

# 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 factor DCE into two conceptual layers:

1. **Reactive layer:** aggregate all file fragments into a global graph view.
2. **Incremental layer:** maintain DCE results over that graph as files change.

## 2 Layer 1 (Reactive): 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  $(G \oplus f) \ominus f = G$ . Conceptually, Layer 1 is a *reactive view*: the graph state  $G$  is just the accumulator of this well-formed reducer over the current multiset of file fragments, and updates to files are propagated by the generic reactive machinery for reducers.

### 3 Layer 2 (Incremental): DCE over the global graph

Layer 2 sits on top of the reactive graph view  $G$  and is purely incremental: its job is to maintain, under updates, the same live/dead partition we would get by recomputing from scratch. Given the aggregated graph  $G$ , define:

- $E$  = deduped edges as a finite set of pairs.
- $R$  = deduped roots.
- $\text{live}$  = 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  $\Delta$  be a change to the aggregated graph  $G$  (a multiset of file deltas). The fragment reducer (Layer 1) processes  $\Delta$  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  $\Delta_{\text{live}}$ .
- + 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 incremental work in Layer 2 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( $\text{frontier}$ )

---

```

1:  $Q \leftarrow \text{frontier}$  ▷ BFS queue
2: while  $Q \neq \emptyset$  do
▷ inv:  $\text{live} \cup Q \supseteq$  nodes reachable from frontier
▷ inv:  $\forall v \in \text{live}$ . 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, refcount) should satisfy:

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

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

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

## 5 Lean artefact

The Lean development is organized into two layers with `lean-formalisation/DCE.lean` as a thin entry module:

- **Layer 1:** `DCE/Layer1.lean` (reactive graph aggregation).
- **Layer 2:** split into four files under `DCE/Layer2/`:
  - `Spec.lean`: basic definitions (`Reachable`, `liveSet`, `RefState`, `refInvariant`).
  - `Algorithm.lean`: algorithm framework (`RefcountAlg`, `runRefcount`).
  - `Characterization.lean`: BFS/cascade characterization lemmas.
  - `Bounds.lean`: delta bounds and end-to-end correctness.

`DCE/Layer2.lean` re-exports all four sub-modules.

### 5.1 Layer 1 (Reactive graph aggregation)

- `Frag`, `GraphState`: multiset-based fragments and global state, defined in `DCE/Layer1.lean`.
- `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/Layer1.lean` imports those definitions and instantiates them for fragments.

### 5.2 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.

### 5.2.1 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.

### 5.2.2 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.

### 5.2.3 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.

### 5.2.4 Algorithm vs. specification and future work

The current Lean development in `DCE/Layer2/` does not directly formalize the queue-based pseudocode algorithms from Section 4 as imperative loops. Instead, the concrete step function `refcountDeltaStep` recomputes `liveSet( $G'$ )` and `refcountSpec( $G'$ )` for the updated graph  $G'$  and proves that this recompute-based step preserves the invariant `refInvariant`. The characterization and delta-bound theorems in the preceding subsections show that the new live set is exactly the closure of a small frontier and that cascades only affect nodes in a bounded region of the graph; they thus provide strong guidance for implementing a BFS/cascade-style incremental algorithm that would satisfy the same invariant. As future work, one could either formalize the queue-based loops in Lean and prove loop invariants, or follow a refinement approach that derives a functional incremental step from the specification and then relates it to an imperative implementation; in either case, the goal is to replace `refcountDeltaStep`'s recomputation with a proven incremental BFS/cascade algorithm.

These lemmas formalize that the incremental work in Layer 2 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`); this is the reactive layer, where the global graph is maintained as a reducer-backed view of the file-fragment collection. 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 the incremental algorithm (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 incremental compute node layered on top of that reactive reducer (Layer 2).