

Reactive Dead Code Elimination (DCE) as Reducers and Mappers

1 Problem

Given a distributed program graph split across files, each file contributes:

- nodes (definitions),
- roots (entry points that must stay live),
- edges (references / call edges between nodes).

A node is *dead* if it is not reachable from any root in the global graph. Files change over time via + (additions) and - (removals) of nodes/roots/edges. We want a two-layer reactive service:

1. Aggregate all file fragments into a global graph.
2. Maintain DCE results incrementally as files change.

2 Layer 1: Aggregating the distributed graph

Use a per-file **mapper** that turns a file into a fragment (nodes, roots, edges). Use a **reducer** to combine fragments into a global graph.

State type: multisets of nodes, roots, edges:

$$G = (\text{nodes} : \mathcal{M}(\text{Node}), \text{roots} : \mathcal{M}(\text{Node}), \text{edges} : \mathcal{M}(\text{Node} \times \text{Node}))$$

Reducer:

$$\begin{aligned} \iota &= (\emptyset, \emptyset, \emptyset) \\ G \oplus f &= (G.\text{nodes} + f.\text{nodes}, G.\text{roots} + f.\text{roots}, G.\text{edges} + f.\text{edges}) \\ G \ominus f &= (G.\text{nodes} - f.\text{nodes}, G.\text{roots} - f.\text{roots}, G.\text{edges} - f.\text{edges}) \end{aligned}$$

Because we use multisets, $(G \oplus f) \ominus f = G$ (“remove undoes add”). In the general reducer calculus (`reduce.tex`, `lean-formalisation/Reduce.lean`) a reducer is considered well-formed when *both* of the following hold:

- pairwise commutativity of the add/remove operations (order-independence of folding adds/removes);
- an inverse law: removing a just-added fragment restores the prior state.

These algebraic properties are established abstractly in `reduce.tex`; in the DCE setting, they amount to the obvious commutativity/associativity of multiset union/subtraction and the cancellation law $(G \oplus f) \ominus f = G$.

3 Layer 2: Incremental DCE over the global graph

Given the aggregated graph G , define:

- E = deduped edges as a finite set of pairs.
- R = deduped roots.
- live = nodes reachable from R via E .
- $\text{dead} = G.\text{nodes} \setminus \text{live}$.

An incremental online algorithm (conceptual):

- Maintain per-node **refcounts** of live in-edges: $\text{liveIn}(v) = |\{(u, v) \in E \mid u \in \text{live}\}|$.
- On + of a root, seed BFS and propagate liveness, bumping refcounts.
- On + of an edge (u, v) : if u is live, increment $\text{liveIn}(v)$; if it was zero, mark v live and propagate.
- On - of an edge (u, v) : if u is live, decrement $\text{liveIn}(v)$; if it hits zero, v may become dead and you recursively retire its outgoing edges.
- On +/- of a node: add/remove its incident edges and refcount entry; if it is a root, treat as root add/remove.

The refcount discipline ensures \ominus is the inverse of \oplus for liveness when all incident updates are processed. If a refcount would go negative (e.g. inconsistent deletes), fall back to recomputing from scratch.

Delta and complexity. Let Δ be a change to the aggregated graph G (a multiset of file deltas). The fragment reducer (Layer 1) processes Δ in $O(|\Delta|)$ work and produces an updated G' where

$$G' = \begin{cases} G \oplus \Delta & \text{if add} \\ G \ominus \Delta & \text{if remove.} \end{cases}$$

For Layer 2:

- + root: touches only nodes reachable from that root along currently live edges; worst-case $O(|E|)$ but typically proportional to the reachable slice Δ_{live} .
- + edge (u, v) : $O(1)$ if u is dead or v already live; otherwise marks v live and propagates to v 's reachable slice.
- - edge (u, v) : $O(1)$ if u is dead or $\text{liveIn}(v) > 1$; otherwise may cascade along the subgraph reachable from v until nodes with alternate incoming live edges halt the cascade.
- +/- node: reduce to adds/removes of its incident edges and root status.

Thus the reactive work is bounded by the size of the affected reachable component; in the worst case it is linear in $|E|$, but it is *delta-bounded* to the portion of the graph whose liveness actually changes. The output delta (changes in live/dead sets) is similarly bounded by the size of that affected slice.

4 Incremental DCE Algorithm

The state maintained by Layer 2 consists of:

- $\text{live} \subseteq \text{Node}$: the set of currently live nodes.
- $\text{refcount} : \text{Node} \rightarrow \mathbb{N}$: for each node v , the count of incoming edges from live nodes.

4.1 Adding a Fragment

When a fragment $f = (\text{nodes}_f, \text{roots}_f, \text{edges}_f)$ is added:

Algorithm 1 AddFragment(f)

```

1: precondition:  $\text{live} = \text{liveSet}(G)$  and  $\text{refcount} = \text{refcountSpec}(G)$ 
2:  $\text{frontier} \leftarrow \emptyset$  ▷ Nodes to propagate liveness from
3: for each  $r \in \text{roots}_f$  do ▷ inv:  $\text{frontier} = \text{processed new roots not yet live}$ 
4:   if  $r \notin \text{live}$  then
5:      $\text{frontier} \leftarrow \text{frontier} \cup \{r\}$ 
6:   end if
7: end for
8: for each  $(u, v) \in \text{edges}_f$  do ▷ inv: refcounts updated for processed edges from live sources
9:   if  $u \in \text{live}$  then
10:     $\text{refcount}[v] \leftarrow \text{refcount}[v] + 1$ 
11:    if  $v \notin \text{live}$  then
12:       $\text{frontier} \leftarrow \text{frontier} \cup \{v\}$ 
13:    end if
14:  end if
15: end for
16:  $\text{PROPAGATELIVENESS}(\text{frontier})$ 

```

Algorithm 2 PropagateLiveness(frontier)

```

1:  $Q \leftarrow \text{frontier}$  ▷ BFS queue
2: while  $Q \neq \emptyset$  do
▷ inv:  $\text{live} \cup Q \supseteq$  nodes reachable from frontier
▷ inv:  $\forall v \in \text{live}$ .  $\text{refcount}$  includes  $v$ 's outgoing edges
3:    $v \leftarrow Q.\text{dequeue}()$ 
4:   if  $v \notin \text{live}$  then
5:      $\text{live} \leftarrow \text{live} \cup \{v\}$ 
6:     for each  $(v, w) \in G.\text{edges}$  do ▷ All outgoing edges
7:        $\text{refcount}[w] \leftarrow \text{refcount}[w] + 1$ 
8:       if  $w \notin \text{live}$  then
9:          $Q.\text{enqueue}(w)$ 
10:      end if
11:    end for
12:  end if
13: end while
14: postcondition:  $\text{live}$  includes closure of  $\text{frontier}$ ; refcounts updated

```

4.2 Removing a Fragment

When a fragment f is removed, we must handle cascading deaths:

Algorithm 3 RemoveFragment(f)

```

1: precondition: live = liveSet( $G$ ) and refcount = refcountSpec( $G$ )
2: cascade  $\leftarrow \emptyset$  ▷ Nodes that may become dead
3: for each  $(u, v) \in \text{edges}_f$  do ▷ inv: refcounts decremented for processed removed edges
4:   if  $u \in \text{live}$  then
5:     refcount[ $v$ ]  $\leftarrow$  refcount[ $v$ ] - 1
6:     if refcount[ $v$ ] = 0  $\wedge$   $v \notin (G.\text{roots} - \text{roots}_f)$  then
7:       cascade  $\leftarrow$  cascade  $\cup \{v\}$ 
8:     end if
9:   end if
10: end for
11: for each  $r \in \text{roots}_f$  do ▷ inv: processed removed roots with refcount 0 in cascade
12:   if  $r \in \text{live} \wedge r \notin (G.\text{roots} - \text{roots}_f)$  then
13:     if refcount[ $r$ ] = 0 then
14:       cascade  $\leftarrow$  cascade  $\cup \{r\}$ 
15:     end if
16:   end if
17: end for
18: CASCADEDEATH(cascade)

```

Algorithm 4 CascadeDeath(cascade)

```

1:  $Q \leftarrow \text{cascade}$ 
2: while  $Q \neq \emptyset$  do
▷ inv: live  $\supseteq$  liveSet( $G'$ ) (live only shrinks toward spec)
▷ inv: nodes in  $Q$  have refcount 0 and may need removal
▷ inv: for removed nodes, successors' refcounts decremented
3:    $v \leftarrow Q.\text{dequeue}()$ 
4:   if  $v \in \text{live} \wedge \text{refcount}[v] = 0 \wedge v \notin G.\text{roots}$  then
5:     live  $\leftarrow \text{live} \setminus \{v\}$ 
6:     for each  $(v, w) \in G.\text{edges}$  do ▷ Decrement successors
7:       refcount[ $w$ ]  $\leftarrow$  refcount[ $w$ ] - 1
8:       if refcount[ $w$ ] = 0  $\wedge$   $w \notin G.\text{roots}$  then
9:          $Q.\text{enqueue}(w)$ 
10:      end if
11:    end for
12:   end if
13: end while
14: postcondition: all unreachable nodes removed from live

```

Correctness invariant. Let G denote the current aggregated graph state (after Layer 1 applies the fragment delta). After each call to ADDFRAGMENT or REMOVEFRAGMENT, the algorithm state

(live, recount) should satisfy:

$$\text{live} = \{v \mid v \text{ is reachable from } G.\text{roots} \text{ via } G.\text{edges}\}$$

$$\forall v. \text{recount}[v] = |\{(u, v) \in G.\text{edges} \mid u \in \text{live}\}|$$

That is, the algorithm’s live set equals the specification $\text{liveSet}(G)$, and the recount of each node equals the number of incoming edges from live nodes in the current graph.

Relationship to the Lean formalization. The Lean formalization (Section 5) provides:

1. **Specification:** Formal definitions of `liveSet` and `recountSpec` that define what the correct answer is.
2. **Correctness theorem:** Any algorithm maintaining the invariant `refInvariant` produces the specification result (`runRecount_eq_refSpec`).
3. **Characterization theorems:** Proven properties that *describe* how liveness changes:
 - `liveSet_add_as_closure`: $\text{new live} = \text{old live} \cup \text{closure}(\text{frontier})$
 - `liveSet_remove_as_difference`: $\text{new live} = \text{old live} \setminus \text{newlyDead}$
 - `recount_zero_iff_no_live_incoming`: cascade triggers when `recount = 0`
4. **Delta bounds:** Proven bounds on which nodes are affected.

Gap: Algorithm vs. specification. The Lean file does *not* directly verify the pseudocode algorithms above. The concrete step function `recountDeltaStep` in Lean simply recomputes $\text{liveSet}(G')$ from scratch rather than implementing the incremental BFS/cascade logic. The characterization theorems provide strong *guidance* that the algorithms are correct—they show that the new live set is exactly what BFS from the frontier would compute, and that cascade triggers exactly when recount hits zero—but there is no machine-checked proof that the imperative loops implement this specification.

To fully verify the algorithms, one would need to either: (a) formalize the queue-based loops in Lean and prove loop invariants, or (b) use a refinement approach connecting a functional algorithm to the specification. For practical purposes, the characterization theorems provide high confidence that a correct implementation of the pseudocode will satisfy the invariant.

TODO. Port the incremental BFS/cascade algorithm into the Lean step function (replacing `recountDeltaStep`’s recomputation) and prove it preserves `refInvariant`.

5 Lean artefact

The Lean file `lean-formalisation/DCE.lean` formalizes:

Layer 1 (Graph aggregation).

- `Frag`, `GraphState`: multiset-based fragments and global state.
- `addFrag/removeFrag`: the reducer operations.

- `fragReducer` instantiates `Reducer` from `Reduce.lean`; `fragReducer_wellFormed` proves the `WellFormedReducer` law (`remove` undoes `add` on any accumulated state), and `fragReducer_pairwiseComm` shows pairwise commutativity of `addFrag/removeFrag`.
- The general reducer calculus is documented in `reduce.tex` (and `lean-formalisation/Reduce.lean`); `DCE.lean` imports those definitions and instantiates them for fragments.

Layer 2 (Incremental DCE).

- `Reachable`: inductive definition of reachability from roots via edges.
- `liveSet`, `deadSet`: specification of live/dead nodes.
- `refcountSpec`: $\text{liveIn}(v) = |\{(u, v) \in E \mid u \in \text{live}\}|$.
- `refInvariant`: correctness invariant ($\text{live} = \text{liveSet}(G) \wedge \text{refcount} = \text{refcountSpec}(G)$).
- `RefcountAlg`: abstract interface for incremental algorithms with a `preserves` proof obligation.
- `refcountDeltaStep`: concrete step function for processing fragment deltas.
- `refcountDelta_preserves`: proof that the step maintains `refInvariant`.
- `runRefcount_eq_refSpec`: end-to-end correctness—any algorithm preserving the invariant produces the specification result after folding deltas.

BFS characterization for additions.

- `Reachable_mono`: reachability is monotonic (adding edges/roots only expands liveness).
- `liveSet_mono_addFrag`: adding a fragment can only expand the live set.
- `initialFrontierAdd`: the BFS frontier consists of new roots and targets of new edges from live sources.
- `liveSet_add_as_closure`: *proven* that $\text{liveSet}(G') = \text{liveSet}(G) \cup \text{closure}(\text{frontier})$, formalizing the “+ root” and “+ edge” rules from Section 3.

Cascade characterization for removals.

- `liveSet_removeFrag_subset`: removing a fragment can only shrink the live set (anti-monotonicity).
- `newlyDead`: the set of nodes that become dead after removal (were live, now unreachable).
- `liveSet_remove_as_difference`: *proven* that $\text{liveSet}(G') = \text{liveSet}(G) \setminus \text{newlyDead}$.
- `cascade_single_node`: if a node loses all live incoming edges and is not a remaining root, it becomes dead.
- `cascade_propagates`: dead nodes propagate: if v becomes dead, nodes only reachable through v also die.
- `refcount_zero_iff_no_live_incoming`: *proven* that $\text{refcount}(v) = 0$ iff v has no incoming edges from still-live nodes, formalizing when the cascade triggers.

Complexity and delta bounds.

- **newlyLive**: the set of nodes that become live after adding (the “add delta”).
- **newlyDead**: the set of nodes that become dead after removing (the “remove delta”).
- **liveSet_add_eq_union_delta**: $\text{liveSet}(G') = \text{liveSet}(G) \cup \text{newlyLive}$.
- **liveSet_remove_eq_diff_delta**: $\text{liveSet}(G') = \text{liveSet}(G) \setminus \text{newlyDead}$.
- **newlyLive_disjoint_old**: the add delta is disjoint from the old live set.
- **newlyDead_subset_old**: the remove delta is a subset of the old live set.
- **add_delta_bound**: *proven* that $\text{newlyLive} \subseteq \text{closure}(\text{frontier})$, bounding work by the frontier’s reachable set.
- **outside_delta_unchanged**: *proven* that nodes outside the delta have unchanged liveness.
- **totalDelta**: unified characterization of changed nodes for both add and remove.
- **directlyAffected**, **potentiallyDead**: characterization of nodes that may be affected by a removal; **potentiallyDead** is defined as nodes reachable from directly affected nodes.
- **remove_delta_bound**: *proven* that $\text{newlyDead} \subseteq \text{potentiallyDead}$, bounding the remove cascade.
- **refcount_change_bound**: *proven* that refcounts only change for nodes with edges from the delta or edges in the fragment, enabling efficient incremental updates.

These lemmas formalize that the reactive work is *delta-bounded*: only nodes in the delta (and their neighbors for refcount updates) require processing. The frontier for add consists of $O(|f.\text{roots}| + |f.\text{edges}|)$ nodes, and the reachable set from the frontier bounds the actual work. All proofs are complete with no remaining `sorry`s in the Lean formalization.

6 Toward a Skip service

A realistic Skip service can be structured in two layers of resources:

1. **Aggregated graph resource** (Layer 1): an input collection of file fragments `files : File × Frag` mapped directly to a reducer `fragReducer` (multiset union). This yields a single resource `graph : Unit → GraphState` containing the multisets of all nodes/roots/edges.
2. **DCE resource** (Layer 2): a custom compute resource that subscribes to `graph` updates and maintains internal state (`live`, `refcount`). On each graph delta, it runs the incremental refcount algorithm (add/remove roots/edges, propagate liveness, cascade deletions) and emits two derived collections: `live : Unit → Set Node` and `dead : Unit → Set Node`.

In the Skip bindings, Layer 1 is a standard `EagerCollection.reduce` with a well-formed reducer (`addFrag/removeFrag`). Layer 2 is not a pure reducer; it is best implemented as a `LazyCompute` or service module that:

- subscribes to the `graph` resource;

- keeps a refcount map and a live set in memory;
- applies the delta-handling rules from Section 2 (refcount bumps/drops, reachability propagation);
- publishes live/dead as derived resources.

This preserves the algebraic guarantees where they apply (Layer 1) while accommodating the global, graph-shaped logic of DCE in a custom reactive compute node (Layer 2).

7 Layer 2 as an invertible reducer?

Conceptually, Layer 2 consumes graph deltas and maintains liveness. One might ask whether it could be packaged as a single invertible reducer in the sense of `reduce.tex`, with some accumulator A and update operations (\oplus, \ominus) that:

- are pairwise commutative (order-independent) and satisfy the inverse law, and
- from which the current live set can be read off as a projection of A .

There is a trivial positive answer and a negative answer that explains why we do not pursue this design:

- *Trivial positive.* Take A to contain the Layer 1 graph G together with any Layer 2 state, and let \oplus, \ominus be exactly the Layer 1 multiset add/remove on G (with liveness recomputed or updated as a view). This is an invertible reducer, but it is just “Layer 1 reducer + derived view” and does not add any new algebraic structure for liveness.
- *Negative for smaller A .* Any attempt to shrink A to something that only stores Layer 2 summaries (e.g. just live/dead sets and refcounts) forgets information about the graph (such as edges between currently dead nodes) that can affect the liveness of future updates. Two different graphs can induce the same summary state but respond differently to a new fragment, so no single pair of operations (\oplus, \ominus) on summaries can both satisfy the reducer laws and coincide with the graph-based specification for all inputs.

For this reason we keep the split: Layer 1 is an invertible reducer over graph fragments; Layer 2 is a graph algorithm layered on top of that reducer, not an invertible reducer in its own right.