeletionSet* monoid laws, all 11 gradation patterns in both directions, roundtrip properties, Writer pipeline equivalence, and CG rule correctness. Full implementation details, Rust code listings, and test methodology are provided in Appendix A.

5 Evaluation

5.1 Experimental Setup

We evaluate on UD Finnish-TDT v2.14 (Haverinen et al., 2014), distributed under CC-BY-SA 4.0, dev split (1,364 sentences, ∼25,000 tokens) with gold tokenization. Since our CG rules are hand-written and the suffix tagger was trained exclusively on the training split, the development set provides an unbiased evaluation. Accuracy is micro-averaged token-level accuracy; out-of-vocabulary (OOV) tokens (0.36%) default to NOUN. The morphological analyzer uses the Voikko Finnish morphological dictionary (voikko-fi v2.6, 39,607 lexical entries; `mor.vfst`, 4 MB). All experiments run on a single CPU core (Apple M2 Max); no GPU is required.

5.2 Morphophonological Correctness

All 11 consonant gradation patterns are correctly formalized as coKleisli arrows, verified by the Writer comonad pipeline. Of these, 7 exhibit full bidirectional roundtrip correctness ($\mathsf{strong}(\mathsf{weak}(w)) = w$); the 4 non-roundtrip cases all involve deletion, confirming deletion as the sole source of irreversibility. The Writer comonad laws hold as follows. L1′ (left identity) and L2′ (right identity) hold for arbitrary coKleisli arrows. L3′ (associativity) holds on the value (Zipper) component; the log component is treated as an accumulating output rather than a structural part of the comonad, and is materialized once at the end of each pipeline stage. In practice we invoke extend

Component	Mean (μs)	Std (μs)
CoKleisli (char-level)		
Gradation (avg/11)	0.61	0.24
Harmony (avg/4)	0.85	0.22
Possessive (avg/3)	0.70	0.24
Full pipeline (avg/4)	2.66	0.58
CG-lite (sentence-level)		
Single rule (avg/20)	7.66	1.34
Full CG (62 rules)	376.04	32.35

Table 3: Per-rule latency (10K iterations, Apple M-series). CoKleisli arrows: sub-μs; full pipeline: under 3 μs.

System	UPOS	Lemma	Approach
TNPP†	96.91	95.54	Neural
MCE-full (ours)	**94.66**	**93.09**	**FST+CG+stat**
Omorfi 1-best	83.88	82.63	FST + freq.
MCE-rules (ours)	83.92	93.09	FST+CG

Table 4: UPOS accuracy (%) on UD Finnish-TDT dev set with gold tokenization. Coverage: 99.35%. MCE-rules uses only comonadic CG disambiguation (62 active rules); MCE-full adds a lightweight suffix tagger (§5.4). †TNPP = Turku Neural Parser Pipeline. TNPP and Omorfi figures from Pirinen (2019), evaluated on an earlier UD Finnish-TDT version; POS annotation guidelines have remained stable across versions, so direct comparison is approximate but meaningful.

once per stage, so the question of log-component associativity under repeated extend does not arise in our pipeline; the Zipper alone satisfies all three comonad laws. Forty-four dedicated tests verify these properties. Full pattern-level results are in Appendix B.4.

5.3 System Performance

Throughput. The complete MCE pipeline processes **84,973 tokens/sec** on Apple M-series (∼12 μs/token, ∼0.21 ms for a 15-word sentence), demonstrating that coKleisli composition introduces no significant overhead compared to monolithic implementations.

Per-rule latency. Table 3 reports per-rule microbenchmarks (10K iterations). The full morphophonological pipeline averages 2.66 μs per word, confirming that coKleisli composition introduces negligible overhead beyond the sum of its components.

UPOS accuracy. With rule-only disambiguation (62 active CG rules), MCE achieves 83.92% UPOS and 93.09% lemma accuracy (Table 4) with 99.35% coverage. Adding a suffix tagger raises UPOS to 94.66%, narrowing the gap to TNPP from 12.99 pp to 2.25 pp.

Property	Comonadic	FST cascade
Composition	$O(k)$ functions	$O(\|S_1\| \times \|S_2\|)$
Modularity	Independent fns	Merged FST
Debugging	Per-rule tests	Composite trace
Deletion	Writer comonad	ε-transitions
Rule count	20 arrows‡	8,157 LEXICONs

Table 5: Comonadic vs. FST-based rule encoding. ‡20 total arrows: 13 morphophonological (11 gradation + harmony + possessive) + 7 system-level (tokenization, OOV fallback, case mapping, compound boundary, AUX resolution, coverage, Viterbi integration); at the morphophonological level, 13 vs. 874 (67:1).

Comparison with FST-based systems. Table 5 summarizes the key differences. Voikko's 8,157 LEXICON entries encode the full lexicon (stems, paradigms, compounds, *and* morphophonological rules), so the gross 408:1 ratio overstates the difference. At the morphophonological-rule-only level, Omorfi uses 874 continuation classes (810 harmony × suffix-class variants, 62 harmony × gradation, 2 gradation only)[2] vs. MCE's 13 arrows (67:1 at the rule-representation level). This comparison concerns rule-representation units: the MCE analysis pipeline itself uses a lexicon-based analyzer (Voikko-derived VFST) compatible with Omorfi-style approaches. The insight is *orthogonal composition*: FSTs require $O(P \times H \times C)$ entries per combination, while coKleisli composition is $O(P + H + C)$. The FST approach retains advantages within the same module: native ε-transitions, minimization, and direct encoding of lexical exceptions. Our framework operates in the SPE-era modular tradition (Chomsky and Halle, 1968), scoping itself to the rule layer and delegating lexical idiosyncrasies to the lexicon—in MCE, the Voikko-derived analyzer.

Developer ergonomics. Adding a new morphophonological rule requires defining a single coKleisli arrow (∼5 lines of Rust) that automatically composes with all existing rules. In contrast, adding a rule to an FST system requires modifying potentially hundreds of `LEXICON` entries to account for all interaction combinations—a direct consequence of the multiplicative vs. additive composition difference.

[2]Counted from Omorfi's `continuations.tsv` (commit `6ce67ac`).

Configuration	UPOS	Δ
FST only (no disambig.)	38.96	—
+ Fallback + coverage	∼70	+31
+ AUX/participle map	∼80	+10
+ Viterbi bigram	∼81.20	+1.2
+ Priors + CG-lite	83.92	+2.72
+ Suffix tagger	94.66	+10.74

Table 6: Ablation study (% UPOS, Δ in pp). Values prefixed with ∼ are approximate, measured during incremental development before the final pipeline was frozen. The suffix tagger is a separate statistical component beyond the comonadic pipeline.

5.4 Ablation and Error Analysis

Table 6 shows each component's contribution. The CG-lite rules target top confusion pairs (NOUN/VERB, ADJ/NOUN, ADV/NOUN, PRON/NOUN), each providing deterministic, explainable disambiguation. Structural fixes (+31 pp coverage, +10 pp mapping) contribute more than corpus bigrams (+1.2 pp).

An optional suffix tagger (logistic regression, 157K features, 5 MB) adds 10.74 pp by resolving ambiguities requiring distributional evidence—primarily NOUN/VERB pairs (*tuuli* 'wind' vs. 'blew'), contextual ADJ/NOUN distinctions, and verbal forms following negation verbs where CG rules lack sufficient context. It operates *after* the comonadic pipeline using CG-filtered candidates; it is a separate statistical module. Detailed CG examples are in Appendix B.3.

6 Discussion

The Cofree comonad could provide incremental re-analysis for interactive spell-checking, and the comonad/monad adjunction $W \dashv M$ suggests a connection between comonadic (context-consuming) morphophonology and monadic (effect-producing) generation. We conjecture that every ISL-k function over strings can be expressed as a coKleisli arrow with window size at most $k{+}1$, establishing a formal bridge between subregular phonology (Chandlee and Heinz, 2018) and comonadic composition. The intuition is direct: an ISL-k function's output at each position depends on at most k contiguous input symbols; a coKleisli arrow over a Zipper of the same string has access to the full context but need only examine $k{-}1$ neighbors, matching the ISL sliding-window semantics; `extend` then lifts this local computation to all positions simultaneously. A formal proof requires careful treatment of boundary conditions (string-initial and string-final positions where the Zipper context is truncated) and is our primary direction for future work. We further conjecture that the framework extends to other agglutinative languages: Turkish 4-way vowel harmony, Hungarian ternary harmony, and Estonian consonant gradation all exhibit local context-dependent alternation expressible as coKleisli arrows. The framework should also extend to other length-preserving phenomena: Korean consonant assimilation, Japanese rendaku, North Sámi consonant gradation. Templatic morphology (Semitic root-and-pattern: Arabic K-T-B yields *kataba*, *kitaab*, *maktab*) and reduplication (Indonesian *orang-orang*, Tagalog *bi-bili*) require segment insertion or copying and are not directly captured by the present framework; we develop an extension using an InsertionMap monoid in follow-up work. Verification requires language-specific implementation, which we leave to future work.

7 Related Work

Finite-state morphology. Koskenniemi (1983) established the finite-state paradigm, formalized by Kaplan and Kay (1994) and systematized by Beesley and Karttunen (2003); Hulden (2009) and HFST (Lindén et al., 2013) provide efficient open-source FST tooling. Our contribution is not a faster runtime but a *composition algebra* that keeps rules individually addressable after composition—something FST intersection does not preserve. Our framework operates in the SPE tradition (Chomsky and Halle, 1968), orthogonal to OT's constraint ranking (Prince and Smolensky, 2004).

Functional Morphology. Forsberg and Ranta (2004) treat morphological paradigms as Haskell functions, achieving paradigm compression through functional abstraction. Our framework provides a specific structural guarantee that general functional composition lacks: `extend` uniformly lifts a local rule to all positions with automatic context threading. Forsberg and Ranta (2004) focus on paradigm *generation*; our framework addresses *application* of context-dependent rules, though the MorphGenerator demonstrates that generation also fits naturally.

Neural morphological transducers. Our work targets a different design point—interpretable, offline morphological processing—and the distinct

error profiles of neural vs. rule-based systems suggest hybrid potential (Pirinen, 2019). Recent neural transducers achieve strong results across diverse typologies (see Baxi and Bhatt, 2024, for a survey); SIGMORPHON shared tasks (Kodner et al., 2022; Goldman et al., 2023; Batsuren et al., 2022) continue to drive progress. Combining comonadic rules with neural disambiguation is a natural direction.

Subregular hierarchy. Finnish consonant gradation is input strictly local (ISL); vowel harmony exhibits TSL-2 locality (Heinz, 2018; Chandlee, 2014). Our gradation arrows examine only the immediate left neighbor (ISL); the harmony arrow scans through transparent vowels (matching TSL-2 locality on the vowel tier). The subregular classification could inform arrow design for other languages.

Category theory in NLP. de Felice (2022) applies monoidal categories to compositional semantics; our contribution is orthogonal: *comonadic* structure for context-dependent local transformation.

8 Conclusion

We have shown that the Writer comonad (DeletionSet $\times$ Zipper) restores strict coKleisli compositionality for length-changing morphophonological rules, resolving algebraically a problem that FST cascades address only through multiplicative state explosion—to our knowledge, the first application of the comonad/cellular-automaton correspondence (Capobianco and Uustalu, 2010) to natural language morphophonology.

Several directions for future work emerge: extending the framework to other agglutinative languages (§6); enlarging the coKleisli arrow vocabulary to cover epenthesis and metathesis for a more complete algebraic classification; combining comonadic rules with neural disambiguation to exploit complementary error profiles (Pirinen, 2019); and using the Cofree comonad for incremental re-analysis in interactive spell-checking.

Code and data. The MCE implementation (Rust, Apache-2.0) and evaluation scripts will be released at `https://github.com/yongsk0066/mce` upon publication, with a version tag corresponding to the results reported here. UD Finnish-TDT is distributed under CC-BY-SA 4.0. The Voikko morphological dictionary (voikko-fi) is distributed under GPL-2.0-or-later.

Limitations

The Zipper$\langle char \rangle$ does not encode morpheme boundaries; a Zipper$\langle Morpheme \rangle$ would address this but requires a prior segmentation step. Consonant deletion requires the Writer comonad (§3.6), verified by 44 tests. The 24 CG rule types do not cover the full CG specification's BARRIER conditions and MAP/SUBSTITUTE operations (Karlsson, 1990; Bick, 2000); extending the rule vocabulary is straightforward within the coKleisli framework. The rule-only 83.92% is 12.99 pp below neural TNPP (2.25 pp with the suffix tagger); the gap reflects deep syntactic disambiguation—not a limitation of the comonadic framework itself.

References

Khuyagbaatar Batsuren, Gábor Bella, Aryaman Arora, Viktor Martinovic, Kyle Gorman, Zdeněk Žabokrtský, Amarsanaa Ganbold, Šárka Dohnalová, Magda Ševčíková, Kateřina Pelegrinová, Fausto Giunchiglia, Ryan Cotterell, and Ekaterina Vylomova. 2022. The SIGMORPHON 2022 shared task on morpheme segmentation. In *Proceedings of the 19th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, pages 103–116.

Jatayu Baxi and Brijesh Bhatt. 2024. Recent advancements in computational morphology: A comprehensive survey. *arXiv preprint arXiv:2406.05424*.

Kenneth R. Beesley and Lauri Karttunen. 2003. *Finite State Morphology*. CSLI Publications, Stanford, CA.

Eckhard Bick. 2000. *The Parsing System "Palavras": Automatic Grammatical Analysis of Portuguese in a Constraint Grammar Framework*. Aarhus University Press.

Silvio Capobianco and Tarmo Uustalu. 2010. A categorical outlook on cellular automata. In *Journées Automates Cellulaires (JAC 2010)*, volume 13 of *TUCS Lecture Notes*, pages 88–99. Turku Centre for Computer Science.

Jane Chandlee. 2014. *Strictly Local Phonological Processes*. Ph.D. thesis, University of Delaware.

Jane Chandlee and Jeffrey Heinz. 2018. Strict locality and phonological maps. *Linguistic Inquiry*, 49(1):23–60.

Noam Chomsky and Morris Halle. 1968. *The Sound Pattern of English*. Harper & Row, New York.

Giovanni de Felice. 2022. *Categorical Tools for Natural Language Processing*. Ph.D. thesis, University of Oxford.

Markus Forsberg and Aarne Ranta. 2004. Functional morphology. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming (ICFP 2004)*, pages 213–223, Snowbird, Utah. ACM.

Omer Goldman, Khuyagbaatar Batsuren, Salam Khalifa, Aryaman Arora, Garrett Nicolai, Reut Tsarfaty, and Ekaterina Vylomova. 2023. SIGMORPHON–UniMorph 2023 shared task 0: Typologically diverse morphological inflection. In *Proceedings of the 20th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, pages 117–125.

Auli Hakulinen, Maria Vilkuna, Riitta Korhonen, Vesa Koivisto, Tarja Riitta Heinonen, and Irja Alho. 2004. *Iso suomen kielioppi*. Suomalaisen Kirjallisuuden Seura, Helsinki. Online edition (2008): http://scripta.kotus.fi/visk. URN:ISBN:978-952-5446-35-7.

Katri Haverinen, Jenna Nyblom, Timo Viljanen, Veronika Laippala, Samuel Kohonen, Anna Missilä, Stina Ojala, Tapio Salakoski, and Filip Ginter. 2014. Building the essential resources for Finnish: The Turku Dependency Treebank. *Language Resources and Evaluation*, 48(3):493–531.

Jeffrey Heinz. 2018. The computational nature of phonological generalizations. In Larry M. Hyman and Frans Plank, editors, *Phonological Typology*, pages 126–195. De Gruyter Mouton.

Gérard Huet. 1997. The zipper. *Journal of Functional Programming*, 7(5):549–554.

Mans Hulden. 2009. Foma: A finite-state compiler and library. In *Proceedings of the 12th Conference of the European Chapter of the ACL: Demonstrations (EACL 2009)*, pages 29–32.

Jenna Kanerva, Filip Ginter, Niko Miekka, Akseli Leino, and Tapio Salakoski. 2018. Turku Neural Parser Pipeline: An end-to-end system for the CoNLL 2018 shared task. In *Proceedings of the CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 133–142, Brussels, Belgium. Association for Computational Linguistics.

Ronald M. Kaplan and Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378.

Fred Karlsson. 1990. Constraint grammar as a framework for parsing running text. In *Proceedings of the 13th International Conference on Computational Linguistics (COLING 1990)*.

Jordan Kodner, Salam Khalifa, Khuyagbaatar Batsuren, Hossep Dolatian, Ryan Cotterell, Faruk Akkus, Antonios Anastasopoulos, Taras Andrushko, Aryaman Arora, Nona Atanalov, Gábor Bella, Elena Budianskaya, Yustinus Ghanggo Ate, Omer Goldman, David Guriel, Simon Guriel, Silvia Guriel-Agiashvili, Witold Kieras, Andrew Krizhanovsky, and 11 others. 2022. SIGMORPHON–UniMorph 2022 shared task 0: Generalization and typologically diverse morphological inflection. In *Proceedings of the 19th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, pages 176–203.

Kimmo Koskenniemi. 1983. *Two-Level Morphology: A General Computational Model for Word-Form Recognition and Production*. Ph.D. thesis, University of Helsinki. Publication No. 11, Department of General Linguistics.

Krister Lindén, Erik Axelson, Senka Drobac, Sam Hardwick, Juha Kuokkala, Jyrki Niemi, Tommi A. Pirinen, and Miikka Silfverberg. 2013. HFST—a system for creating NLP tools. In *Systems and Frameworks for Computational Morphology*, volume 380 of *Communications in Computer and Information Science*, pages 53–71. Springer.

Minh Van Nguyen, Viet Dac Lai, Amir Pouran Ben Veyseh, and Thien Huu Nguyen. 2021. Trankit: A light-weight Transformer-based toolkit for multilingual natural language processing. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: System Demonstrations (EACL 2021)*, pages 80–90. Association for Computational Linguistics.

Dominic Orchard and Alan Mycroft. 2012. A notation for comonads. In *Implementation and Application of Functional Languages (IFL 2012)*, volume 8241 of *LNCS*, pages 1–17. Springer.

Tommi A. Pirinen. 2015. Omorfi—free and open source morphological lexical database for Finnish. In *Proceedings of the 20th Nordic Conference of Computational Linguistics (NODALIDA 2015)*, pages 313–315. Linköping University Electronic Press.

Tommi A. Pirinen. 2019. Neural and rule-based Finnish NLP models—expectations, experiments and experiences. In *Proceedings of the Fifth International Workshop on Computational Linguistics for Uralic Languages*, pages 104–114, Tartu, Estonia. Association for Computational Linguistics.

Harri Pitkänen. 2006. Hunspell-fi in Kesäkoodi 2006: Final report. Technical report, Finnish Centre for Open Source Solutions (COSS). Available at https://www.puimula.org/htp/archive/kesakoodi2006-report.pdf.

Alan Prince and Paul Smolensky. 2004. *Optimality Theory: Constraint Interaction in Generative Grammar*. Blackwell, Malden, MA.

Tarmo Uustalu and Varmo Vene. 2005. The essence of dataflow programming. In *Programming Languages and Systems: Third Asian Symposium (APLAS 2005)*, volume 3780 of *LNCS*, pages 2–18. Springer.

A Implementation Details

A.1 Rust and the Absence of Higher-Kinded Types

The formalization in Section 3 is presented in terms of Haskell-style type classes, where a generic `Comonad` class abstracts over any functor W. Rust lacks higher-kinded types (HKTs), so a direct translation of `class Comonad w` is not possible. We therefore adopt a pragmatic approach: the comonad operations are implemented directly on the concrete `Zipper<T>` type, without a generic `Comonad` trait.

```
pub struct Zipper<T> {
    left:  Vec<T>,  // reversed
    focus: T,
    right: Vec<T>,
}

impl<T: Clone> Zipper<T> {
    pub fn extract(&self) -> &T {
        &self.focus
    }

    pub fn extend<U, F>(&self, f: F)
        -> Zipper<U>
    where F: Fn(&Zipper<T>) -> U {
        let focus = f(self);
        let left  = Lefts::new(self)
            .map(|z| f(&z)).collect();
        let right = Rights::new(self)
            .map(|z| f(&z)).collect();
        Zipper { left, focus, right }
    }
}
```

The `Lefts` and `Rights` are internal iterators that produce successive left-shifted and right-shifted zippers by calling `move_left()` and `move_right()` respectively. Each shift clones the internal vectors, resulting in $O(n)$ work per shift and $O(n^2)$ total allocation for a word of length n. For Finnish words (typically 5–15 characters), this cost is negligible; our benchmarks show that a 15-word sentence processes in approximately 0.21 ms. A `SliceZipper` optimization that holds a reference to the underlying slice and moves only a position index would reduce intermediate allocation to $O(n)$.

This approach sacrifices generality—it is not possible to write code that is polymorphic over arbitrary comonads—but gains clarity and avoids the complex trait-bound machinery that Rust's Generic Associated Types (GATs) would require for a full `Comonad` trait. Since our framework requires only a single comonad instance (`Zipper`), this trade-off is appropriate.

A.2 CoKleisli Arrows in Rust

Each morphophonological rule is a plain Rust function with the signature `fn(&Zipper<char>) -> char`. For consonant gradation, the grade parameter is supplied via partial application using a closure:

```
pub fn apply_gradation(
    z: &Zipper<char>, grade: Grade
) -> char {
    let focus = *z.extract();
    let left  = z.peek_left(1).copied();
    let right = z.peek_right(1).copied();
    match find_pattern_at_pos1(
        left, focus, right, grade
    ) {
        Some(pat) => /* target char */,
        None      => focus,
    }
}
// Usage with extend:
let weak = zipper.extend(
    |z| apply_gradation(z, Grade::Weak)
);
```

The CG-lite rules use a trait-based dispatch instead:

```
pub trait CgRule {
    fn apply(&self,
        z: &Zipper<ReadingSet>
    ) -> ReadingSet;
}
```

Each concrete rule type (24 types including `RemoveIfPreceded`, `SelectIfFollowed`, `RemoveByClass`, `SelectByBaseform`, and others) implements this trait. The `apply` method has the coKleisli arrow signature `&Zipper<ReadingSet>` $\rightarrow$ `ReadingSet`, and can be passed directly to extend:

```
let result = zipper.extend(
    |z| rule.apply(z)
);
```

A.3 The Gradation Pattern Table

The 11 consonant gradation patterns are encoded in a static table of `GradationPattern` structures, each specifying a two-character window for both the strong and weak grades:

```
const PATTERNS: &[GradationPattern] = &[
  // Geminates (highest priority)
  GP { s: ['p','p'], w: ['p','\0'] },
  GP { s: ['t','t'], w: ['t','\0'] },
  GP { s: ['k','k'], w: ['k','\0'] },
  // Clusters
  GP { s: ['m','p'], w: ['m','m'] },
  GP { s: ['n','t'], w: ['n','n'] },
  // ... (6 more patterns)
  // Single consonants (lowest priority)
  GP { s: ['\0','p'], w: ['\0','v'] },
  GP { s: ['\0','t'], w: ['\0','d'] },
  GP { s: ['\0','k'], w: ['\0','\0'] },
];
```

The ordering within the table encodes pattern priority: the first matching pattern wins. The `'\0'` character serves as an implementation artifact in the internal representation: at position 0 of single-consonant patterns, it indicates "preceded by any vowel," distinguishing these patterns from cluster patterns that require a specific left neighbor; as

an output character, it signals deletion. When the pattern table produces a `'\0'` output, the Writer comonad arrow (§3.6) intercepts it and records the position in a $DeletionSet$ instead of emitting the null character, restoring strict coKleisli compositionality. The `'\0'` never appears in final output—it exists solely as an internal sentinel consumed by the Writer comonad's materialize step.

A.4 Bidirectional Support

All 11 patterns support both weakening (strong → weak) and strengthening (weak → strong). The find_pattern_at_pos1 function selects which side of the pattern to match against based on the $Grade$ parameter: for weakening, it matches the strong side and produces the weak side; for strengthening, vice versa.

A.5 Test Methodology

The implementation is verified through 317 comonad-related tests organized in six files:

Component	File	#	Verification
Zipper	`zipper`	27	Laws, navigation
Morpho	`finnish`	89	11 patterns, roundtrip
Writer	`writer`	44	Monoid/comonad laws, pipeline equiv.
CG-lite	`cg`	138	24 rule types, safety
Proptest	`proptest`	12	Property-based law verification
Bench	`bench`	7	Perf. regression

Table 7: Test suite organization. All 317 tests pass on CI.

Comonad law tests. The left identity law (extend extract = id) is tested for zippers at multiple focus positions. The right identity law (extract ∘ extend f = f) is tested at every position in a 4-element zipper, confirming that the `Lefts`/`Rights` iterators correctly enumerate all positions. The Writer comonad (`WriterZipper`) is additionally verified for all three comonad laws with monoid accumulation, monoid laws for DeletionSet (identity, associativity, idempotence), and equivalence between the Writer pipeline and the sentinel-based pipeline on all 11 gradation patterns.

A.6 L3′ Proof for the Writer Comonad

We must show $\mathsf{extend}_W\ g \circ \mathsf{extend}_W\ f = \mathsf{extend}_W\ (g \circ \mathsf{extend}_W\ f)$. Expanding the left side, the first extend_W applies f at every position, accumulating a deletion set $D_f = \bigcup_i \pi_1(f(w, z_i))$; the second applies g to the result, accumulating $D_g = \bigcup_j \pi_1(g(w \cup D_f, z'_j))$. The total accumulated set is $(w \cup D_f) \cup D_g$. On the right side, extend_W applies the composed arrow $g \circ \mathsf{extend}_W\ f$ at each position, feeding f's output (including its accumulated deletions) directly into g, yielding $w \cup (D_f \cup D_g)$. Because $(DS, \cup, \emptyset)$ is an associative monoid, $(w \cup D_f) \cup D_g = w \cup (D_f \cup D_g)$, so the deletion components coincide. The character components coincide by L3′ of the underlying Zipper comonad.

Roundtrip tests. For non-destructive gradation patterns (clusters and single consonants), the roundtrip property $\mathsf{strong}(\mathsf{weak}(word)) = word$ is verified. For example, weakening *kampa* to *kamma* and then strengthening back produces *kampa*. The geminate patterns do not satisfy roundtrip because weakening *kaappi* to *kaapi* (via $DeletionSet$ materialization) loses the information that the original contained a geminate; strengthening *kaapi* cannot distinguish between a weakened geminate and a lexical singleton. This asymmetry is a fundamental property of the deletion operation, not a limitation of the comonadic framework.

B Extended Evaluation Details

B.1 Vowel Harmony Verification

Vowel harmony resolution is tested for back-vowel stems (*talo* + *-ssA* → *talossa*), front-vowel stems (*pöydä* + *-ssA* → *pöydässä*), and stems with only neutral vowels (*tie* + *-ssA* → *tiessä*, defaulting to front). The detect_harmony function correctly scans leftward through neutral vowels to find the nearest non-neutral vowel. All three archiphonemes (A, O, U) are tested in both back and front contexts. The harmony function is idempotent: applying it to already-resolved text produces no change.

B.2 Pipeline Integration Examples

The morphophonological pipeline gradation ⪢ harmony ⪢ possessive is tested on words requiring multiple simultaneous transformations:

- *rantAssA* (weak) → *rannassa* (gradation: nt→nn, harmony: A→a)
- *pukussA* (weak) → *puussa* (gradation: k deleted, harmony: A→a)

- *kampAstAVn* (weak) → *kammastaan* (all three rules active)
- *kenkässtAVn* (weak, front) → *kengästään* (gradation: nk→ng, harmony: A→ä, possessive: V→ä)

B.3 CG-lite Disambiguation Examples

The CG-lite module is evaluated on both constructed and corpus-derived Finnish examples. In the rules below, Finnish CG tag names are used: *lukusana* = numeral, *nimisana* = noun, *teonsana* = verb, *laatusana* = adjective, *seikkasana* = adverb.

"kuusi koiraa" ('six dogs'). The word *kuusi* has readings {numeral/kuusi, noun/kuusi}. The rule `SELECT lukusana IF (+1 nimisana)` correctly selects the numeral reading when followed by the noun *koiraa*.

"kuusi kasvaa" ('spruce grows'). With the rule `SELECT nimisana IF (+1 teonsana)`, the noun reading of *kuusi* is correctly selected when followed by the verb *kasvaa*.

"ei voi" ('cannot'). The word *voi* is ambiguous between 'butter' (noun) and 'can' (verb). The rule `SELECT teonsana IF (-1 BASEFORM=ei)` correctly selects the verb reading when preceded by the negation verb *ei*.

3-rule cascade. A sentence position with 4-way ambiguity {noun, verb, adjective, adverb} is reduced to a single reading through three sequential extend passes: (1) `REMOVE laatusana IF (NOT -1 lukusana)`, (2) `REMOVE seikkasana IF (-1 nimisana)`, (3) `SELECT teonsana IF (+1 teonsana)`. This demonstrates the progressive disambiguation enabled by coKleisli composition, where each rule's pruning enables subsequent rules to make sharper decisions.

B.4 Qualitative Coverage Details

The non-roundtrip cases (geminate weakening and *k* deletion) involve information loss: the deletion of a character removes the evidence needed for reverse reconstruction. Of the 11 patterns, 7 exhibit full bidirectional roundtrip correctness; the 4 non-roundtrip cases all involve deletion.

Pattern	In	Out	RT
pp → p	kaappi	kaapi	–
tt → t	matto	mato	–
kk → k	kukka	kuka	–
p → v	tupa	tuva	✓
t → d	katu	kadu	✓
k → ∅	puku	puu	–
mp → mm	kampa	kamma	✓
lt → ll	kulta	kulla	✓
nt → nn	ranta	ranna	✓
rt → rr	parta	parra	✓
nk → ng	kenkä	kengä	✓

Table 8: Gradation patterns as coKleisli arrows (weakening direction). RT = roundtrip $\mathsf{strong}(\mathsf{weak}(w)) = w$.