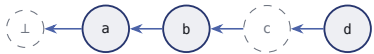
SUMMER INTERNSHIP PRESENTATION

Collaborative text editing, verified

RGA, tombstones, and a tombstone-free sequence **MRDT** mechanized in Lean 4

the **Sal** framework · a Lean port of **Neem** (F*, OOPSLA 2025)



a chain of characters,
one of them already deleted

Harisankar B

National Institute of Science Education and Research

under the guidance of **Dr. KC Sivaramakrishnan**

FP Launchpad, IIT Madras · June–July 2026

The arc of the talk

Part I — the problem

1. Offline editing, and why replication makes it hard
2. Indices break under concurrency
3. CRDTs; RGA as one

Part II — what "correct" means

4. MRDTs, Neem, Sal
5. Convergence is not enough
6. RA-linearizability

Part III — the work

7. Tombstones, and a tombstone-free RGA that *rehomes*
8. It was proved RA-linearizable…
9. …and it is still wrong
10. **EmbedRGA**: immutable coordinates

We all live in these

Google Docs | Overleaf | Notion | Apple Notes

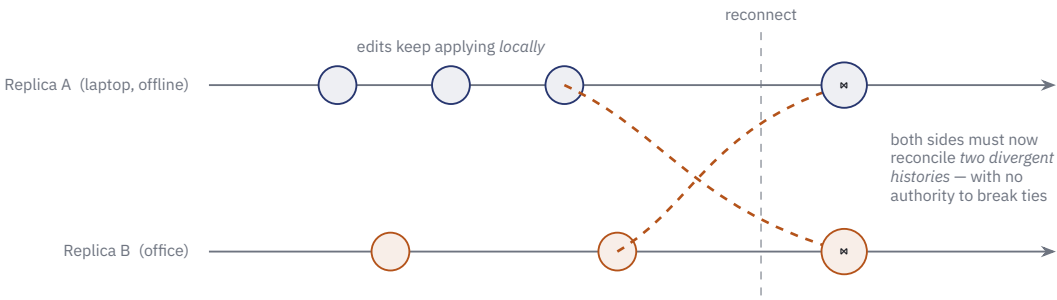
multiple people, one document, edits landing in real time

- Two authors typing in the same paragraph, at the same instant, is *normal*.
- Nobody accepts a lock, a “document is busy” dialog, or a merge conflict marker in the middle of a sentence.
- And nobody accepts losing the paragraph they wrote on a plane.

The product requirement is: **always writable, never lost, everyone eventually sees the same document** — and it should be the document people meant.

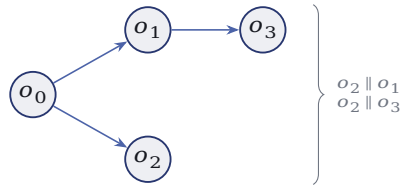
Offline editing, concretely

Every client is a full replica of the document; the network is optional



- No round trip on the critical path: the edit is applied to the local copy *first*.
- So divergence is not a failure mode — it is the *steady state* of the system.

Why distribution makes this hard



visibility is a partial order, not a sequence

- **No global order.** No clock and no leader that both replicas trust. “Which edit came first?” often has *no answer*.
- **Arbitrary delays.** Tuesday’s edit can arrive after Friday’s.
- **Local-first.** You cannot wait for consensus — availability during a partition is the point.
- **Concurrency is genuine.** Any rule resolving two mutually unaware edits must give the *same* answer at every replica, in any arrival order.

Sequential reasoning — “apply the ops in order” — is exactly what we are not allowed to assume.

The obvious protocol — and it is wrong

Send what your editor already knows: a position and a character

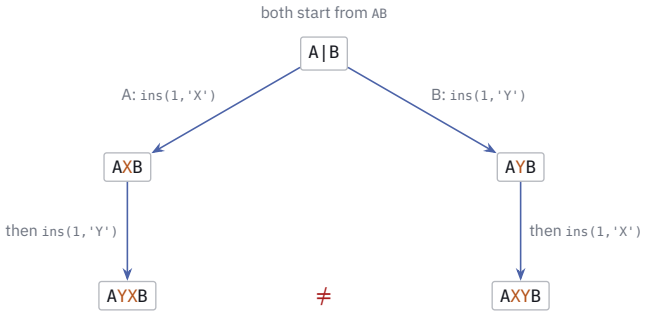
```
insert(index = 2, char = 'X')
delete(index = 0)
```

this is the vocabulary of `String.insert` / `splice` — the API every editor already has

- An index is a *coordinate in a document that is changing underneath it*.
- It only means something relative to *one* replica's state at *one* instant.
- The moment a concurrent edit lands, the index has silently changed what it points at.

An index is not an identity. Next two slides: exactly how that kills you.

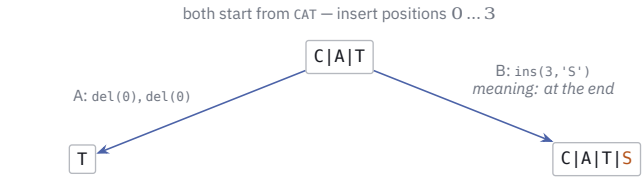
Break #1 — two concurrent inserts, two documents



Same two operations. Same causal history. **Two different documents, forever.** The replicas will never agree again.

Break #2 — the index stops referring to anything

Not “two answers” this time — *no* answer: the arriving operation cannot be applied at all



position 3 does not exist.
T has two positions. The operation is not wrong — it is *meaningless*.

So what does A do?

- **reject it** → T
  the author’s S is silently lost
- **clamp to the end** → TS
  a position nobody asked for

Every repair is a **guess about intent** — and a rule that forces the replicas to agree just makes them agree on the guess. **Convergence was never the goal.**

The fix in principle: name characters, don't count them

`insert` *at position 2*

position 2 of *whose* document, *when*?

`insert` *after character* id_7

id_7 means the same thing at every replica, forever

- Give every inserted character a globally unique, immutable identifier: *(timestamp, replica id)*.
- An insertion names its **anchor** — the character it goes after.
- Now the operation is *context-free*: it can be applied at any replica, in any order, and still mean the same thing.
- The document is no longer a string. It is a **tree of identities**, read out by a deterministic traversal.

This is the move that makes convergence achievable at all — and it is what a **sequence CRDT** is.

CRDTs: convergence as an algebraic contract

A *state-based* CRDT is a state space with a merge:

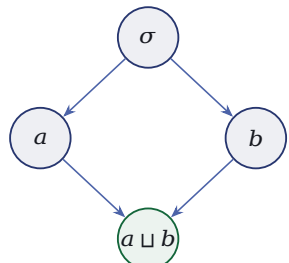
$$\text{merge} : \Sigma \to \Sigma \to \Sigma$$

required to be

- **commutative** $a \sqcup b = b \sqcup a$
- **associative** $(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$
- **idempotent** $a \sqcup a = a$

i.e. a **join-semilattice**. Then:

> **Strong Eventual Consistency (SEC).** Any two replicas that have received the same set of updates are in the same state — regardless of *order*, *duplication*, or *delay*.



Shapiro, Preguiça, Baquero, Zawirski, INRIA RR-7506 (2011).

RGA: the sequence CRDT behind Automerge and Yjs

Replicated Growable Array — Roh, Jeon, Kim, Lee, JPDC 71(3), 2011

State — a grow-only set of records

$$(id,\ char,\ afterId)$$

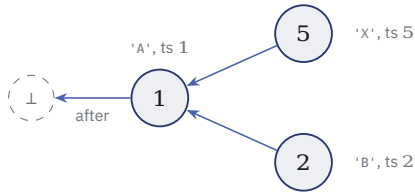
plus a set of **tombstones**: ids that have been deleted.

Operations

- Insert — add a record, anchored at an existing id
- Remove — add the id to the tombstone set

Merge — componentwise union.

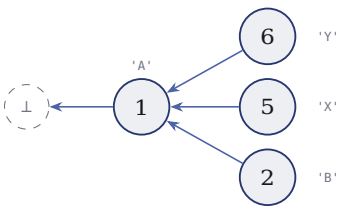
Both components are grow-only, so merge is trivially a semilattice join. **All the interesting content is in the read.**



the birth tree: each node points at the character it was inserted after

RGA: the read is a traversal, and it settles the race

Depth-first from the root, and among **siblings** sharing an anchor, newest id first:



concurrent under `'A'`: ids $6 > 5$, so `Y` then `X`

Read: `A Y X B` — *at every replica.*

`docs/rga-vs-paper.md`

In Lean, the order is an inductive relation `visible_lt` with four rules (`RGA_ReadSide.lean`):

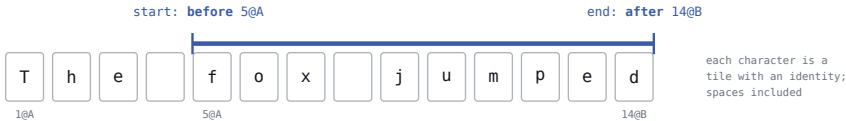
`parent_child`	anchor before its children
`sibling`	larger id first
`left_desc_of_sib`	a sibling's subtree stays together
`trans`	transitive closure

Three intent theorems, kernel-checked, on both the CRDT and the MRDT:

- `causal_order_visible_lt`
- `tombstone_monotone_under_remove`
- `concurrent_insert_tiebreak_deterministic`

Real documents are rich text

Bold, italics, links, comments — and formatting by *index range* breaks for exactly the reason inserting by index breaks



A mark is not a range of positions. It is a **pair of anchors**, each naming a character *and a side*:

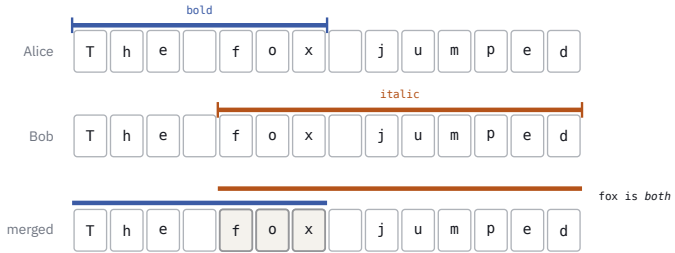
```
action: "addMark"
markType: "bold"
start:  { type: "before", opId: "5@A" }
end:    { type: "after", opId: "14@B" }
```

- Concurrent inserts shift every index, so “bold characters 5–14” means a different thing at each replica — *the same failure as slide 7*.
- Anchoring to identities fixes the span. The **side bit** then decides what happens to text typed *at* the boundary.

Litt, Lim, Kleppmann, van Hardenberg, CSCW 2022. Figure style after `inkandswitch.com/peritext`.

Peritext = RGA + a set of marks

The character sequence is an RGA; the marks ride on top of it and merge by union




```
[ {text: "The ",  format: {bold}     },
 {text: "fox",  format: {bold, italic}},
 {text: " jumped", format: {italic}    } ]
```

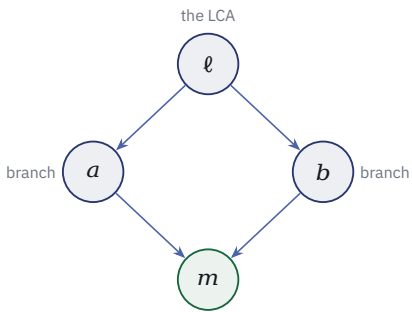
The render is a left-to-right fold over reading order, opening and closing marks — no conflict to resolve, because the marks set is grow-only.

In Sal (`Peritext_with_tombstones/`, CRDT *and* MRDT):

- 24 VCs closed on four grow-only components (`chars`, `afters`, `deleted`, `marks`).
- Paper examples **1, 2, 3, 5, 6, 7, 8** as kernel-checked read-side intent theorems — including Ex 7 bold-expand (`bold_expand_reach`) and Ex 8 link-contract.
- Ex 4 (two *colours* of one mark type) is a stated gap: `MarkOp` has no value field.

MRDTs: git, but for data types

Mergeable Replicated Data Types — the merge gets to see the common ancestor



version DAG, like commits

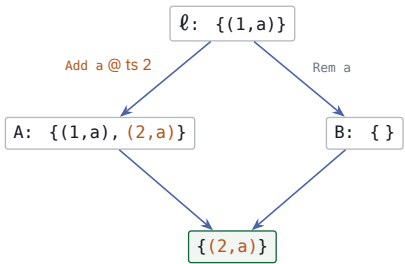
$$\text{merge} : \Sigma \to \Sigma \to \Sigma \to \Sigma$$

$$\text{merge}\ \ell\ a\ b$$

- ℓ is the **lowest common ancestor** of the two branches.
- The merge can now tell *"x was never there"* from *"x was there and was deleted"* — by looking at ℓ.
- That is exactly the information a CRDT has to encode in **tombstones** — so the LCA can pay for it instead.
- OR-Set drops its remove-set entirely: $(\ell \cap a \cap b) \cup (a \setminus \ell) \cup (b \setminus \ell)$

OR-Set: one execution of the merge

State is a set of `(timestamp, element)` tags — and there is no remove-set anywhere



a is live — **add wins**

both replicas converge here

merge $\ell\ a\ b\ =\ (\ell \cap a \cap b) \cup (a \setminus \ell) \cup (b \setminus \ell)$

$\ell \cap a \cap b = \{\}$ B removed it

$a \setminus \ell = \{(2, a)\}$ the new tag

$b \setminus \ell = \{\}$

- Every add stakes a **fresh, globally unique** id, which each replica mints for itself — *no coordination required*. So `(2,a)` is a different tag from `(1,a)`.
- `(1,a)` was in ℓ, so B's remove **observed** it. `(2,a)` was not, so B *cannot* have seen it — and it survives.

The question this work starts from: if the LCA can replace OR-Set's tombstones, can it replace RGA's?

Neem, and Sal

Reduce “is this data type correct?” to a fixed, discharge-able set of obligations

Neem (Soundarapandian, Nagar, Rastogi, Sivaramakrishnan — OOPSLA 2025, F*)

You give a signature

$$\langle\, \Sigma,\ \sigma_0,\ \mathrm{do},\ \mathrm{merge},\ \mathrm{rc} \,\rangle$$

where rc resolves *non-commuting* operation pairs (`Fst_then_snd` / `Snd_then_fst` / `Either`).

Neem’s metatheorem: discharge **24 verification conditions** over do, merge, rc — and RA-linearizability follows *for every execution*.

`rc_non_comm`, `no_rc_chain`, `merge_comm`, `merge_idem`, `base_1op`, `base_2op`, `ind_lca_2op`, `inter_left_1op`, `lem_0op`, …

Sal — the Lean 4 port (Ramesh, Soundarapandian, Sivaramakrishnan)

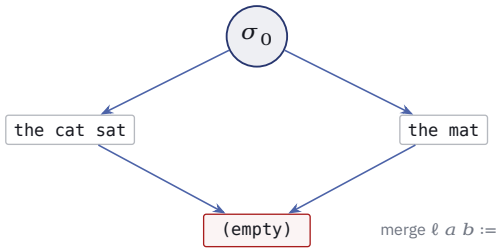
- Same reduction, plus a *multi-modal* tactic that stages automation by **trustworthiness**:

  `dsimp+grind` → `lean-blaster` (Z3) → interactive
- Counterexample pipeline: an *invalid* VC becomes an inspectable execution trace (Plausible + ProofWidgets).
- Decidable set/map interfaces so `grind` is effective on RDT goals.

29 RDTs — 17 CRDTs, 12 MRDTs. **648 VCs** for state convergence, the vast majority kernel-checked.

Convergence is not enough

SEC is a statement about *agreement*. It says nothing about *content*.



commutative ✓
associative ✓
idempotent ✓
SEC ✓

useless ×

- “Delete everything on merge” is a perfectly good CRDT. So is “ignore all operations”. Both converge.
- Convergence is a *necessary* condition that any number of wrong data types satisfy.

A verified CRDT needs a correctness criterion that mentions **what the operations mean**, not only that replicas agree.

And for sequences the problem is sharper

The vacuousness gradient across the Sal suite

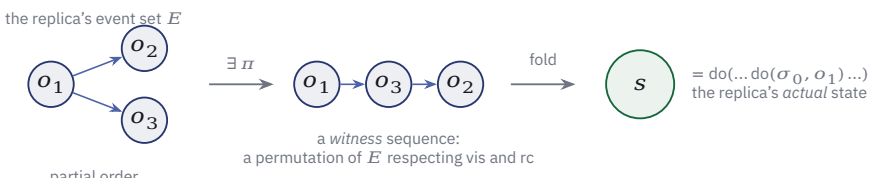
Tier	Character	Examples	The 24 VCs prove…
A	state *is* the semantic content	LWW-/Max-/Min-Register, PN-Counter	lattice join, arithmetic — real content
B	merge does real computation	MVR, LWW-Map, LWW-Element-Set	conflict collection — real content
C	state is a grow-only bag; **meaning lives in the read**	OR-Set, **RGA**, Add-Win-PQ, **Peritext**	grow-only union — near-trivial

- Roughly half the suite — and *all* the interesting data types — are Tier C.
- For RGA, the 24 VCs prove that two grow-only sets converge under union. They never mention the traversal that turns those sets into a document.
- Hence: every RDT in Sal carries a `*_ReadSide.lean` companion, and the Tier-C ones carry *intent-preservation* theorems matched to their papers.

`docs/papoc-2026-keynote-notes.md` — "the honest claim is not *proven correct*".

Replication-aware linearizability, in words

Every state a replica can ever hold must be **explainable** as the result of running its operations **one at a time, in some order** — an order that *respects causality* and *obeys the data type's own tie-breaking rule*.



- So a merged state is never "some third thing": it is always a sequential history someone *could* have executed.

RA-linearizability, precisely

```
def IsRALinearizable3 (C : Configuration D) : Prop :=
  ∀ (v : Version) (s : D.State) (E : Set (Op D.AppOp)),
    C.ver v = some (s, E) →
    ∃ π : List (Op D.AppOp),
      listPermOf π E ∧                  -- π enumerates exactly E
      respects π (lo (Configuration.core C)) ∧   -- causality + rc
      applySeq D.toCRDTSig D.init π = s          -- replaying π gives s
```

Remember this shape. **The reference sequence is the data type's *own* `do`-fold.** Slide 27 is about what that does *not* buy.

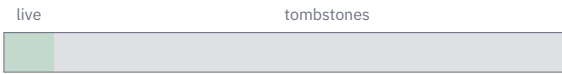
- Quantified over *every version in the configuration* — replica heads *and* historical versions used as LCAs.
- lo = visibility ∪ the rc arbitration; its acyclicity is exactly what `no_rc_chain` buys you.

`Sal/ConditionedMRDTs/Metatheory/Adequacy.lean`

The tombstone problem

A tombstone is a deleted character that we **cannot throw away**, because other characters are positioned relative to it.

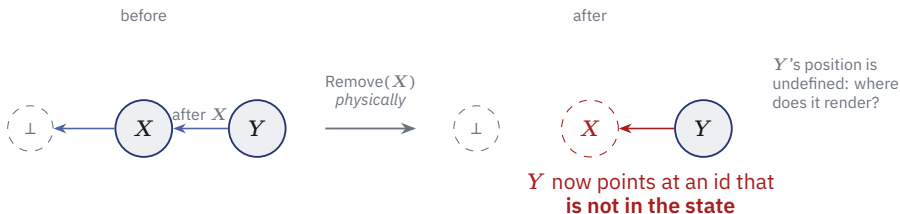
- State grows with the *total* number of edits, not the document size.
- A heavily-revised paper is mostly graveyard: a few thousand live characters, hundreds of thousands of dead ones.
- Garbage collection needs to know that *every* replica has seen the delete — which needs consensus, which is what we gave up.



what the reader sees vs. what the replica stores

The goal: Remove physically removes the record, so state is bounded by *live* characters. And the MRDT's LCA is exactly the extra information that might pay for it.

Why you cannot simply delete the record



Tombstones in OR-Set and in RGA carry different kinds of information. OR-Set's tombstone is a *boolean* (membership history) — an LCA can recompute it. RGA's tombstone is a *graph node* (structural position) — it is load-bearing geometry.

Rehoming: how a survivor finds a new anchor

`Sal/MRDTs/RGA_Rehoming/` — state is `map ℕ (α × ℕ)`: id ↦ (element, anchor). Deletion really removes the record.

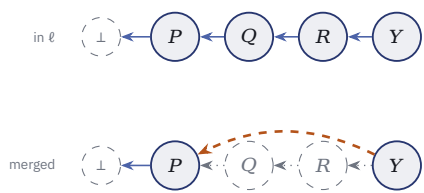
In merge: climb the LCA.

Survivorship is the OR-set formula, giving a survivor set I. A survivor whose recorded anchor is dead walks *up the ancestor chain the LCA already knows*, until it reaches a node that also survives — or the root.

```
let ancL := fun y => anc l y   -- from the LCA

def climb_aux (ancL) (I) : ℕ → ℕ → ℕ
  | 0,      x => x
  | fuel+1, x => if x = 0 || I x then x
                 else climb_aux ancL I fuel (ancL x)
```

merge reads no operation and no path: the dead node's position is still in ℓ, and *that* is what pays for the tombstone.



Y's recorded anchor is R. A branch deleted Q and R — both records physically removed — leaving $I = \{P, Y\}$.

In do there is no LCA — the operation *carries* its target's ancestor chain, and `resolve` takes the first entry still live. Both halves together: `rc = Either` at every reachable state (`rc_non_comm'`).

It was proved — and the 24-VC schema had to be widened

Why it falls outside the standard schema. Commutation holds only on *reachable* states — the op's claimed path must really be its ancestor chain (`accurate`), timestamps fresh, root never stored. That is a *conditioned* commutativity; the plain 24 VCs quantify over all states.

A prefix-free variant is *impossible*: `RGA_PrefixFree_Impossible.lean`.

So we built the conditioned metatheory and proved the theorem directly:

`rga_ra_linearizable3_eq`

RA-linearizability **up to observational** ≈ at *every reachable configuration*, under a single honest-delivery assumption. **Kernel-clean**, 0 `sorry`.

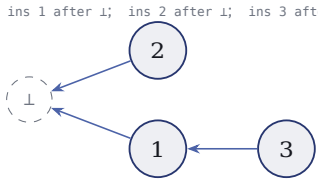
The mechanization pushed back — seven times. Each item below is a premise we *believed*, stated in Lean, and then *refuted* with a machine-checked countermodel:

- `hEnum` as first stated is **false**: an LCA-first delta enumeration pre-applies deletes that are concurrent with delta inserts.
- K2 is **false**: the *criss-cross* execution — 7 ops, two branches that observed *incompatible delete orders*. No single guarded sequence replays both.
- Even the *capstone statement* was unsatisfiable as first written, and had to be restated at the raw/≈ level.

The lesson underneath all seven: tombstone-free rehoming makes *delete order observable* in the survivors' stored parents.

...and it is still wrong

Four operations on **one** replica. No concurrency. No merge. Just a backspace.



`reads [2, 1, 3]`

at ⊥: ids $2 > 1$, so 2 first; then 1, then 1's child 3

`reads [3, 2]`

the survivors swapped

`should be [2, 3]`

⊥'s children are now $\{2, 3\}$ — and $3 > 2$

3 was *buried* under 1, so its timestamp never competed with 2's. The splice promotes it to the root, where it wins the tiebreak.

The defect is a theorem, not an anecdote

```
-- The independent intent spec: a delete removes its target and nothing else moves.
def DeleteOrderPreserving : Prop :=
  ∀ (s : concrete_st) (t r x : ℕ) (p ids : List ℕ),
    List.Pairwise (· > ·) ids → (∀ i, contains s i = true → i ∈ ids) →
    document (do_ s (t, r, .Del p x)) (ids.filter (· ≠ x))
      = (document s ids).filter (· ≠ x)

theorem tombstone_free_violates_delete_order : ¬ DeleteOrderPreserving := by
  -- witness: s_bac with the complete, strictly-descending candidate list [3,2,1]
  ...  exact absurd key (by native_decide)
```

- The candidate list is guarded **descending** and **complete**, so the discrepancy cannot be an artifact of how we read.
- Companion: `del_a_breaks_survivor_order`.
- The positive contrast is a theorem too: `remove_preserves_visible_lt` holds for the *tombstoned* RGA.

The rehoming RGA converges, is RA-linearizable, is kernel-clean — and **does not implement a text buffer**. A single backspace reorders text the user never touched.

`Sal/MRDTs/RGA_Rehoming/RGA_Tombstone_Free_SPOT.lean`

Why the proof did not see it

both sides are computed with **the same** `do` — so if `do` is wrong, *both sides are wrong the same way*, and the VC passes

Own-fold RA-linearizability certifies **convergence and self-consistency**.

It does **not** certify *fidelity to the intended sequence semantics*.

- The verification analogue of *"your tests pass but the code is wrong"*: proofs validate the **proof**, not the **spec**.
- Catching it needs a specification **independent of the implementation's own fold** — here, the published RGA's traversal order.
- Same failure mode as Peritext: `in_span_boundary` encoded the *opposite* of the paper's §3.3, and ∼400 lines of correct proofs about it were deleted.

Is it patchable? No — and that is informative

A fooling-pair argument: consider two executions whose *live* characters and live geometry agree, but whose *deleted* history differs. RGA's traversal distinguishes them. A tombstone-free state cannot.

No *bounded* tombstone-free state can preserve RGA order.

Slogan: *the deleted path ≡ the tombstones*. If the geometry you need is the dead nodes, you must either keep them, or stop needing them.

So where is the real culprit?

Delete reorders survivors *because* survivors' positions are stored *relatively* — and a relative position must be recomputed when its reference dies.

Fix the representation, not the merge. Make position a *birth constant*.

EmbedRGA: immutable birth coordinates

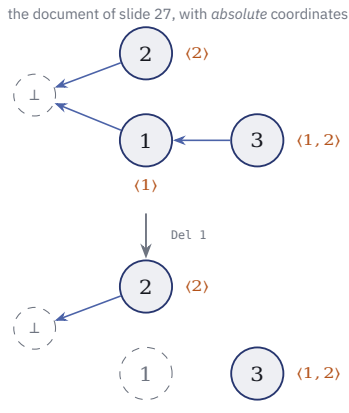
`Sal/MRDTs/RGA_Embed/` — state is `map ℕ (α × List Bool)`: id ↦ (element, *coordinate*)

On insert, mint the newcomer's coordinate from its anchor's — once, and forever:

$$\text{coord}(t) \;=\; \text{coord}(a) \parallel \Gamma.\text{enc}(t - a)$$

- a = anchor id, t = the new timestamp; $t - a$ is the **delta**.
- Γ is any *order-preserving, prefix-free* code.
- do on Ins **never reads the state** — so every operation pair commutes *unconditionally* enough that `rc = Either` discharges with no path algebra at all.

Del carries *no path* and rehomes *nothing*. merge *never climbs*: OR-set survival, values copied.



Why sorting coordinates *is* the RGA traversal

Compare two coordinates as delta sequences (`chainBefore`):

1. If one is a **prefix** of the other, the prefix comes first.
   = RGA's `parent_child` rule
2. Otherwise, at the **first differing delta**, the **larger** delta comes first.
   = RGA's `sibling` rule (larger delta ⇔ newer timestamp under a shared anchor)

Reading off slide 31: $\langle 2 \rangle \prec \langle 1 \rangle \prec \langle 1, 2 \rangle \Rightarrow [\,2, 1, 3\,]$; after the delete, $[\,2, 3\,]$. **Order preserved.**

Mechanically, it is one lexicographic compare. Concatenate the codewords, append a terminator symbol above the digit alphabet, sort descending:

```
def key (c : coord) : List ℕ :=
  (c.map fun b => if b then 2 else 1) ++ [3]
```

- `display_iff_chainBefore` — the key comparison computes *exactly* `chainBefore`.
- `coordOf_inj` — prefix-freeness gives **unique decodability**: distinct birth chains, distinct coordinates.
- `subtree_convex` — a subtree is a coordinate prefix, hence a *contiguous* block of the display: **non-interleaving**, free.
- `before_do_stable` — **axiom-free** step stability; at Del this is general delete-order preservation — the clause the rehoming RGA refutes.

Same witness, opposite verdict — and the delete is a filter

```
-- The rehoming RGA's reorder witness, run through EmbedRGA's real do_ / merge.
theorem reorder_document_before : document s_reorder [5,6,8] = [6, 5, 8] := by native_decide

theorem del_preserves_order :
    document (do_ unaryCode s_reorder (11, 1, .Del 5)) [5,6,8] = [6, 8] := by native_decide

-- Should-FAIL pin: the rehoming RGA's answer on this witness is [8,6]. Not ours.
theorem del_preserves_order_not_rehoming :
    document (do_ unaryCode s_reorder (11, 1, .Del 5)) [5,6,8] ≠ [8, 6] := by native_decide
```

Every SPOT block is PASS+FAIL shaped: each positive verdict is pinned by a ≠ companion that rules out the tempting degenerate answer — here, the rival design's verdict.

- Values are immutable, so pairwise display order is stable at *every* operation *and* every merge (`before_merge_stable`).
- The proof of delete-stability at the list level is, in the end, `List.filter_sublist`.

The two capstones

1. Convergence. `embed_ra_linearizable3`

RA-linearizability *per version* at every honestly reachable configuration — with **strict** $=$, not $\approx$.

Route: the *mergeable-queue* hook, whose join needs canonical states unique per event set. Supplied by `e_fold_canon`: *any two well-formed enumerations of one event set fold to the same state.*

- Parametric in the code: **unary**, **binary-δ** and **Elias-δ** instances all proved, zero new proof content each.
- ≈ 1.4–1.8 bits per level on real traces.

2. Fidelity. `rga_read_eq_embed_read`

The compaction theorem. On honest executions, EmbedRGA's read equals the *published, tombstoned* RGA's read — **element for element**.

⇒ The specification is now *external*: RGA's own `visible_lt`, not our fold. This is what closes `oq:linspec`.

⇒ A by-product the published side never proved: `visible_lt_total` — RGA's relational order *is* total on the birth tree.

Axioms: `propext`, `Classical.choice`, `Quot.sound`. No `sorryAx`.

Tombstone-free, and provably the same document as RGA.

Rich text inherits the fix — and inherited the defect

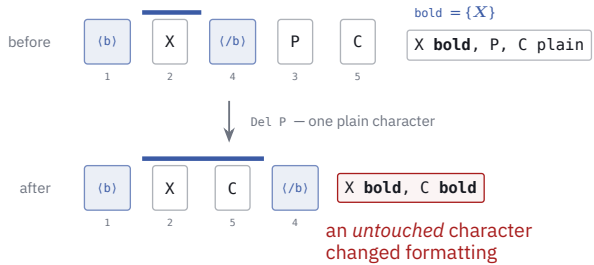
The *fused* design: put the mark boundaries **into** the sequence, so rich text is **one** RGA

```
inductive PeritextElt
  | char  : ℕ → PeritextElt
  | bound : ℕ → Mark → Bool → PeritextElt
--        markId  mark   isStart
```

⇒ The payload is *opaque* to the kernel, so convergence is a one-line instantiation — **no new proof**:

```
theorem peritextEmbed_ra_linearizable3
    (hReach : EReach Γ C) : IsRALinearizable3 C :=
  embed_ra_linearizable3 hReach
```

Both shapes are in the repo: marks in a separate set (as published), and marks fused into the sequence. The fused one rides the embed kernel.



Rehoming kernel: C rehomes onto X, out-ranks the close boundary ($5 > 4$), leapfrogs *inside* the span. `fused_delete_reformats_survivor`

Embed kernel: `renderIds_del` — the post-delete render is the pre-delete render *minus exactly* the deleted entries. Formatting of every survivor is preserved bitwise.

Conclusion

1. Indices are not identities. Collaborative editing works because characters are *named*, and the document is a tree read by a deterministic traversal.

2. Convergence is a substrate, not correctness. "All replicas agree on the empty document" satisfies SEC. For grow-only sequence state, the 24 VCs prove union commutes — the *meaning* lives entirely in the read.

3. The specification must be external. Own-fold RA-linearizability cannot see a broken `do`: both sides of *merge* = *fold* use it. Fidelity needed a theorem against the *published* RGA.

4. Mechanization pays in refutations. It produced a machine-checked *negative*: a proved, converging, RA-linearizable tombstone-free RGA that reorders text under a single backspace — plus seven over-strong premises and a criss-cross countermodel we would never have found on paper.

5. Design for the proof. Making position an *immutable birth constant* turned Del into a filter, merge into a lattice join, and the read into a sort — and only then did both a strict-equality convergence proof *and* read-equivalence with published RGA become available.

Tombstone-free. Order-faithful. Kernel-clean.
And rich text rides along for free.

Thank you

the document, in display order:
coordinates, not indices

the framework	`Sal` — Lean 4; a port of **Neem** (F*, OOPSLA 2025)
the paper	`kcsrk.info/papers/sal_jan26.pdf`
the designs	`Sal/MRDTs/RGA_with_tombstones/`, `RGA_Rehoming/`, `RGA_Embed/`
the defect	`RGA_Tombstone_Free_SPOT.lean`
the fix	`RGA_Embed_ChainLex.lean`, `RGA_Embed_ReadEquiv.lean`

With thanks to **Dr. KC Sivaramakrishnan** for the supervision,
and to **Vimala Soundarapandian** and **Pranav Ramesh**. **FP Launchpad**, IIT Madras.

Backup — one more thing the coordinates bought

Sides as payload: RGA and Fugue become two *policies* over one kernel

Let each chain entry carry a **side** as well as a delta — "I was inserted to the *left* / *right* of my anchor" — with the marker alphabet

R band < marker < L band

and the L band in the *complemented* code.

- `sided_embed_ra_linearizable3` — RA-linearizable for **every** side assignment. Sides are *payload* to convergence.
- `schainBefore_liftR` — the all-right fragment orders exactly like the one-sided design.

The clean statement this makes possible:

Convergence is a property of the datatype. *Which side to pick* — and therefore whether concurrent runs interleave — is a property of the **generation policy**.

Two policies, one proved kernel.

Caveat we state rather than bury: the repo's "Fugue policy" is *not* published Fugue — its right band keeps RGA's newest-first rule where Weidner–Kleppmann use ascending id. Our non-interleaving results are about *our hybrid kernel*.

Backup — rehabilitating orphans inside merge is structurally blocked

Let merge $\ell\ a\ b$ notice that Y's anchor is missing from the merged domain, and rewrite it to the nearest ancestor that survives — reading the ancestor chain out of the LCA ℓ.

Structurally blocked, and the framework says so. The VC

```
lem_0op : eq (merge (do_ l ol) (do_ a ol) (do_ b ol))
             (do_ (merge l a b) ol)
```

fails for ol = Remove X under *any* rehab scheme whose orphan test consults the merged domain.

Why. do and merge disagree about *who is an orphan*:

- LHS delete first: $X \notin$ merged domain, so Y *is* an orphan ⇒ rewritten.
- RHS merge first: all three inputs still have X, so Y is *not* an orphan; the outer delete then strands it ⇒ never rewritten.

The lattice-join merges the 24 VCs were designed around touch ℓ only through *set algebra*. Rehabilitation is a *structural query* on ℓ (anc $\ell\ x$) — and that is precisely what an outer do can invalidate.

`docs/rga-rehab-limitation.md` — also: static-payload splice-at-do fails `no_rc_chain` on chain-aligned removes.