Reactive Reanalyze: DCE with Dis-aggregation and Incremental Fixpoint

1 Overview

This document provides a mathematical representation of the `reanalyze` dead code elimination algorithm as implemented in a reactive Skip service.

Three-Layer Architecture. The implementation has three distinct layers with different computational models:

Layer	Computation Model	What It Does
1. Skip Runtime	Server mapper + SSE	Dis-aggregation, delta streaming
2. Fixpoint Combinator	`SkipruntimeFixpoint`	Incremental liveness
3. Post-Fixpoint	Pure functions	Optional args, module deadness

Key boundary: Skip combinators are only available in Layers 1–2. Layer 3 uses the fixpoint *result* but runs as ordinary code after each update.

2 Domain Model

2.1 Base Types

$$\begin{aligned} \mathsf{Name} &\quad \text{declaration names (values, types, modules)} \\ \mathsf{ArgName} &\quad \text{optional argument names (e.g., } \texttt{~format}\text{, } \texttt{~locale}\text{)} \end{aligned}$$

2.2 File Data (Server Input)

Each file f provides complete analysis data:

$$\mathsf{fileData}_f = (\mathsf{decls}_f, \mathsf{refs}_f, \mathsf{annot}_f, \mathsf{optArgCalls}_f)$$

where:

$$\begin{aligned} \mathsf{decls}_f &: \mathcal{P}(\mathsf{Name}) && \text{declarations in file } f \\ \mathsf{refs}_f &: \mathcal{P}(\mathsf{Name} \times \mathsf{Name}) && \text{pairs } (\mathsf{target}, \mathsf{source}) \\ \mathsf{annot}_f &: \mathsf{Name} \rightharpoonup \{\mathsf{dead}, \mathsf{live}\} && \text{partial: annotated decls only} \\ \mathsf{optArgCalls}_f &: \mathcal{P}(\mathsf{Name} \times \mathsf{Name} \times \mathcal{P}(\mathsf{ArgName})) && \text{call site info} \end{aligned}$$

For $\mathsf{optArgCalls}$, each element (c, f, A) represents:

- $c \in \mathsf{Name}$: the **caller** (declaration containing the call)
- $f \in \mathsf{Name}$: the **callee** (function with optional args)
- $A \subseteq \mathsf{ArgName}$: the **passed arguments** at this call site

2.3 Fragments (Server Output)

The server dis-aggregates each file into keyed fragments:

$$(f, \text{"decls"}) \mapsto \mathsf{decls}_f \qquad (f, \text{"refs"}) \mapsto \mathsf{refs}_f$$
$$(f, \text{"annot"}) \mapsto \mathsf{annot}_f \qquad (f, \text{"optArgCalls"}) \mapsto \mathsf{optArgCalls}_f$$

3 Layer 1: Skip Runtime (Server + SSE)

3.1 Server Mapper

The server uses a **mapper** to split file data into fragments:

$$\mathsf{disaggregate} : (f, \mathsf{fileData}_f) \mapsto \{(f, t, v) \mid t \in \mathsf{Types}, v = \mathsf{fileData}_f.t\}$$

This produces four output entries per input file.

3.2 Delta Detection

When file f is updated, Skip compares each output fragment's new value to its old value. Only changed fragments are sent to clients via SSE.

Example: If only annotations change, only $(f, \text{"annotations"})$ is sent—not decls, refs, or optArgCalls.

4 Layer 2: Client-Side Combinators

The client uses two combinators to maintain state incrementally:

1. `ClientReducer`: Aggregates data from multiple sources (files)
2. `SkipruntimeFixpoint`: Computes transitive closure (liveness)

4.1 ClientReducer: Incremental Aggregation

A **ClientReducer** aggregates values from multiple sources while tracking provenance:

$$\mathsf{ClientReducer} : (\mathsf{Source} \times V) \to V_{\mathsf{agg}}$$

Key operations:

- $\mathsf{setContribution}(s, vs)$: Set source s's contribution to vs, returns delta
- $\mathsf{current}()$: Get current aggregated value

Semantics: When source s's contribution changes from old_s to new_s:

$$\begin{aligned} \Delta^+ &= \mathsf{new}_s \setminus \mathsf{old}_s && \text{(added by this source)} \\ \Delta^- &= \mathsf{old}_s \setminus \mathsf{new}_s && \text{(removed by this source)} \end{aligned}$$

For multiset semantics (where the same value can come from multiple sources):

$$\begin{aligned} \mathsf{addedToAggregate} &= \{v \in \Delta^+ \mid \mathsf{count}(v) = 0 \to 1\} \\ \mathsf{removedFromAggregate} &= \{v \in \Delta^- \mid \mathsf{count}(v) = 1 \to 0\} \end{aligned}$$

Complexity: $O(|\Delta|)$ per update, not $O(\text{total})$.

4.2 Reducers for DCE

The client maintains four reducers:

$$\begin{aligned} \mathsf{declsReducer} &: \mathsf{File} \to \mathcal{P}(\mathsf{Name}) \to \mathcal{P}(\mathsf{Name}) \\ \mathsf{refsReducer} &: \mathsf{File} \to \mathcal{P}(\mathsf{Name} \times \mathsf{Name}) \to \mathcal{P}(\mathsf{Name} \times \mathsf{Name}) \\ \mathsf{annotReducer} &: \mathsf{File} \to (\mathsf{Name} \rightharpoonup \mathsf{Annot}) \to (\mathsf{Name} \rightharpoonup \mathsf{Annot}) \\ \mathsf{optArgCallsReducer} &: \mathsf{File} \to \mathcal{P}(\mathsf{Call}) \to \mathcal{P}(\mathsf{Call}) \end{aligned}$$

Aggregated views are now derived from reducers:

$$\begin{aligned} \mathsf{allDecls} &= \mathsf{declsReducer.current}() \\ \mathsf{allRefs} &= \mathsf{refsReducer.current}() \\ \mathsf{allAnnotations} &= \mathsf{annotReducer.current}() \\ \mathsf{allOptArgCalls} &= \mathsf{optArgCallsReducer.current}() \end{aligned}$$

4.3 Update Flow

When fragment (f, t) arrives with new value v:

1. $\Delta = \mathsf{reducer}_t.\mathsf{setContribution}(f, v) \quad (O(|\Delta|))$
2. Use Δ to update base/step for fixpoint $\quad (O(|\Delta|))$
3. Apply fixpoint changes $\quad (O(|\Delta| + \text{cascade}))$

Total: $O(|\Delta|)$, not $O(\text{total files})$.

4.4 Fixpoint Combinator

The client uses `SkipruntimeFixpoint` to maintain the live set incrementally.

4.4.1 Base Set

A declaration is in the base if live without needing references:

$$\mathsf{base} = \{d \mid \mathsf{allAnnotations}(d) = \mathsf{live}\} \cup \{d \mid \mathsf{hasExternalRef}(d)\}$$

where:

$$\mathsf{hasExternalRef}(d) \iff \exists s \in \mathsf{refsByTarget}(d).\ s \notin \mathsf{allDecls}$$

4.4.2 Step Edges

Edges propagate liveness, filtered by `@dead`:

$$\mathsf{stepEdges} = \{(s, d) \mid d \in \mathsf{refsByTarget}(s) \wedge \mathsf{allAnnotations}(s) \neq \mathsf{dead}\}$$

4.4.3 Fixpoint

The live set is the least fixpoint:

$$\mathsf{live} = \mu X.\, \mathsf{base} \cup \{d \mid \exists s \in X.\, (s, d) \in \mathsf{stepEdges}\}$$

4.4.4 Incremental Updates

When a reducer delta Δ_{agg} arrives, compute the corresponding fixpoint deltas:

- From Δ_{annot}: compute $\Delta\mathsf{base}^{\pm}$ (live annotations) and $\Delta\mathsf{step}^{\pm}$ (dead blocks)
- From Δ_{refs}: compute $\Delta\mathsf{base}^{\pm}$ (external refs) and $\Delta\mathsf{step}^{\pm}$ (edges)
- From Δ_{decls}: compute $\Delta\mathsf{base}^{\pm}$ (external ref changes)

Then apply:

$$\mathsf{applyChanges}(\Delta\mathsf{base}^{+}, \Delta\mathsf{base}^{-}, \Delta\mathsf{step}^{+}, \Delta\mathsf{step}^{-})$$

The fixpoint combinator handles propagation with cost $O(|\Delta| + \text{cascade})$.

5 Layer 3: Incremental Derived Analyses

Additional analyses depend on the live set. While these don't use Skip combinators, they can be **incrementally updated** using the fixpoint's change notifications.

5.1 Fixpoint Change Notifications

The `applyChanges` method returns which elements changed:

```
type changes = {
  added: array<string>,   // Elements that became live
  removed: array<string>, // Elements that became dead
}
```

This enables incremental updates to derived analyses.

5.2 Optional Arguments (Incremental)

Depends on: live (from fixpoint) + optArgCalls (from fragments)

Aggregated call data:

$$\mathsf{allOptArgCalls} = \bigcup_{f} \mathsf{optArgCalls}_f \quad : \quad \mathcal{P}(\mathsf{Name} \times \mathsf{Name} \times \mathcal{P}(\mathsf{ArgName}))$$

For each function f with optional arguments, the used args (from live callers only) are:

$$\mathsf{usedArgs}(f) = \bigcup_{\substack{(c,f,A) \in \mathsf{allOptArgCalls} \\ c \in \mathsf{live}}} A$$

Key insight: Track provenance to enable incremental updates:

$$\mathsf{usedArgsWithProvenance} : \mathsf{Name} \to \mathsf{ArgName} \to \mathcal{P}(\mathsf{Name})$$

where $\mathsf{usedArgsWithProvenance}(f)(a) = \{c \mid (c, f, A) \in \mathsf{allOptArgCalls} \wedge a \in A \wedge c \in \mathsf{live}\}$.

Incremental update algorithm:

First, build an index by caller:

$$\mathsf{callsByCaller}(c) = \{(f, A) \mid (c, f, A) \in \mathsf{allOptArgCalls}\}$$

Then, when the fixpoint changes:

1. For each $c \in \mathsf{changes.added}$ (caller became live):
   - For each $(f, A) \in \mathsf{callsByCaller}(c)$:
   - For each $a \in A$: add c to $\mathsf{usedArgsWithProvenance}(f)(a)$
2. For each $c \in \mathsf{changes.removed}$ (caller became dead):
   - For each $(f, A) \in \mathsf{callsByCaller}(c)$:
   - For each $a \in A$: remove c from $\mathsf{usedArgsWithProvenance}(f)(a)$

Complexity: $O(|\mathsf{changes}| \cdot \bar{k})$ where $\bar{k}$ is the average calls per declaration.

Implementation:

```
let changes = fixpoint->applyChanges(...)

// Incremental update using fixpoint changes
changes.removed->Array.forEach(removeCallerFromUsedArgs)
changes.added->Array.forEach(addCallerToUsedArgs)
```

5.3 Module-Level Deadness

Depends on: live (from fixpoint) + module membership

A module is dead iff all its declarations are dead:

$$\mathsf{moduleDead}(m) \iff \forall d \in m.\, d \notin \mathsf{live}$$

This can also be updated incrementally using change notifications, but the benefit is smaller (module count is typically much less than declaration count).

6 Example: Adding a New Live Caller

When `feature.res` is added with `@live` annotation and calls `utils(~timezone)`:

1. **Layer 1 (Skip Runtime):** Server sends 4 fragment deltas for the new file
2. **Layer 2 (Fixpoint):**
   - $\Delta\mathsf{base}^+ = \{\mathsf{feature}\}$ (new @live annotation)
   - applyChanges returns $\mathsf{changes.added} = [\mathsf{feature}, \mathsf{dead\_util}]$

3. **Layer 3 (Incremental):**
   - For $\mathsf{feature} \in \mathsf{changes.added}$:
   - Look up $\mathsf{optArgCallsByCaller}(\mathsf{feature}) = [(\mathsf{utils}, [\tilde{}\mathsf{timezone}])]$
   - Add feature to $\mathsf{usedArgsWithProvenance}(\mathsf{utils})(\tilde{}\mathsf{timezone})$
   - Result: ~timezone now marked as used!

Total optional args work: process 1 caller's calls—not recompute from all callers.

7 Combinator Boundary Summary

Component	Combinator	Incremental?
Server dis-aggregation	Skip Mapper	Per-fragment deltas
SSE delta streaming	Skip Runtime	Only changed fragments
Client aggregation	`ClientReducer`	$O(\Delta)$ per fragment
Liveness fixpoint	`SkipruntimeFixpoint`	$O(\Delta + \text{cascade})$
Optional args analysis	(uses `changes`)	Yes, via provenance
Module deadness	(uses `changes`)	Yes (optional)
Issue reporting	—	Full recompute (cheap)

Key insight: `ClientReducer` + fixpoint `changes` enable end-to-end $O(\Delta)$ updates.

8 Performance Characteristics

Operation	Complexity	Layer
Server dis-aggregation	$O(\lvert\mathsf{fileData}\rvert)$ per file	1
Network transfer	$O(\lvert\text{changed fragments}\rvert)$	1
Client aggregation (`ClientReducer`)	$O(\lvert\Delta_{\mathsf{fragment}}\rvert)$	2
Fixpoint delta computation	$O(\lvert\Delta_{\mathsf{agg}}\rvert)$	2
Fixpoint propagation	$O(\lvert\Delta\rvert + \text{cascade})$	2
Optional args update	$O(\lvert\mathsf{changes}\rvert \cdot \bar{k})$	3
Module deadness	$O(\lvert\mathsf{changes}\rvert)$	3

Key wins:

- **End-to-end** $O(\Delta)$: Each layer processes only what changed
- **No full scans**: No iteration over all files or all declarations
- **Composable**: Reducer deltas feed fixpoint deltas feed derived analyses

9 References

- `dce_reactive_view.tex` — Simple DCE model
- `examples/ReanalyzeDCEService.ts` — Server implementation (Layer 1)
- `examples/ReanalyzeDCEHarness.res` — Client implementation (Layers 2–3)
- `reanalyze/src/DeadCommon.ml` — Original batch implementation