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 *well-founded cascade*:

- Use the iterative construction rank to identify well-founded derivers
- Remove elements with no well-founded derivers (and not in base)
- Propagate: removing an element may eliminate derivers for others

The key insight: cycle members have equal rank, so they don't provide well-founded support to each other. This correctly handles unreachable cycles.

3.2 Specification (Not Implementation)

Important: What follows is a *mathematical specification* of what the update algorithms compute, not an executable implementation. The Lean formalization proves correctness of these specifications but does not provide runnable code.

The user provides:

- base: seed elements
- $\mathsf{stepFromDelta}$: derive new elements from a delta
- Proof that step is *element-wise*: $x \in \mathsf{step}(S) \Rightarrow \exists y \in S.\, x \in \mathsf{step}(\{y\})$
- Proof that step is *additive*: $\mathsf{step}(A \cup B) = \mathsf{step}(A) \cup \mathsf{step}(B)$

Example 2 (DCE)**.** *DCE satisfies both properties: if* v *is reachable from* S*, there's a specific predecessor* $u \in S$ *with* $(u, v) \in \mathsf{edges}$.

3.3 Update Algorithms (Specification)

Expansion. When F grows, semi-naive iteration computes:

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

The sequence C_n is monotonically increasing. If it stabilizes (i.e., $\Delta_{n+1} = \emptyset$ for some n), then $C_n = \mathsf{lfp}(F')$.

Contraction. When F shrinks, well-founded cascade computes:

$$K_0 = \mathsf{lfp}(F), \qquad K_{n+1} = K_n \setminus \{x \in K_n \mid x \notin \mathsf{base}' \wedge \text{no wf-deriver in } K_n\}$$

where an element's *rank* is when it first appears in $F^n(\emptyset)$, and y is a *well-founded deriver* of x if $\mathsf{rank}(y) < \mathsf{rank}(x)$ and $x \in \mathsf{step}'(\{y\})$. Cycles don't provide support because cycle members have equal rank.

The sequence K_n is monotonically decreasing. If it stabilizes, then $K_n = \mathsf{lfp}(F')$.

3.4 From Specification to Implementation

The specifications above operate on abstract sets and assume convergence. A real implementation requires additional ingredients:

1. Finite representation. The specifications use $\mathsf{Set}\ A$ (arbitrary sets). An implementation needs:

- Finite domain or lazy enumeration
- Efficient set operations (membership, union, difference)

2. Termination. The specifications define $K^* = \bigcap_n K_n$ (infinite intersection). An implementation needs:

- Proof that iteration stabilizes in finite steps
- Or: a bound on the number of iterations (e.g., $|A|$ for finite domains)

3. Rank computation. Well-founded cascade requires comparing ranks. An implementation needs:

- Either: precompute and store ranks for all elements
- Or: compute ranks on-demand during cascade
- For DCE: rank = BFS distance from roots (computable)

4. Detecting stabilization.

- Expansion: check if $\Delta_{n+1} = \emptyset$
- Contraction: check if $K_{n+1} = K_n$ (no elements removed)

5. Complexity analysis.

- Expansion: $O(\text{new elements})$ iterations, each processing a delta
- Contraction: $O(|K_0|)$ iterations in the worst case

Remark 1 (What the Lean Formalization Provides)**.** *The Lean formalization proves:*

- ***Correctness:*** *If the algorithms stabilize, they compute the new fixpoint*
- ***Soundness:*** *Intermediate results are always subsets/supersets of the target*

It does not *provide:*

- *Termination proofs*
- *Complexity bounds*
- *Executable code*

3.5 Formal Definitions and Correctness

3.5.1 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* $\mathsf{lfp}(F') \subseteq \mathsf{lfp}(F)$*.*

3.5.2 Semi-Naive Iteration (Expansion)

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

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

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* $n \geq 0$*.*

3.5.3 Well-Founded Cascade (Contraction)

Definition 6 (Iterative Construction and Rank)**.** *The least fixpoint is constructed iteratively:*

$$F^0(\emptyset) = \emptyset, \qquad F^{n+1}(\emptyset) = F(F^n(\emptyset)), \qquad \mathsf{lfp}(F) = \bigcup_{n \geq 0} F^n(\emptyset)$$

The rank *of $x \in \mathsf{lfp}(F)$ is the minimum n such that $x \in F^n(\emptyset)$. Elements not in $\mathsf{lfp}(F)$ have no finite rank.*

Definition 7 (Well-Founded Derivation)**.** *Element y* well-foundedly derives *x if $\mathsf{rank}(y) < \mathsf{rank}(x)$ and $x \in \mathsf{step}(\{y\})$.*

Definition 8 (Well-Founded Cascade)**.** *Given initial set I:*

$$\mathsf{wfShouldDie}(S) = \{x \in S \mid x \notin B \wedge \textit{no wf-deriver in } S\}$$
$$K_0 = I, \qquad K_{n+1} = K_n \setminus \mathsf{wfShouldDie}(K_n)$$

Key insight: cycle members have equal (or no) rank, so they don't provide well-founded support to each other.

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 $n \geq 0$.*

3.5.4 Overall Correctness

Theorem 6 (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, then $C_n = S'$.*

Theorem 7 (Contraction Correctness)**.** *Let $F' \sqsubseteq F$ (contraction), $S = \mathsf{lfp}(F)$, $S' = \mathsf{lfp}(F')$. Assume* step *is* element-wise*: $x \in \mathsf{step}(T) \Rightarrow \exists y \in T.\, x \in \mathsf{step}(\{y\})$. Then well-founded cascade from S converges to $K^* = S'$.*

Remark 2 (Lean Formalization)**.** *All definitions and theorems are formalized in Lean.*[1]

Fully proven: *Well-founded contraction correctness, semi-naive soundness, fixpoint monotonicity, all helper lemmas.*

Remaining assumption *(1 `sorry`): Expansion completeness requires proving* $\mathsf{step}(\mathsf{current}) \subseteq \mathsf{current}$ *when stable.*

The API requires: ***`stepElementWise`*** *and* ***`stepAdditive`****. Both hold for DCE.*

4 Level 2: DSL with Automatic Derivation (Future)

The low-level API requires the user to provide stepFromDelta and prove that step is element-wise and additive. A higher-level approach would let users define F in a structured DSL, from which these properties are derived automatically.

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

4.1 Analogy: Automatic Differentiation

	Differentiation	**Incremental Fixpoint**
Low-level	User provides $f(x)$ and $\frac{df}{dx}$	User provides F, $\mathsf{stepFromDelta}$, proofs
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:

- $\mathsf{stepFromDelta}(\Delta) = \mathsf{join}(\mathsf{edges}, \Delta, (u, v) \mapsto v)$
- *Proof that step is element-wise (each edge provides a single derivation)*
- *Proof that step is additive (union distributes over step)*

4.3 Connection to Datalog

Datalog engines already perform similar derivations:

- Rules are the structured representation of F
- Semi-naive evaluation is derived from rule structure
- Well-founded cascade generalizes deletion handling

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 `stepFromDelta` and prove step is element-wise and additive.
2. A **high-level DSL** (future work) where these proofs are derived automatically from a structured definition of F, analogous to how automatic differentiation derives gradients from expression structure.

The key contribution is *well-founded cascade*: using the iterative construction rank to handle cycles correctly. Elements not in the new fixpoint have no finite rank, so they have no well-founded derivers and are removed.

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