Incremental Fixpoint Computation:
A Two-Level Architecture

Abstract

We observe that the incremental dead code elimination (DCE) algorithm from our reactive DCE work is an instance of a more general pattern: *incremental fixpoint computation.* This note proposes a two-level architecture for incremental fixpoints: (1) a low-level API that assumes user-provided incremental operations, and (2) a potential high-level DSL where these operations are derived automatically from a structured definition of the fixpoint operator. The relationship between these levels is analogous to that between manual gradient computation and automatic differentiation.

1 Motivation: DCE as Incremental Fixpoint

In reactive DCE, the live set is defined as the least fixpoint of a monotone operator:

$$F_G(S) = G.\mathsf{roots} \cup \{v \mid \exists u \in S.\, (u, v) \in G.\mathsf{edges}\}$$

That is, $\mathsf{liveSet}(G) = \mathsf{lfp}(F_G)$.

When the graph changes ($G \to G' = G \pm f$), we want to update the fixpoint incrementally rather than recomputing from scratch. The key observations are:

- **Expansion** ($G \to G \oplus f$): The operator grows, so $\mathsf{lfp}(F_G) \subseteq \mathsf{lfp}(F_{G'})$. The old fixpoint is an underapproximation; we iterate upward.
- **Contraction** ($G \to G \ominus f$): The operator shrinks, so $\mathsf{lfp}(F_{G'}) \subseteq \mathsf{lfp}(F_G)$. The old fixpoint is an overapproximation; we must remove unjustified elements.

This pattern—incremental maintenance of a least fixpoint under changes to the underlying operator—arises in many domains beyond DCE.

2 The General Pattern

Definition 1 (Monotone Fixpoint Problem)**.** *Given a complete lattice* $(L, \sqsubseteq)$ *and a monotone operator* $F : L \to L$*, the* least fixpoint *is* $\mathsf{lfp}(F) = \bigcap\{x \mid F(x) \sqsubseteq x\}$*.*

For set-based fixpoints (our focus), $L = \mathcal{P}(A)$ for some element type A, ordered by $\subseteq$, and F is typically of the form:

$$F(S) = \mathsf{base} \cup \mathsf{step}(S)$$

where base provides seed elements and step derives new elements from existing ones.

Definition 2 (Incremental Fixpoint Problem)**.** *Given:*

- *A current fixpoint* $S = \mathsf{lfp}(F)$
- *A change that transforms* F *into* F'

Compute $S' = \mathsf{lfp}(F')$ *efficiently, in time proportional to* $|S' \triangle S|$ *rather than* $|S'|$.

3 Level 1: Low-Level Incremental Fixpoint API

The low-level API assumes the user provides the necessary incremental operations.

3.1 Required Ingredients

For Semi-Naive Expansion. When $F' \supseteq F$ (the operator grows), we use *semi-naive evaluation*:

- Maintain the "delta" ΔS = elements added in the last iteration
- Instead of computing $F'(S)$, compute only $\mathsf{stepFromDelta}(\Delta S) \setminus S$

The user provides:

$$\mathsf{stepFromDelta} : \mathsf{Params} \times \mathcal{P}(A) \to \mathcal{P}(A)$$

Given the current parameters and a delta set, return elements derivable from that delta.

Example 1 (DCE)**.** $\mathsf{stepFromDelta}(G, \Delta) = \{v \mid \exists u \in \Delta.\, (u, v) \in G.\mathsf{edges}\}$

For Counting-Based Contraction. When $F' \subseteq F$ (the operator shrinks), we use *counting-based deletion*:

- Track how many "derivations" support each element
- When a derivation is removed, decrement the count
- When count reaches zero, remove the element and propagate

The user provides:

$$\mathsf{derivationCount} : \mathsf{Params} \times \mathcal{P}(A) \times A \to \mathbb{N}$$

Given the current fixpoint, how many ways is element x derived?

Example 2 (DCE)**.** $\mathsf{derivationCount}(G, \mathsf{live}, v) = |\{u \mid (u, v) \in G.\mathsf{edges} \wedge u \in \mathsf{live}\}|$
This is exactly the refcount *maintained by the DCE algorithm.*

3.2 The API

```
interface IncrementalFixpoint<A, Params, Delta> {
  // User provides:
  base: (params: Params) => Set<A>
  stepFromDelta: (params: Params, delta: Set<A>) => Set<A>
  derivationCount: (params: Params, fp: Set<A>, x: A) => Nat
  applyDelta: (params: Params, delta: Delta) => Params

  // System provides:
  current: Set<A>
```

```
  counts: Map<A, Nat>

  update(delta: Delta): { added: Set<A>, removed: Set<A> }
}
```

3.3 Update Algorithm

Expansion. When F grows:

1. Compute initial delta: $\Delta_0 = F'(\mathsf{current}) \setminus \mathsf{current}$
2. Semi-naive iterate:

$$\Delta_{n+1} = \mathsf{stepFromDelta}(\Delta_n) \setminus \mathsf{current}$$
$$\mathsf{current} \leftarrow \mathsf{current} \cup \Delta_{n+1}$$

3. Update derivation counts for new elements

Contraction. When F shrinks:

1. Update derivation counts for removed derivations
2. Initialize cascade: $Q = \{x \in \mathsf{current} \mid \mathsf{counts}[x] = 0 \wedge x \notin \mathsf{base}\}$
3. Propagate:
   - Remove x from current
   - Decrement counts of elements derived from x
   - Add newly-zero elements to Q

3.4 Formal Definitions and Correctness

3.4.1 Decomposed Operators

Definition 3 (Decomposed Operator)**.** *An operator $F : \mathcal{P}(A) \to \mathcal{P}(A)$ is* decomposed *if $F(S) = B \cup \mathsf{step}(S)$ where B is a fixed base set and* step *is monotone: $S \subseteq T \Rightarrow \mathsf{step}(S) \subseteq \mathsf{step}(T)$.*

Definition 4 (Operator Expansion and Contraction)**.** *We say F* expands to *F', written $F \sqsubseteq F'$, if $\forall S.\, F(S) \subseteq F'(S)$. Dually, F* contracts to *F' if $F' \sqsubseteq F$.*

Theorem 1 (Fixpoint Monotonicity)**.**

1. *If $F \sqsubseteq F'$ then $\mathsf{lfp}(F) \subseteq \mathsf{lfp}(F')$.*
2. *If $F' \sqsubseteq F$ then $\mathsf{lfp}(F') \subseteq \mathsf{lfp}(F)$.*

3.4.2 Semi-Naive Iteration (Expansion)

Definition 5 (Semi-Naive Iteration)**.** *Given a decomposed operator (B, step) and initial set I, define:*

$$C_0 = I \qquad \Delta_0 = I$$
$$C_{n+1} = C_n \cup \Delta_{n+1} \qquad \Delta_{n+1} = \mathsf{step}(\Delta_n) \setminus C_n$$

Theorem 2 (Semi-Naive Monotonicity)**.** *$C_n \subseteq C_{n+1}$ for all $n \geq 0$.*

Theorem 3 (Semi-Naive Soundness)**.** *If $I \subseteq \mathsf{lfp}(F)$, then $C_n \subseteq \mathsf{lfp}(F)$ for all $n \geq 0$.*

3.4.3 Counting-Based Cascade (Contraction)

Definition 6 (Derivation Count). *A function* $\mathsf{count} : \mathcal{P}(A) \times A \to \mathbb{N}$ *is a* valid derivation count *for operator* (B, step) *if:*

$$x \in \mathsf{step}(S) \iff \mathsf{count}(S, x) > 0$$

Definition 7 (Cascade Iteration). *Given operator* (B, step)*, count function* count*, and initial set* I*:*

$$\begin{aligned}
\mathsf{shouldDie}(S) &= \{x \in S \mid x \notin B \wedge \mathsf{count}(S, x) = 0\} \\
\mathsf{cascadeStep}(S) &= S \setminus \mathsf{shouldDie}(S) \\
K_0 &= I \\
K_{n+1} &= \mathsf{cascadeStep}(K_n)
\end{aligned}$$

Theorem 4 (Cascade Monotonicity). $K_{n+1} \subseteq K_n$ *for all* $n \geq 0$.

Theorem 5 (Base Preservation). *If* $B \subseteq I$*, then* $B \subseteq K_n$ *for all* $n \geq 0$.

Definition 8 (Cascade Fixpoint). $K^* = \bigcap_{n \geq 0} K_n$.

Definition 9 (Cascade Stability). *Cascade is* stable at step n *if* $K_{n+1} = K_n$.

Theorem 6 (Stability Persistence). *If cascade is stable at step* n*, then* $K_m = K_n$ *for all* $m \geq n$.

Theorem 7 (Stable Fixpoint Characterization). *If cascade is stable at step* n*, then* $K^* = K_n$.

Theorem 8 (Cascade Soundness). *If cascade is stable at step* n*, then* $K^* \subseteq F(K^*)$*. That is, the cascade fixpoint is a prefixpoint of* F.

3.4.4 Overall Correctness

Theorem 9 (Expansion Correctness). *Let* $F \sqsubseteq F'$ *(expansion),* $S = \mathsf{lfp}(F)$*, and* $S' = \mathsf{lfp}(F')$*. If semi-naive iteration from* S *with operator* F' *stabilizes at step* n*, then:*

$$C_n = S'$$

Theorem 10 (Contraction Correctness). *Let* $F' \sqsubseteq F$ *(contraction),* $S = \mathsf{lfp}(F)$*, and* $S' = \mathsf{lfp}(F')$*. If cascade from* S *with operator* F' *stabilizes at step* n*, then:*

$$K^* = S'$$

These theorems state that the incremental update algorithm is *correct*: starting from the old fixpoint and applying the appropriate algorithm (semi-naive for expansion, cascade for contraction), we obtain exactly the new fixpoint.

Remark 1 (Lean Formalization). *All definitions and theorems above are formalized in Lean.*[1]

Fully proven *(no* `sorry`*):*

- *Semi-naive soundness:* $C_n \subseteq \mathsf{lfp}(F)$
- *Cascade soundness:* $K^* \subseteq F(K^*)$ *when stable*

[1] See `lean-formalisation/IncrementalFixpoint.lean`.

- *Fixpoint monotonicity under expansion/contraction*
- *Cascade stability persistence and fixpoint characterization*
- *Contraction direction: $S' \subseteq K^*$ (new fixpoint survives cascade)*

Overall correctness *requires additional assumptions:*

- ***Expansion*** *($C_n = S'$): Proven assuming (1) step is* additive*, i.e.,* $\mathsf{step}(A \cup B) = \mathsf{step}(A) \cup \mathsf{step}(B)$*, and (2) new base $\subseteq$ old fixpoint.*
- ***Contraction*** *($K^* = S'$): The direction $S' \subseteq K^*$ is proven assuming count is* monotone *in its set argument. The reverse $K^* \subseteq S'$ remains future work.*

Both additivity and count monotonicity hold for DCE-style operators.

4 Level 2: DSL with Automatic Derivation (Future)

The low-level API requires the user to provide stepFromDelta and derivationCount. A higher-level approach would let users define F in a structured DSL, from which these operations are derived automatically.

4.1 Analogy: Automatic Differentiation

	Differentiation	**Incremental Fixpoint**
Low-level	User provides $f(x)$ and $\frac{df}{dx}$	User provides F, stepFromDelta, derivationCount
High-level (DSL)	User writes expression; system derives gradient	User writes F in DSL; system derives incremental ops
Requirement	f given as expression tree	F given as composition of primitives
Black-box	Finite differences (slow)	Full recomputation (slow)

Just as automatic differentiation requires f to be expressed as a composition of differentiable primitives, automatic incrementalization requires F to be expressed as a composition of "incrementalizable" primitives.

4.2 Potential DSL Primitives

A DSL for fixpoint operators might include:

- $\mathsf{const}(B)$: constant base set
- $\mathsf{union}(F_1, F_2)$: union of two operators
- $\mathsf{join}(R, S, \pi)$: join S with relation R, project via π
- $\mathsf{filter}(P, S)$: filter S by predicate P
- $\mathsf{lfp}(\lambda S.F(S))$: least fixpoint

Each primitive would come with:

- Its incremental step function (for semi-naive)

- Its derivation counting semantics (for deletion)

Example 3 (DCE in DSL).
```
live = lfp(S =>
  union(
    const(roots),
    join(edges, S, (u, v) => v)
  )
)
```
The system derives:

- $\mathsf{stepFromDelta}(\Delta) = \mathsf{join}(\mathsf{edges}, \Delta, (u, v) \mapsto v)$
- $\mathsf{derivationCount}(v) = |\{u \mid (u, v) \in \mathsf{edges} \land u \in \mathsf{live}\}|$

4.3 Connection to Datalog

Datalog engines already perform this derivation:

- Rules are the structured representation of F
- Semi-naive evaluation is derived from rule structure
- Counting-based deletion (DRed) handles retraction

A general incremental fixpoint DSL would extend this beyond Horn clauses to richer operators (aggregation, negation, etc.).

5 Examples Beyond DCE

The incremental fixpoint pattern applies to many problems:

Problem	**Base**	**Step**	**Derivation Count**
DCE/Reachability	roots	successors	in-degree from live
Type Inference	base types	constraint propagation	# constraints implying type
Points-to Analysis	direct assignments	transitive flow	# flow paths
Call Graph	entry points	callees of reachable	# callers
Datalog	base facts	rule application	# rule firings

6 Relationship to Reactive Systems

In a reactive system like Skip:

- **Layer 1** (reactive aggregation) handles changes to the *parameters* of F (e.g., the graph structure).
- **Layer 2** (incremental fixpoint) maintains the fixpoint as those parameters change.

The two layers compose: reactive propagation delivers deltas to the fixpoint maintainer, which incrementally updates its state and emits its own deltas (added/removed elements) for downstream consumers.

7 Future Work

1. **Design Level 2 DSL**: Define a language of composable fixpoint operators with automatic incrementalization.
2. **Integrate with Skip**: Implement the incremental fixpoint abstraction as a reusable component in the Skip reactive framework.
3. **Explore stratification**: Extend to stratified fixpoints (with negation) where layers must be processed in order.
4. **Benchmark**: Compare incremental vs. recompute performance on realistic workloads.

8 Conclusion

The incremental DCE algorithm is an instance of a general pattern: maintaining least fixpoints incrementally under changes to the underlying operator. We propose a two-level architecture:

1. A **low-level API** where users provide the incremental operations (stepFromDelta, derivationCount).
2. A **high-level DSL** (future work) where these operations are derived automatically from a structured definition of F, analogous to how automatic differentiation derives gradients from expression structure.

This abstraction unifies incremental algorithms across domains (program analysis, databases, reactive systems) and provides a foundation for building efficient, correct incremental computations.