Why the Path Matters

A visual account of the tombstone-free, path-carrying RGA

Sal / FP Launchpad

Abstract

The replicated growable array (RGA) in `RGA_Tombstone_Free` stores only *live* elements: a delete physically removes its target, leaving no tombstone. To stay convergent without tombstones, each operation carries the *ancestor path* of the node it references. This note explains, with pictures, two things: **(i) why the implementation converges** — the three-way merge recovers a concurrently-deleted node's parent from the lowest common ancestor (LCA); and **(ii) why the path is necessary *for the single-replica commutation proof*** — to discharge the framework's do-commutation VC, where there is no LCA, an insert whose anchor was deleted has nowhere to recover its position *unless it carries that position itself*. The qualification matters: the merge alone already converges the concurrent case *without* any path (`merge_converges_concurrent`); it is this commutation *obligation*, not operational convergence, that the path serves. We give the counterexample that diverges without the path, the fix that the path provides, and a two-sided asymmetry: the insert path is mandatory *semantically* (commutation fails without it), while the delete path, though *runtime*-dispensable, is *proof*-load-bearing — the operation-static certificate that decouples a delete's well-formedness from the reachable-state invariant. Every claim here is mechanised in Lean (§8).

1 The data type: a forest of live records

The state is a finite map $\sigma : \mathrm{id} \mapsto (\mathsf{el}, \mathsf{anc})$ from an identity to its element and its *immediate anchor* — the identity it was inserted after. Identity 0 is the root sentinel and is never stored. So a state *is* a forest rooted at 0: an edge from a node to its anchor (Figure 1). We write $\mathsf{ids}(\sigma)$ for the stored identities, $\mathsf{el}(\sigma, t)$ and $\mathsf{anc}(\sigma, t)$ for the element and anchor of t, and $\mathsf{contains}(\sigma, t)$ for "t is live". Records use the reference notation (id, element, anchor).

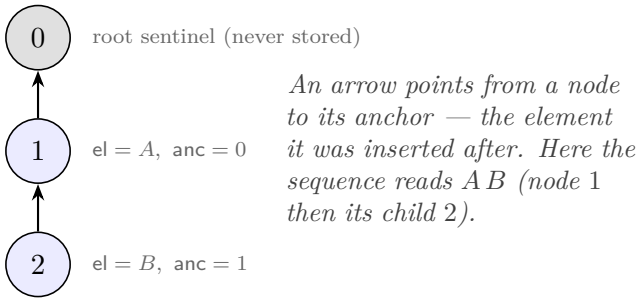

Figure 1: The state as a forest. $\sigma = \{(1, A, 0), (2, B, 1)\}$.

Reading the text. The state stores *anchors*, not positions, so the visible sequence is recovered from the forest at read time by a *depth-first preorder* traversal from the root sentinel 0: visit a node,

emit its element, then recurse into its children — and among the children anchored at one node, visit them in *descending timestamp order* (newest first). That newest-first tiebreak is the standard RGA sibling rule; it is what makes concurrent insertions at the same anchor converge to a single deterministic order, and it is where the linear order lives (the forest itself records only parent edges). In Figure 1 the traversal visits 1 then its child 2, reading $A\,B$. The traversal is purely a read-side computation over the *converged* forest — merge and the VCs operate on the forest, and the sequence is materialised only when read (mechanised as `document` in `RGA_Tombstone_Free_ReadSide.lean`).

Operations. There are two, and (crucially) each carries a *path* p — the ancestor chain of its referenced node, nearest first:

- $\mathsf{Ins}(e, p, a)$ at a fresh id t: insert e anchored at a. It stores $(t, e, \mathsf{resolve}(\sigma, a :: p))$, where $\mathsf{resolve}$ returns the first *live* candidate in its list, else 0. When a is live this is just a; when a is dead it climbs p to the nearest live ancestor.
- $\mathsf{Del}(p, x)$: *splice* out x — physically remove it and re-parent its children to $\mathsf{resolve}(\sigma, p)$, the nearest live ancestor of x. No tombstone is kept.

Figure 9 shows the defining move of a tombstone-free delete: the splice. Deleting 2 removes it and lifts its child 3 one level, onto 2's parent 1.

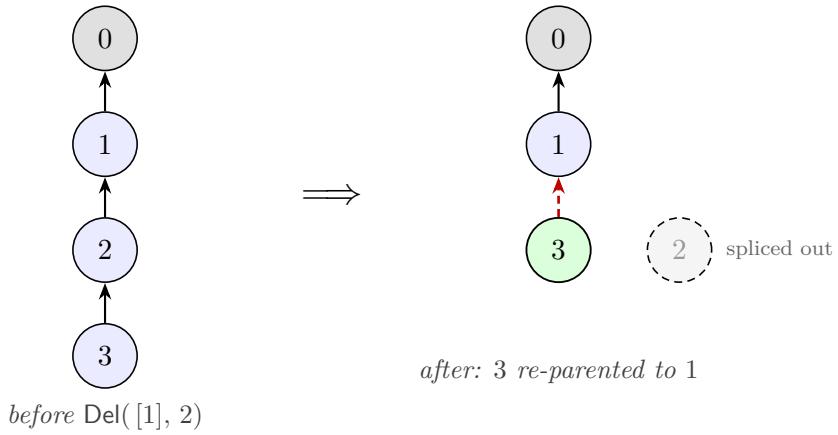

Figure 2: The splice. A tombstone-free $\mathsf{Del}(2)$ physically removes 2 and rehomes its child 3 to 2's live parent 1 (red dashed = the new anchor). Compare a tombstone design, which would instead keep 2 as a dead marker.

2 Why the implementation converges: the LCA does the remembering

Concurrency is reconciled by a three-way merge $\mathsf{merge}(L, A, B)$ where L is the least common ancestor of branches A and B. It has two steps: *survival* (an OR-set merge on the three id sets) and *re-parenting* (climb each survivor's birth-anchor up L's chain). The survivor set is

$$I \;=\; \big(\mathsf{ids}(L) \cap \mathsf{ids}(A) \cap \mathsf{ids}(B)\big) \;\cup\; \big(\mathsf{ids}(A) \setminus \mathsf{ids}(L)\big) \;\cup\; \big(\mathsf{ids}(B) \setminus \mathsf{ids}(L)\big)$$

— an id lives iff it is kept on both sides, or freshly added on either; equivalently, a node lives unless it was in L and deleted on a side. Each survivor t takes its element $\mathsf{el}(t)$ and *birth-anchor* $\beta(t)$ from

L if present, else A, else B; the birth-anchor is then climbed up L's chain — via L's anchor map anc_L — to the nearest live survivor:

$$\mathsf{climb}_I(a) \;=\; \begin{cases} a, & a = 0 \text{ or } a \in I, \\ \mathsf{climb}_I(\mathsf{anc}_L(a)), & a \in \mathsf{ids}(L) \setminus I. \end{cases}$$

$$\mathsf{merge}(L, A, B) \;=\; \big\{\, (t,\ \mathsf{el}(t),\ \mathsf{climb}_I(\beta(t))) \;\big|\; t \in I \,\big\}.$$

The point for convergence is the phrase **"up L's chain"**: even when a node was deleted on one branch, *the LCA still holds it live, with its parent.* So the merge can always recover where a survivor should attach — it reads parents from L, and never needs the operations' paths. Figure 3 is the canonical case: A inserts 3 after 2 while B concurrently deletes 2; the merge climbs 3's dead birth-anchor 2 up the LCA chain to the live 1.

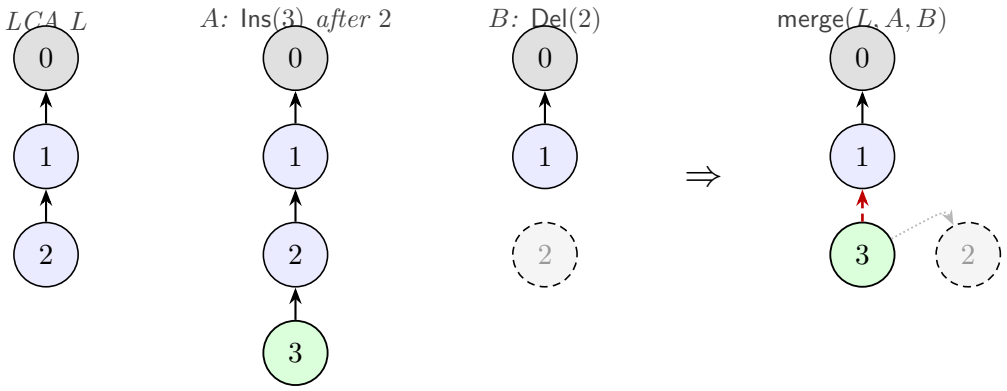

Figure 3: Why merge converges. 3 was born under 2 (dotted, 2's ghost), but 2 is deleted on B. The merge climbs 3's birth-anchor up the LCA chain — $2 \in \mathsf{ids}(L) \setminus I$, so $\mathsf{climb}_I(2) = \mathsf{climb}_I(\mathsf{anc}_L(2)) = \mathsf{climb}_I(1) = 1$ — and rehomes 3 to 1 (red dashed). The LCA is where "2's parent is 1" is read. **No path on the operations is used.**

This is the whole reason a tombstone-free RGA can converge at all: the merge has a third input, L, that retains the deleted node's ancestry. Keep that fact in mind — the next section is about the one place where there is *no* such third input.

3 Why the path is necessary: a single replica has no LCA

Convergence of a replicated data type also requires that operations applied in different orders on *one* replica agree — the framework discharges this as a *commutation* condition on do (single-replica application). And here is the catch: do **has no LCA**. There is only the current state. So when an insert's anchor has already been deleted, the parent the merge would have read from L is simply *gone* — unless the insert brought it along.

The counterexample (no path). Take $\sigma = \{(1, A, 0), (2, B, 1)\}$ and the two operations $\mathsf{Ins}(C, 2)$ (insert C as id 3 after 2) and $\mathsf{Del}(2)$. Apply them in both orders, with a *prefix-free* insert (anchor only, no path):

The two orders disagree on 3's anchor (1 versus 0): the design does *not* commute. Because the delete physically removed 2, the fact "$\mathsf{anc}(2) = 1$" survives nowhere on the right — and a prefix-free insert has no way to reconstruct it.

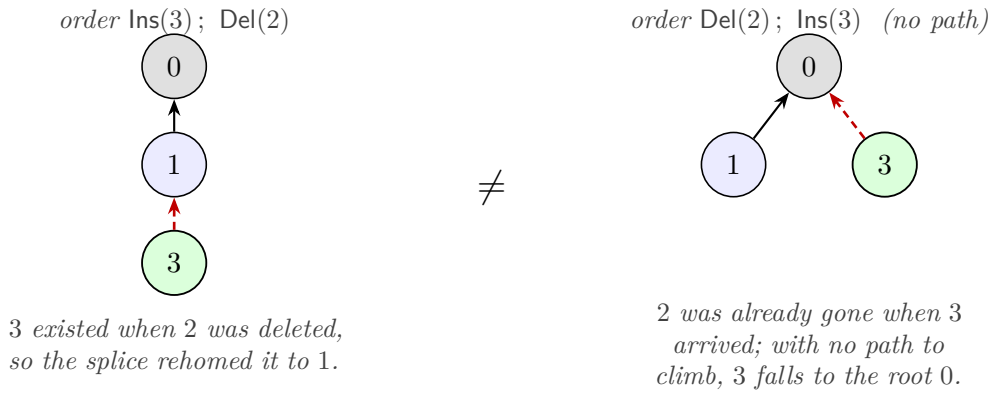

Figure 4: **Divergence without the path.** The same insert/delete pair, two orders, two different trees: 3 is anchored at 1 on the left but at 0 on the right. The information "2's parent is 1" was destroyed by the tombstone-free delete, and nothing on the right carries it. Mechanised as `prefixfree_diverges`.

The fix (carry the path). Give the insert the ancestor path of its anchor, $\mathsf{Ins}(C, [1], 2)$. Now when 2 is dead, resolve climbs the carried path [1] to the live 1, and both orders land 3 under 1 (Figure 5). The path is exactly the ancestry that the tombstone-free state threw away; carrying it on the operation restores commutation.

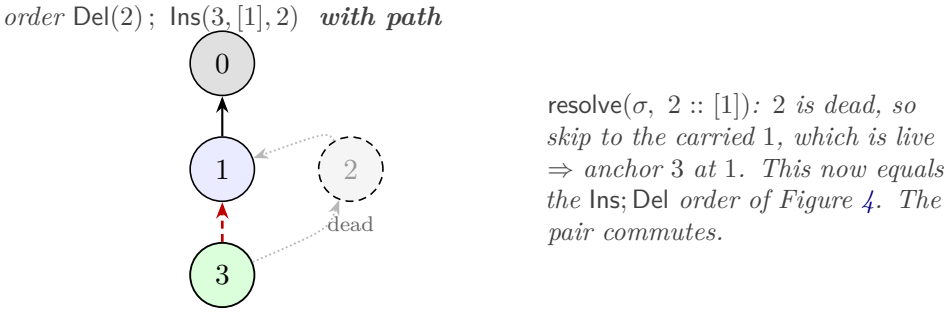

Figure 5: **The path restores convergence.** The carried path [1] lets the late insert climb past the deleted 2 to the live 1, matching the other order. Mechanised as `prefix_converges`.

The slogan. *A tombstone keeps the deleted node's parent in the **state**; the path keeps it on the **operation**.* Remove the tombstone and that parent has to ride along on any operation that might still need it — and on a single replica (no LCA) the insert is exactly such an operation. This is also why the two are *mutually exclusive but jointly unavoidable for the proof*: to satisfy the do-commutation VC you may drop the tombstone or drop the path, but not both. Dropping both defeats that route — provably (`RGA_PrefixFree_Impossible.lean`); the runnable witness that drops both yet still converges *via merge* is `RGA_PrefixFree_Impl.lean`.

4 The path is ghost state: needed for the proof, not for execution

The previous section's "necessary" is a statement about the *proof obligation*, and its scope is worth being exact about: in a real execution the path is **never read**. It is *ghost state* — pure proof

scaffolding that a runtime implementation can drop entirely.

Why it is never read comes down to how operations are generated. A client issues an operation against **its own local replica**, on the state it currently sees; you insert after, or delete, a node that is *live in front of you*. On any state where the anchor is live, `resolve` short-circuits on that anchor and never inspects the prefix — exactly what `ins_path_free` and `del_path_free` prove. So for every operation a client can actually *issue*, the path is dead code. The one state that reads it — an insert whose anchor was *already deleted on the same replica* — is manufactured only by the do-commutation VC, which quantifies over *all* states and, in forcing the two orders to agree, constructs exactly that intermediate. It sits **outside the reachable set** (Figure 6).

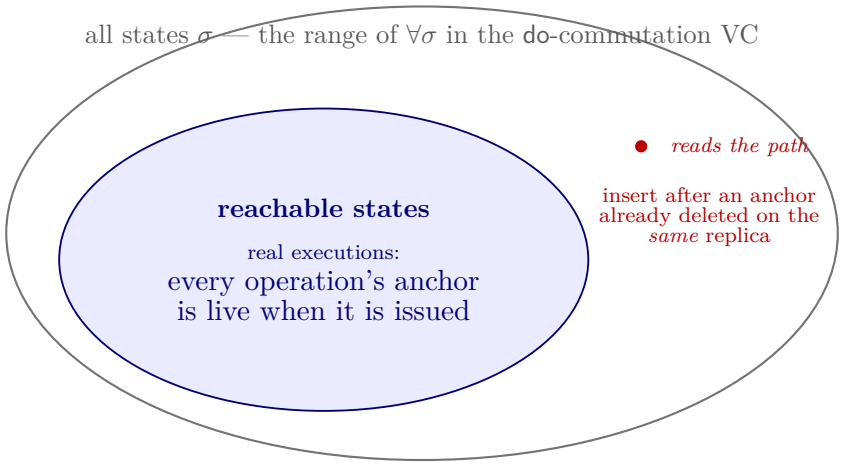

Figure 6: The path is ghost state. The do-commutation VC ranges over *all* states; real executions live in the shaded *reachable* region, where every operation's anchor is live and `resolve` short-circuits before the prefix (`ins_path_free` / `del_path_free`). The path is consulted only at the red state — an insert whose anchor was already deleted on its own replica — which a voluntary, pull-based merge never produces.

Why the system model rules it out. Merge here is **voluntary**, pull-based — like `git pull`. A replica's state does not change under its feet: remote updates are incorporated only when the replica *chooses* to merge, at well-defined points, never in the middle of forming an operation. So the anchor a client references is live when the operation is issued and stays live until it is applied locally; the dead-anchor state never arises on the replica that generated the op. And even in a model where remote state *could* interleave, the client can **reject an operation whose anchor is no longer present** — a cheap local check — so the problematic state is never entered.

The upshot is a clean separation. The *runtime* operations carry no path: Ins needs only its immediate anchor, Del only its target, and the merge recovers position from the LCA (§2). The path exists solely to discharge the framework's universally-quantified commutation VC over states that no execution produces. It is ghost state — read by the proof, erased at run time.

5 The asymmetry: Ins needs the path at run time, Del in the proof

Strikingly, only the *insert* needs the path *at run time*. A delete can always recompute its target from the live state, because at the moment a delete runs, the node it reparents *to* (its target's

parent) is still present. Figure 7 contrasts the two: the delete reads its answer from a node that is alive *now*; the insert may need a node that was alive *then* but is gone *now*.

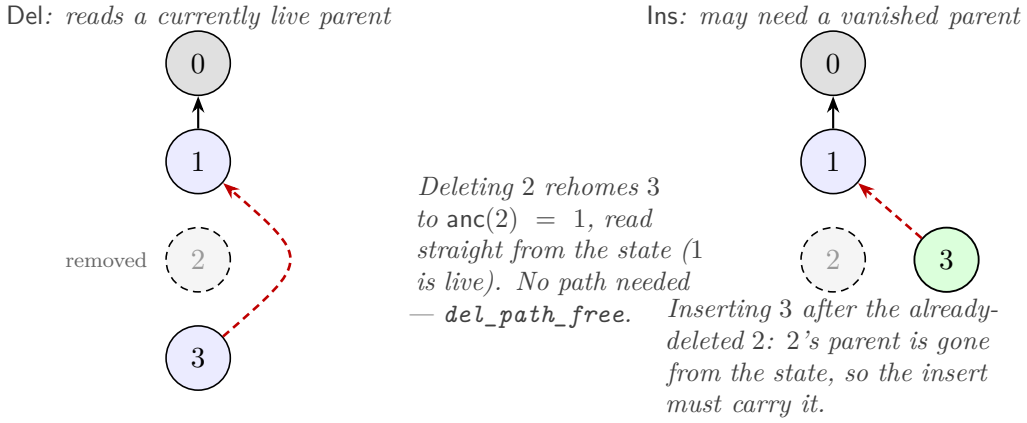

Figure 7: The asymmetry. Del hands off children to a parent that is still live at delete time, so it reads from state; a late Ins references a node that may have been spliced away, so it must carry the path. (`del_path_free` / `prefixfree_diverges`.)

The intuition is temporal: a delete reparents *all current children* the instant it runs, keeping the forest connected; a concurrent (or reordered) insert is a child that was *not present* at that instant, so it missed the handoff and must supply its own ancestry.

But the delete path is not idle scaffolding. That a delete *can* recompute its target does not make the path it carries inert in the proof — and this is where the asymmetry is genuinely two-sided rather than "insert yes, delete no." In the conditioned framework the recorded prefix on a delete is its *operation-static well-formedness certificate*, and it is what keeps a delete's admissibility *independent of the state invariant*. The operation-only side condition on $\mathsf{Del}(p, x)$ is $x \notin p$; since $\mathsf{resolve}(\sigma, p)$ always lands in $\{0\} \cup p$, this forces $\mathsf{resolve}(\sigma, p) \neq x$ — "the node does not reparent onto itself" — from the *operation alone*, with no appeal to any property of σ. That is exactly what lets a delete's invariant preservation be discharged locally: `RgaInv_doDel_opOK` re-establishes the forest invariant across a Del *given only* $x \notin p$, no path accuracy and no state hypothesis.

Drop the path and this decoupling collapses. The reparent target becomes $\mathsf{anc}(\sigma, x)$, a function of the *state*, so "it does not reparent onto itself" is no longer operation-static: it must be re-sourced from the state invariant `id_mono` ($\mathsf{anc}(\sigma, x) < x$, whence the anchor forest is acyclic) and threaded through every invariant-preservation and commutation obligation. The do-commutation verification condition itself — previously an equation guarded only by accuracy and freshness — must then be strengthened with the very forest-acyclicity and delete-applicability premises the path had supplied for free. Removing the delete path is therefore not a simplification of the data type but a *migration* of the delete's well-formedness out of the operation layer and into the reachable-state invariant.

So both operations carry a path, for *different* reasons, and that is the real asymmetry. The insert needs it *semantically*: single-replica commutation genuinely fails without the vanished ancestor, and no state invariant can substitute (§3, §7). The delete needs it *architecturally*: runtime-dispensable (`del_path_free`) yet proof-load-bearing — the static certificate that decouples an operation's admissibility from the reachable-state invariant it would otherwise have to lean on.

6 A read-side consequence: the delete reorders survivors, and the proof cannot see it

The delete's rehoming (§5) buys convergence, but it has a second consequence the run-time picture hides: it can *reorder the surviving elements*. A tombstone-free delete physically splices its target out and rehomes the orphaned subtree, so a child can migrate to a position where the newest-first sibling tie-break ranks it *ahead* of an element it previously followed. This is not a concurrency artefact — it happens on a single replica, in a plain sequential run — and, crucially, the correctness theorem cannot register it.

A minimal witness. Three inserts and one delete. Event 1 inserts a after the root, event 2 inserts b after the root, event 3 inserts c after a. The same-anchor siblings a, b read newest-first, so b precedes a, and c sits under a: the document reads $[b, a, c]$ (Figure 8, left). Now delete a. Correct list semantics — and a tombstoned RGA — leave the survivors in place and read $[b, c]$; the tombstone-free RGA reads $[c, b]$. Splicing a out rehomes c to the root, where c (event 3) out-ranks b (event 2) under the same newest-first tie-break and jumps ahead of it. A tombstoned RGA keeps a as a dead position-holder, so c never migrates and the order is preserved — the physical splice cannot offer that.

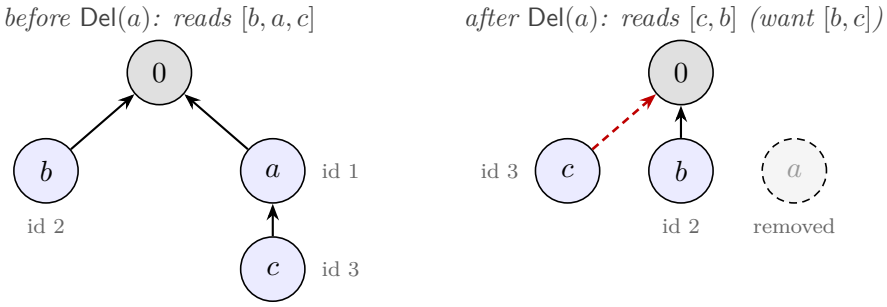

Figure 8: The delete reorders survivors. **Left:** a (id 1) and b (id 2) are root siblings read newest-first (b first), with c (id 3) under a; the document is $[b, a, c]$. **Right:** deleting a splices it out and rehomes c to the root (red dashed), where c out-ranks b and now reads first — $[c, b]$, not the list-correct $[b, c]$. A tombstoned RGA would keep a as a dead position-holder, so c never migrates. (`tombstone_free_violates_delete_order` vs. `remove_preserves_visible_lt`.)

This is machine-checked as a fail/pass pair. The tombstone-free RGA *refutes* an independent "delete preserves survivor order" property (`tombstone_free_violates_delete_order`, i.e. $\neg$ `DeleteOrderPreserving`), via exactly this $[b, a, c] \to [c, b]$ run, in `Sal/MRDTs/RGA/RGA_Tombstone_Free_SPOT.lean`; the tombstoned RGA's order relation `visible_lt`, invariant under a delete that touches only the tombstone set and not the insert records the read consults, *satisfies* the same property (`remove_preserves_visible_lt`, in `Sal/MRDTs/RGA_with_tombstones/RGA_ReadSide.lean`).

Why the verification does not catch it. It is tempting to file this as a mere read-side cost of dropping tombstones. It is more than that: it changes the *certified sequence*, and the correctness theorem is structurally unable to see the change. The end-to-end result is RA-linearizability — every reachable and merged state equals the fold, under the data type's *own* `do`, of some delivery-respecting linearisation of its events. The reorder is baked into that `do`, so the fold and the merge

agree on the *same* reordered sequence $[c, b]$; convergence to the own-fold holds, and the theorem certifies exactly it. What the theorem does *not* certify is that $[c, b]$ is the sequence a sequential list would produce — that is an independent *intent* obligation, and against it this run is a visible failure. So this is a point distinct from the two-sided path asymmetry of §5: not about which operation must carry ancestry, but about the reach of the correctness statement itself. Kernel-clean plus RA-linearizable does *not* mean the single-replica sequence is the one list semantics prescribes. It is the cleanest instance of the self-referential-spec limit recorded in the companion note (`mrdt-metatheory.pdf`, Open Question "the self-referential-spec limit"): it arrives with no concurrency and no marks, sharper than that note's earlier Peritext example.

7 Why no unconditioned proof exists

The path fixes the counterexample of §3; it is natural to hope the fixed design then discharges the standard framework schema as-is — the flat verification conditions, in which *commutation is an equation over all states*: for every concurrent pair left unordered by the reconciliation policy rc, $\mathsf{do}(\mathsf{do}(\sigma, a), b) = \mathsf{do}(\mathsf{do}(\sigma, b), a)$ for *every* σ, with the policy itself constrained (every non-commuting concurrent pair must be rc-oriented, and rc-edges must not chain). That hope fails twice, in instructive ways, and the failures are why this data type is verified through the *conditioned* framework instead.

Attempt 1: no path at all (the splice predecessor). The original tombstone-free design was the path-free splice RGA (`RGA_Splice`, parked on branch `wip/rga-splice`): an Ins carried only its immediate anchor, a Del only its target, and delete physically spliced the node out. Physical splice destroys the deleted node's parent pointer, so $\mathsf{Ins}(t \text{ after } x)$ and $\mathsf{Del}(x)$ do not commute: whether the insert ran before the splice decides whether t gets rehomed onto $\mathsf{anc}(x)$ or left dangling off the vanished x. The schema requires every non-commuting concurrent pair to be oriented, so the splice design's rc *must* contain the entry

$$\mathsf{rc}\big(\mathsf{Ins}(t \text{ after } x),\ \mathsf{Del}(x)\big) \ = \ \mathsf{Fst\_then\_snd} \qquad \text{(the insert is sequenced first).}$$

And a non-Either entry awakens a VC that is vacuously true for every data type whose rc is everywhere Either — the *conditional commutation base*. In the schema it reads: for every state σ and distinct operations o_1, o_2, o_3 with $\mathsf{rc}(o_1, o_2) = \mathsf{Fst\_then\_snd}$ and $\mathsf{rc}(o_2, o_3) \neq \mathsf{Either}$,

$$\mathsf{do}(\mathsf{do}(\mathsf{do}(\sigma, o_1), o_2),\ o_3) \ \equiv \ \mathsf{do}(\mathsf{do}(\mathsf{do}(\sigma, o_2), o_1),\ o_3) \qquad (\texttt{cond\_comm\_base})$$

— an rc-ordered pair must still be order-*insensitive* at application time whenever a third operation also conflicts with o_2: the linearisation machinery realises the mandated order by flipping adjacent applications, and this equation is what licenses the flip.

For the splice design the hypothesis is satisfiable, and the conclusion is false, on a three-operation trace (mechanised as `cond_comm_base_violated`, by evaluation). Start from $\sigma_0 = \{(5, \text{'S'}, 0)\}$ — the single record 5 anchored at the root — and take

$$o_1 = \mathsf{Ins}(10 \text{ after } 5), \qquad o_2 = \mathsf{Del}(5), \qquad o_3 = \mathsf{Ins}(12 \text{ after } 5).$$

The hypothesis holds: $\mathsf{rc}(o_1, o_2) = \mathsf{Fst\_then\_snd}$ (insert-after-the-deleted) and $\mathsf{rc}(o_2, o_3) = \mathsf{Snd\_then\_fst} \neq \mathsf{Either}$. Now run both sides (Figure 9). Left, $o_1; o_2; o_3$: the anchor 5 is still live when $\mathsf{Ins}(10 \text{ after } 5)$ runs, so 10 lands under 5; the splice of 5 then rehomes 10 onto $\mathsf{anc}(5) = 0$; finally $\mathsf{Ins}(12 \text{ after } 5)$ finds no 5 and dangles. Right, $o_2; o_1; o_3$: the splice runs first, so *both* inserts

find no 5 — 10 and 12 both dangle. The two final states differ at identity 10: live under the root on the left, dangling off the vanished 5 on the right. No re-orientation of rc escapes: the non-commuting pair forces *some* non-Either entry, and any such entry makes the VC non-vacuous with the same trace shape refuting it.

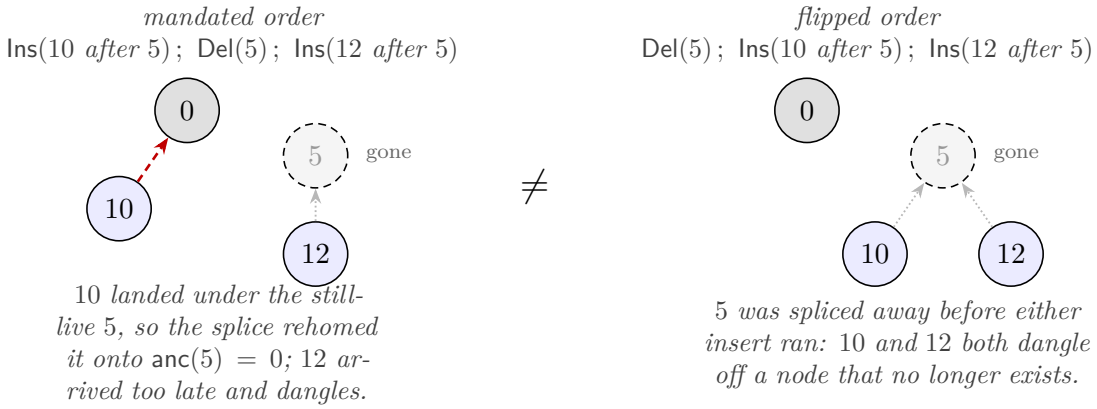

Figure 9: **The splice design refutes `cond_comm_base`.** The three-operation trace $o_1 = \mathsf{Ins}(10 \text{ after } 5)$, $o_2 = \mathsf{Del}(5)$, $o_3 = \mathsf{Ins}(12 \text{ after } 5)$ from $\sigma_0 = \{(5, \text{'S'}, 0)\}$ satisfies the VC's hypothesis ($\mathsf{rc}(o_1, o_2) = \mathsf{Fst\_then\_snd}$, $\mathsf{rc}(o_2, o_3) \neq \mathsf{Either}$), yet the two application orders diverge at identity 10: rehomed onto the root (left) versus dangling off the deleted 5 (right). Mechanised as `cond_comm_base_violated` on branch `wip/rga-splice`.

Attempt 2: the path — which commutes only on truthful states. The path-carrying design of this note repairs Attempt 1's trace: the very same three operations, now carrying their (empty, and for this state *accurate*) prefixes, converge in both orders — 5 hangs off the root, so after $\mathsf{Del}(5, \langle\rangle)$ each $\mathsf{Ins}(\cdot \text{ after } 5, \langle\rangle)$ climbs its carried prefix and lands deterministically at 0 (`trio_converges`). But look at what the mechanisation actually discharges. The schema's commutation VC (`rc_non_comm`) demands, for every pair the policy leaves unordered, commutation over *all* states:

$$\mathsf{rc}(o_1, o_2) = \mathsf{Either} \iff \forall \sigma,\ \mathsf{do}(\mathsf{do}(\sigma, o_1), o_2) \equiv \mathsf{do}(\mathsf{do}(\sigma, o_2), o_1).$$

What the tombstone-free RGA proves is `rc_non_comm'`, whose right-hand side is *reachability-conditioned* commutation (`commutes_with'`):

$$\forall \sigma,\quad \underbrace{\mathsf{contains}(\sigma, 0) = \mathsf{false}}_{\text{root not stored}} \to \underbrace{\mathsf{accurate}(o_1, \sigma) \to \mathsf{accurate}(o_2, \sigma)}_{\text{carried paths are true in } \sigma} \to \underbrace{\mathsf{fresh}(o_1, \sigma) \to \mathsf{fresh}(o_2, \sigma)}_{\text{fresh ids}}$$
$$\to\ \mathsf{do}(\mathsf{do}(\sigma, o_1), o_2) \equiv \mathsf{do}(\mathsf{do}(\sigma, o_2), o_1),$$

where $\mathsf{accurate}(o, \sigma)$ says the op's carried prefix is the genuine root-ward ancestor chain of its leaf *in* σ. Those hypotheses are not decoration: drop them and the equation is false, mechanised as `inconsistent_diverges` (Figure 10). Take $\sigma_1 = \{(7, \text{'F'}, 0),\ (5, \text{'S'}, 7)\}$ — 7 under the root, 5 under 7 — and the pair

$$\begin{aligned} \mathsf{good\_ins} &= \mathsf{Ins}(10 \text{ after } 5,\ \langle 7\rangle) && \text{(accurate: 5's chain is } \langle 7\rangle\text{)}, \\ \mathsf{bad\_del} &= \mathsf{Del}(5,\ \langle\rangle) && \text{(a lie: claims 5 hangs off the root)}. \end{aligned}$$

Whichever operation runs *second* consults *its own* record: in the order good_ins; bad_del, node 10 lands under the live 5 and the delete then rehomes it along the delete's *lying* prefix, onto 0; in the order bad_del; good_ins, the anchor is already gone and the insert climbs its *honest* prefix, landing at 7. Two orders, two sources of truth, two trees. And no local check can exclude this: whether a carried path is accurate is a property of the *execution* that generated the operation, not of the state it is applied to — do cannot tell an honest record from a plausible lie.

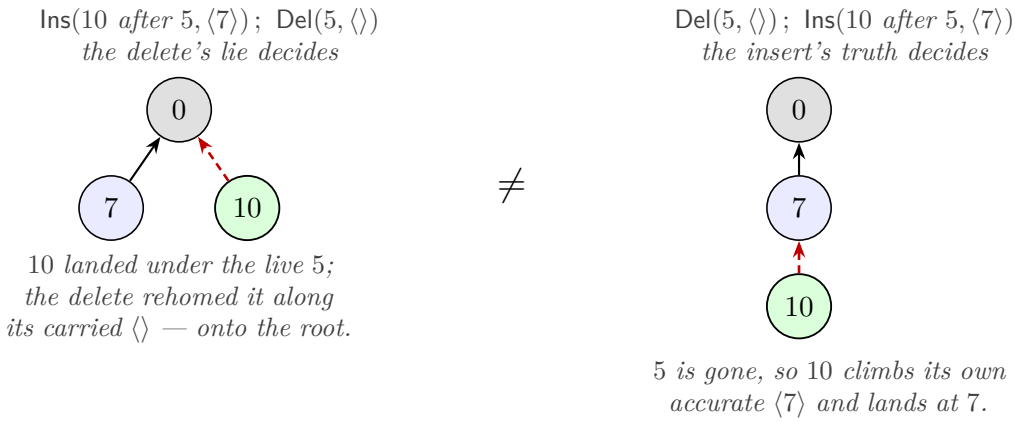

Figure 10: **A lying path breaks unconditioned commutation.** State σ_1: 7 under 0, 5 under 7. The insert's prefix $\langle 7 \rangle$ is accurate; the delete's $\langle\rangle$ lies. Left order: 10 ends under 0 (the delete's record). Right order: 10 ends under 7 (the insert's record). The schema's $\forall\sigma$ ranges over such payload/state pairs, so `rc_non_comm` is unprovable; `rc_non_comm'` must and does assume accurate. Mechanised as `inconsistent_diverges`.

Why this cannot be patched inside the schema. Two escape routes suggest themselves, and both are closed. *Weakening the VC* — baking "accurate, fresh, root-free" into the commutation statement — silently changes the schema: the framework's soundness theorem was proved for the original VCs, so the modified bundle discharges nothing until that soundness proof is redone (this project's costliest lesson; the redone proof *is* the conditioned framework). *Re-engineering the operations* — finding some tombstone-free design whose operations commute unconditionally — is refuted in general: `RGA_PrefixFree_Impossible.lean` shows no tombstone-free, prefix-free, local design can satisfy the rc = Either commutation VC, and ordering the insert/delete pair instead founders on rehoming non-transitivity (`RGA_DepComp_Gate`).

The resolution. The side conditions are promoted to first-class citizens: an *invariant* on states, an *applicability* discipline on operation generation (a well-behaved replica only issues an Ins whose recorded path is its actual view — which is all any honest replica can do), and commutation proved *conditioned* on both, propagated along reachability under honest delivery. That is the conditioned MRDT framework (`Sal/ConditionedMRDTs/`; companion note `mrdt-metatheory.pdf`), and this data type is its founding instance: the end-to-end theorem (`rga_ra_linearizable3_eq`) is RA-linearizability up to observational equivalence at every reachable configuration under honest delivery. The price of leaving the unconditioned schema is real — the tombstoned RGA discharges the flat VCs in ∼170 lines, while this data type's conditioned chain runs to ∼10,000 — and, by the results above, unavoidable.

8 What is mechanised

Every figure corresponds to a Lean fact in `Sal/MRDTs/RGA/`:

- `prefix_converges` / `prefixfree_diverges` — Figures 4–5: with the path the pair commutes; drop it and the same accurate pair diverges. The *insert* path is necessary.
- `del_path_free` / `deldel_pathfree_converges` — Figure 7: the *delete* path is runtime-dispensable (a Del reads its reparent target $\mathsf{anc}(\sigma, x)$ from the live state) yet proof-load-bearing — `RgaInv_doDel_opOK` discharges the delete's invariant preservation from the operation-static $x \notin p$ alone, with no state hypothesis (§5).
- `rc_non_comm'` — the full commutation VC: on accurate, fresh-id, root-not-stored states, *every* operation pair commutes.
- `RGA_PrefixFree_Impossible.lean` — a parameterised theorem: no tombstone-free, prefix-free, local design can satisfy the $\mathsf{rc} = \mathsf{Either}$ commutation VC.
- Figure 9 — the splice predecessor's refuted conditional-commutation VC: `cond_comm_base_violated`, branch `wip/rga-splice` (`Sal/MRDTs/RGA_Splice/RGA_Splice_MRDT.lean`); the trace and both rc verdicts are checked by evaluation. Figure 10 — `inconsistent_diverges` (and `trio_converges`: the path repairs Attempt 1's exact trace); the schema-level content is that `rc_non_comm'`'s accuracy hypothesis cannot be dropped, which is what forces the conditioned framework (§7).
- `RGA_PrefixFree_Impl.lean` — the runnable counterpart of Figures 3 and 4: its *merge* converges the concurrent case (LCA), yet its single-replica do diverges (`merge_converges_concurrent` vs. `single_replica_do_diverges`).

One honest caveat. The merge's climb is bounded by a fuel equal to the node id, so its convergence relies on *id-monotone* anchors ($\mathsf{anc}(t) < t$). That holds under monotone timestamp allocation but is *not* guaranteed by the forest shape alone; a non-monotone state can make climb run out of fuel under merge (`merge_breaks_wf` in `RGA_Reachability_Invariant.lean`). So Figure 3's convergence is a theorem about reachable, monotone states — the precise extra condition the soundness argument must carry.