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

3.1 API Specification

3.1.1 Types

A	Element type (e.g., graph nodes)
$\mathsf{Set}(A)$	Finite sets of elements
$\mathsf{Map}(A, \mathbb{N})$	Map from elements to natural numbers (ranks)

3.1.2 Configuration (User Provides)

$\mathsf{base} : \mathsf{Set}(A)$	Seed elements (e.g., roots in DCE)
$\mathsf{stepFwd} : A \to \mathsf{Set}(A)$	Forward derivation: elements derived from one element
$\mathsf{stepInv} : A \to \mathsf{Set}(A)$	Inverse: elements that derive a given element

Define $\mathsf{step}(S) = \bigcup_{x \in S} \mathsf{stepFwd}(x)$ and $F(S) = \mathsf{base} \cup \mathsf{step}(S)$.

3.1.3 State (System Maintains)

$\mathsf{current} : \mathsf{Set}(A)$	Current live set = $\mathsf{lfp}(F)$
$\mathsf{rank} : \mathsf{Map}(A, \mathbb{N})$	Rank of each element (BFS distance from base)

3.1.4 Required Properties

The user must ensure:

1. **stepInv correctness:** $y \in \mathsf{stepInv}(x) \Leftrightarrow x \in \mathsf{stepFwd}(y)$
2. **Element-wise** (automatic): step decomposes per-element via $\mathsf{stepFwd}$
3. **Additive** (automatic): $\mathsf{step}(A \cup B) = \mathsf{step}(A) \cup \mathsf{step}(B)$

Example 1 (DCE Instance)**.**

$$\begin{aligned} \mathsf{base} &= \mathsf{roots} \\ \mathsf{stepFwd}(u) &= \{v \mid (u, v) \in \mathsf{edges}\} \quad \textit{(successors)} \\ \mathsf{stepInv}(v) &= \{u \mid (u, v) \in \mathsf{edges}\} \quad \textit{(predecessors)} \end{aligned}$$

3.2 Algorithms

3.2.1 Expansion (BFS)

When the operator grows ($F \sqsubseteq F'$: base or edges added), propagate new elements:

```
expand(state, config'):
  frontier = config'.base \ state.current
  r = 0
  while frontier != {}:
    for x in frontier:
      state.current.add(x)
      state.rank[x] = r
    nextFrontier = {}
    for x in frontier:
      for y in config'.stepFwd(x):
        if y not in state.current:
          nextFrontier.add(y)
    frontier = nextFrontier
    r += 1
```

3.2.2 Contraction (Worklist Cascade)

When the operator shrinks ($F' \sqsubseteq F$: base or edges removed), remove unsupported elements:

```
contract(state, config'):
  // Initialize: nodes that lost base membership or an incoming edge
  worklist = { x | x lost support }
  dying = {}

  while worklist != {}:
    x = worklist.pop()
    if x in dying or x in config'.base: continue

    // Check for well-founded deriver (strictly lower rank)
    hasSupport = false
    for y in config'.stepInv(x):
      if y in (state.current \ dying) and state.rank[y] < state.rank[x]:
        hasSupport = true
        break

    if not hasSupport:
      dying.add(x)
      // Notify dependents
      for z where x in config'.stepInv(z):
        worklist.add(z)

  state.current -= dying
```

3.2.3 Why Ranks Break Cycles

The rank check $\mathsf{rank}[y] < \mathsf{rank}[x]$ is essential:

- Cycle members have *equal* ranks (same BFS distance from base)
- Therefore, they cannot provide well-founded support to each other
- An unreachable cycle has no well-founded support and is correctly removed

3.3 Correctness and Analysis

3.3.1 Proven Properties (Lean, all complete)

Theorem 1 (Expansion Correctness). *If $F \sqsubseteq F'$ and expansion terminates, then* $\mathsf{current} = \mathsf{lfp}(F')$*.*

Theorem 2 (Contraction Correctness). *If $F' \sqsubseteq F$ and contraction terminates, then* $\mathsf{current} = \mathsf{lfp}(F')$*.*

Theorem 3 (Soundness). *At all intermediate states: expansion gives* $\mathsf{current} \subseteq \mathsf{lfp}(F')$*; contraction gives* $\mathsf{lfp}(F') \subseteq \mathsf{current}$*.*

All proofs complete in Lean with no `sorry`.[1]

3.3.2 Complexity Analysis

Operation	Time	Space
Expansion	$O(\lvert\text{new}\rvert + \lvert\text{edges from new}\rvert)$	$O(\lvert\text{new}\rvert)$
Contraction	$O(\lvert\text{dying}\rvert + \lvert\text{edges to dying}\rvert)$	$O(\lvert\text{dying}\rvert)$
Rank storage	—	$O(\lvert\mathsf{current}\rvert)$ integers

For DCE, this matches the complexity of dedicated graph reachability algorithms.

3.3.3 Gap Between Proofs and Algorithms

Gap	Description	Status
Refinement	Pseudo-code implements the spec	Visually obvious
Termination	Algorithms halt on finite sets	Clear (monotonic)
stepInv	User provides correct inverse	Assumed

These gaps are mechanical to close. A good engineer can implement with confidence.

3.4 Formal Definitions (Reference)

For completeness, the formal definitions used in Lean proofs.

Definition 3 (Decomposed Operator). $F(S) = B \cup \mathsf{step}(S)$ *where* step *is monotone.*

Definition 4 (Rank). $\mathsf{rank}(x) = \min\{n \mid x \in F^n(\emptyset)\}$.

Definition 5 (Well-Founded Derivation). *y wf-derives x if* $\mathsf{rank}(y) < \mathsf{rank}(x)$ *and* $x \in \mathsf{step}(\{y\})$.

Definition 6 (Semi-Naive Iteration). $C_0 = I$, $\Delta_0 = I$, $\Delta_{n+1} = \mathsf{step}(\Delta_n) \setminus C_n$, $C_{n+1} = C_n \cup \Delta_{n+1}$.

Definition 7 (Well-Founded Cascade). $K_0 = I$, $K_{n+1} = K_n \setminus \{x \mid x \notin B \wedge \text{no wf-deriver in } K_n\}$.

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

4 Level 2: DSL with Automatic Derivation (Future)

The low-level API requires the user to provide $\mathsf{stepFromDelta}$ 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.

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 2 (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.