

# Incremental Fixpoint Computation: A Two-Level Architecture

## Contents

<b>1</b>	<b>Motivation: DCE as Incremental Fixpoint</b>	<b>2</b>
<b>2</b>	<b>The General Pattern</b>	<b>2</b>
<b>3</b>	<b>Level 1: Low-Level Incremental Fixpoint API</b>	<b>3</b>
3.1	API Specification . . . . .	3
3.1.1	Types . . . . .	3
3.1.2	Configuration (User Provides) . . . . .	3
3.1.3	State (System Maintains) . . . . .	3
3.1.4	Required Properties . . . . .	3
3.2	Algorithms . . . . .	4
3.2.1	Expansion (BFS) . . . . .	4
3.2.2	Contraction (Worklist Cascade with Re-derivation) . . . . .	5
3.2.3	Why Ranks Break Cycles . . . . .	6
3.3	Correctness and Analysis . . . . .	6
3.3.1	Proven Properties (Lean, all complete) . . . . .	6
3.3.2	Complexity Analysis . . . . .	6
3.3.3	Gap Between Proofs and Algorithms . . . . .	7
3.4	Formal Definitions (Reference) . . . . .	7
<b>4</b>	<b>Worked Example: DCE in Detail</b>	<b>7</b>
4.1	Setup . . . . .	8
4.2	Example: Expansion (Adding an Edge) . . . . .	8
4.3	Example: Contraction (Removing an Edge) . . . . .	9
4.4	Example: Contraction with Cycles . . . . .	9
4.5	Example: Re-derivation (Stale Ranks) . . . . .	10
4.6	Summary . . . . .	11
<b>5</b>	<b>Level 2: DSL with Automatic Derivation (Future)</b>	<b>11</b>
5.1	Analogy: Automatic Differentiation . . . . .	11
5.2	Potential DSL Primitives . . . . .	12
5.3	Connection to Datalog . . . . .	12
<b>6</b>	<b>Examples Beyond DCE</b>	<b>13</b>
<b>7</b>	<b>Relationship to Reactive Systems</b>	<b>13</b>

### 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. All algorithms are formally proven correct in Lean.

## 1 Motivation: DCE as Incremental Fixpoint

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

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

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

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

- **Expansion** ( $G \rightarrow G \oplus f$ ): The operator grows, so  $\text{lfp}(F_G) \subseteq \text{lfp}(F_{G'})$ . The old fixpoint is an underapproximation; we iterate upward.
- **Contraction** ( $G \rightarrow G \ominus f$ ): The operator shrinks, so  $\text{lfp}(F_{G'}) \subseteq \text{lfp}(F_G)$ . 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  $F : L \rightarrow L$ , the least fixpoint is  $\text{lfp}(F) = \bigcap \{x \mid F(x) \sqsubseteq x\}$ .*

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:

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

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

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

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

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

## 3 Level 1: Low-Level Incremental Fixpoint API

### 3.1 API Specification

**Remark 1** (Two API Levels). *The API has two levels depending on which operations are needed:*

- **Simple API** (expansion only): *Supports adding elements to base or edges. Requires only base and stepFwd.*
- **Full API** (expansion + contraction): *Also supports removing elements. Additionally requires stepInv and rank.*

*Many applications only need expansion (e.g., monotonically growing graphs). The simple API suffices and is easier to implement.*

#### 3.1.1 Types

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

#### 3.1.2 Configuration (User Provides)

$\text{base} : \text{Set}(A)$	Seed elements (e.g., roots)	required
$\text{stepFwd} : A \rightarrow \text{Set}(A)$	Forward derivation	required
$\text{stepInv} : A \rightarrow \text{Set}(A)$	Inverse derivation	for contraction

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

**Note:** If  $\text{stepInv}$  is not provided, the system can build it from  $\text{stepFwd}$ :

$$\text{stepInv}[y] = \{x \in \text{current} \mid y \in \text{stepFwd}(x)\}$$

This is computed once during initialization and maintained incrementally.

#### 3.1.3 State (System Maintains)

$\text{current} : \text{Set}(A)$	Current live set = $\text{lfp}(F)$	always
$\text{rank} : \text{Map}(A, \mathbb{N})$	BFS distance from base	for contraction

#### 3.1.4 Required Properties

**User Obligations (Low-Level API)** The low-level API requires the user to provide  $\text{stepFwd}$  and manage deltas explicitly. The correctness of the algorithms depends on the following guarantees:

1. **stepFwd stability:** During any single API call (`make` or `applyDelta`),  $\text{stepFwd}(x)$  must return consistent results for any  $x$ . This ensures the operator  $F$  is well-defined.

*Violation example:* If  $\text{stepFwd}$  reads from mutable external state that changes during an operation, the algorithm may produce incorrect results.

2. **Delta accuracy** (low-level API only): When using `applyDelta`, the delta must accurately describe changes to the step relation. Specifically:

- `addedToStep` must list pairs  $(x, y)$  where  $y$  is now in `stepFwd(x)` but wasn't before
- `removedFromStep` must list pairs  $(x, y)$  where  $y$  was in `stepFwd(x)` but no longer is
- `stepFwd` must already reflect the new state when `applyDelta` is called

**Managed API (No User Obligations)** A managed API can eliminate *both* user obligations by:

- Owning the step relation as explicit data (not a user-provided function)
- Computing `stepFwd` internally from its own state
- Automatically computing deltas when the user calls mutation methods

With a managed API, the user simply calls methods like `addToStep(x, y)` and `removeFromStep(x, y)`, and correctness is guaranteed by construction.

**Automatic Properties** Given the user obligations above, the following properties hold by construction:

1. **Monotonicity:**  $F$  is monotone because  $\text{step}(S) = \bigcup_{x \in S} \text{stepFwd}(x)$  is a union over  $S$ , which is monotone in  $S$ .
2. **stepInv correctness:** If `stepInv` is computed by the system from `stepFwd` (rather than user-provided), correctness is guaranteed:  $y \in \text{stepInv}(x) \Leftrightarrow x \in \text{stepFwd}(y)$ .
3. **Element-wise decomposition:** By definition, `step` decomposes via `stepFwd`.
4. **Additivity:** Follows from element-wise decomposition:  $\text{step}(A \cup B) = \text{step}(A) \cup \text{step}(B)$ .

**Example 1** (DCE Instance).

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

## 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 with Re-derivation)

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

**Subtlety: Stale Ranks.** The algorithm uses ranks computed from the *old* operator  $F$ , but after changes these ranks may be stale. For example, if element  $b$  was directly reachable from base (rank 1) but that edge is removed,  $b$  might still be reachable via a longer path (e.g.,  $\text{base} \rightarrow c \rightarrow b$ , giving rank 2). With stale ranks,  $c$  (rank 1) cannot provide well-founded support to  $b$  (also rank 1) because  $1 \not< 1$ .

The solution is to *re-derive* after contraction: check if any removed element can be re-derived from surviving elements via existing edges.

```

contract(state, config'):
    // Phase 1: Remove elements without well-founded support
    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 derivier (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

    // Phase 2: Re-derive elements that may still be reachable
    // (handles stale ranks from removed shortest paths)
    rederiveFrontier = {}
    for y in dying:
        for x in config'.stepInv(y):
            if x in state.current:
                rederiveFrontier.add(y)

```

```

break

// Run expansion from rederiveFrontier to recover elements
if rederiveFrontier != {}:
    expand(state, rederiveFrontier)

```

**Complexity of Re-derivation.** The re-derive phase iterates over removed elements and their predecessors:  $O(|\text{dying}| + |\text{edges to dying}|)$ . This preserves the incremental complexity—proportional to the change, not the graph size.

### 3.2.3 Why Ranks Break Cycles

The rank check  $\text{rank}[y] < \text{rank}[x]$  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)

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

**Theorem 2** (Contraction Correctness). *If  $F' \sqsubseteq F$  and contraction (cascade + re-derivation) terminates, then  $\text{current} = \text{lfp}(F')$ .*

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

All proofs are complete in Lean with no `sorry`.<sup>1</sup>

### 3.3.2 Complexity Analysis

Operation	Time	Space
Expansion	$O( \text{new}  +  \text{edges from new} )$	$O( \text{new} )$
Contraction (phase 1)	$O( \text{dying}  +  \text{edges to dying} )$	$O( \text{dying} )$
Re-derivation (phase 2)	$O( \text{dying}  +  \text{edges to dying} )$	$O( \text{rederived} )$
Rank storage	—	$O( \text{current} )$ integers

The re-derivation phase does not increase asymptotic complexity: it iterates over dying elements and their incoming edges, the same as phase 1. Total contraction remains  $O(|\text{affected}| + |\text{edges to affected}|)$ .

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

<sup>1</sup>See `lean-formalisation/IncrementalFixpoint.lean`. The proofs use a finiteness axiom: cascade stabilizes after finitely many steps. This is trivially true for finite sets (our practical case).

### 3.3.3 Proof Status

Aspect	Description	Status
Expansion	BFS computes new fixpoint	Proven
Contraction	Cascade + re-derive computes new fixpoint	Proven
Termination	Algorithms halt on finite sets	Axiom
stepInv	User provides correct inverse	Assumed

**Finiteness Axiom.** The proofs use one axiom: `cascadeN_stabilizes`—a decreasing chain of cascade steps stabilizes after finitely many iterations. This is trivially true for finite sets: each strict decrease removes at least one element, so after at most  $|S|$  steps, the sequence stabilizes. Our practical applications always use finite fixpoints.

**Why Cache Ranks?** Recomputing ranks after each change would cost  $O(V + E)$ , defeating incremental computation. By caching ranks and using re-derivation when needed, we achieve  $O(|\text{affected}|)$  complexity.

### 3.4 Formal Definitions (Reference)

For completeness, the formal definitions used in Lean proofs.

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

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

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

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

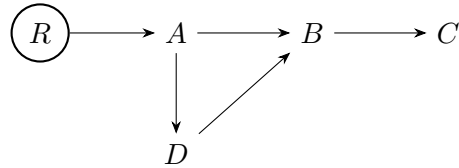
**Definition 7** (Well-Founded Cascade).  $K_0 = I$ ,  $K_{n+1} = K_n \setminus \{x \mid x \notin B \wedge \text{no wf-deriver in } K_n\}$ .

## 4 Worked Example: DCE in Detail

We illustrate the API and algorithms with Dead Code Elimination (DCE), showing how expansion and contraction work on concrete graphs.

### 4.1 Setup

Consider a program represented as a directed graph where nodes are code units and edges represent dependencies (“ $u \rightarrow v$ ” means  $u$  uses  $v$ ).



- $\text{base} = \{R\}$  ( $R$  is the root/entry point)
- $\text{stepFwd}(R) = \{A\}$ ,  $\text{stepFwd}(A) = \{B, D\}$ ,  $\text{stepFwd}(B) = \{C\}$ ,  $\text{stepFwd}(D) = \{B\}$

**Initial state after BFS expansion:**

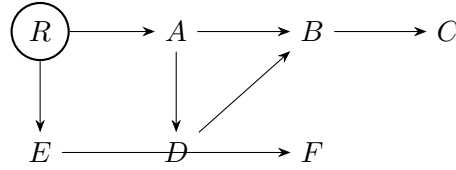
$$\begin{aligned}\text{current} &= \{R, A, B, C, D\} \\ \text{rank} &= \{R \mapsto 0, A \mapsto 1, B \mapsto 2, C \mapsto 3, D \mapsto 2\}\end{aligned}$$

Note:  $B$  and  $D$  have the same rank (both at distance 2 from  $R$ ).

For contraction we also write  $\text{stepInv}(x)$  for the set of predecessors  $y$  with an edge  $y \rightarrow x$ , and maintain a set  $\text{dying}$  of nodes scheduled for removal.

#### 4.2 Example: Expansion (Adding an Edge)

Suppose we add an edge  $R \rightarrow E$  where  $E$  is a new node with  $\text{stepFwd}(E) = \{F\}$ .



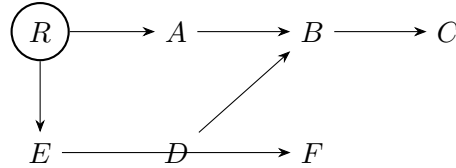
**Expansion algorithm:**

1.  $\text{frontier} = \{E\}$  (new successors of  $R$ ),  $r = 1$
2. Add  $E$  with  $\text{rank}[E] = 1$
3.  $\text{frontier} = \{F\}$ ,  $r = 2$
4. Add  $F$  with  $\text{rank}[F] = 2$
5.  $\text{frontier} = \{\}$ , done

**Result:**  $\text{current} = \{R, A, B, C, D, E, F\}$

#### 4.3 Example: Contraction (Removing an Edge)

Now suppose we remove the edge  $A \rightarrow D$ .



**Contraction algorithm:**

1.  $\text{worklist} = \{D\}$  (lost its incoming edge from  $A$ )
2. Process  $D$ :
  - $D \notin \text{base}$
  - Check for wf-deriver:  $\text{stepInv}(D) = \{A\}$ , but edge  $A \rightarrow D$  removed

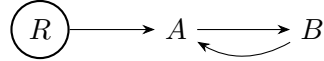
- No wf-deriver found, so  $\text{dying} = \{D\}$
- All dependents of  $D$  already have another wf-deriver, so no additional nodes are added to the worklist

3.  $\text{worklist} = \{\}$ , done

**Result:**  $\text{current} = \{R, A, B, C, E, F\}$  ( $D$  removed)

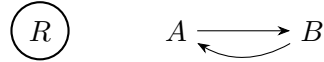
#### 4.4 Example: Contraction with Cycles

This example shows why *ranks* are essential. Consider:



- $\text{rank} = \{R \mapsto 0, A \mapsto 1, B \mapsto 2\}$
- $A$  has a wf-deriver:  $R$  with  $\text{rank}[R] = 0 < 1$
- $B$  has a wf-deriver:  $A$  with  $\text{rank}[A] = 1 < 2$

Now remove edge  $R \rightarrow A$ :



**Contraction:**

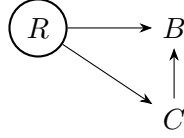
1.  $\text{worklist} = \{A\}$  (lost edge from  $R$ )
2. Process  $A$ :
  - $\text{stepInv}(A) = \{R, B\}$
  - $R \rightarrow A$  removed, so  $R$  doesn't count
  - $B$  is in current, but  $\text{rank}[B] = 2 > 1 = \text{rank}[A]$  — **not a wf-deriver!**
  - No wf-deriver, so  $A$  dies. Add  $B$  to worklist.
3. Process  $B$ :
  - $\text{stepInv}(B) = \{A\}$
  - $A$  is dying, so doesn't count
  - No wf-deriver, so  $B$  dies.
4.  $\text{worklist} = \{\}$ , done

**Result:**  $\text{current} = \{R\}$  (entire cycle removed)

**Key insight:** Without rank checking,  $A$  and  $B$  would keep each other alive (each derives the other). The rank check  $\text{rank}[y] < \text{rank}[x]$  breaks this mutual support because cycle members have equal or increasing ranks along cycle edges.

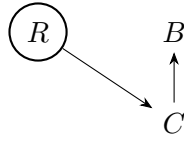
#### 4.5 Example: Re-derivation (Stale Ranks)

This example shows when re-derivation is necessary. Consider:



- $\text{rank} = \{R \mapsto 0, B \mapsto 1, C \mapsto 1\}$
- $B$  has two derivers:  $R$  (direct) and  $C$  (via  $C \rightarrow B$ )

Now remove edge  $R \rightarrow B$ :



**Phase 1 (Contraction with stale ranks):**

1.  $\text{worklist} = \{B\}$  (lost edge from  $R$ )
2. Process  $B$ :
  - $\text{stepInv}(B) = \{C\}$  (after removing  $R \rightarrow B$ )
  - $C$  is in current, but  $\text{rank}[C] = 1 \not\leq 1 = \text{rank}[B]$
  - **Stale rank problem:**  $B$ 's true rank in the new graph should be 2 (via  $R \rightarrow C \rightarrow B$ )
  - No wf-deriver found with stale ranks, so  $B$  added to dying
3.  $\text{dying} = \{B\}$

**Phase 2 (Re-derivation):**

1. For  $B \in \text{dying}$ : check  $\text{stepInv}(B) = \{C\}$
2.  $C$  is in current (surviving), so  $B$  is re-derivable
3.  $\text{rederiveFrontier} = \{B\}$
4. Expansion adds  $B$  back with  $\text{rank}[B] = 2$

**Result:**  $\text{current} = \{R, B, C\}$  (correct!)

Without re-derivation,  $B$  would be incorrectly removed even though it's still reachable via  $R \rightarrow C \rightarrow B$ .

## 4.6 Summary

Operation	Algorithm	Key Property
Add edge/root	BFS expansion	Assigns increasing ranks
Remove edge/root	Worklist cascade + re-derive	Rank check breaks cycles; re-derive handles stale ranks

Both algorithms are fully proven correct in Lean (using a finiteness axiom for cascade stabilization).

## 5 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.

### 5.1 Analogy: Automatic Differentiation

	Differentiation	Incremental Fixpoint
Low-level	User provides $f(x)$ and $\frac{df}{dx}$	User provides $F$ , <code>stepFromDelta</code> , 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.

### 5.2 Potential DSL Primitives

A DSL for fixpoint operators might include:

- `const( $B$ )`: constant base set
- `union( $F_1, F_2$ )`: union of two operators
- `join( $R, S, \pi$ )`: join  $S$  with relation  $R$ , project via  $\pi$
- `filter( $P, S$ )`: filter  $S$  by predicate  $P$
- `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:*

- `stepFromDelta( $\Delta$ ) = join(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)*

### 5.3 Connection to Datalog

The incremental fixpoint algorithms draw on ideas from deductive databases and Datalog:

- **Semi-naive evaluation:** Our expansion algorithm (BFS) is essentially semi-naive iteration [1], which computes only the “delta” at each iteration rather than recomputing the full set.
- **Differential Dataflow:** The delta-based approach to incremental updates is related to Differential Dataflow [2], which maintains recursive queries under changes to input relations.
- **Well-founded semantics:** The rank-based contraction is inspired by well-founded semantics [3], where derivations must be “well-founded” (not circular) to count. Our ranks provide a concrete measure: derivers must have strictly lower rank.
- **DRed (Delete and Rederive):** Our contraction algorithm is related to the DRed algorithm [4] for maintaining materialized views under deletions. DRed over-deletes then rederives; our well-founded cascade is more direct.

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

## 6 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

## 7 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.

## 8 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.

## 9 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.

## References

- [1] F. Bancilhon and R. Ramakrishnan. An amateur’s introduction to recursive query processing strategies. In *ACM SIGMOD*, 1986. (Semi-naive evaluation)
- [2] F. McSherry, D. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *CIDR*, 2013. (Incremental recursive computation)

- [3] A. Van Gelder, K. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):619–649, 1991. (Well-founded derivation)
- [4] A. Gupta, I. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *ACM SIGMOD*, 1993. (DRed algorithm for deletions)
- [5] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955. (Least fixpoint theory)