Incremental Whole-Program Analysis in Datalog with Lattices

Tamás Szabó
JGU Mainz / Workday
Germany

Sebastian Erdweg
JGU Mainz
Germany

Gábor Bergmann
Budapest University of Technology and Economics / IncQuery Labs
Hungary

Abstract

Incremental static analyses provide up-to-date analysis results in time proportional to the size of a code change, not the entire code base. This promises fast feedback to programmers in IDEs and when checking in commits. However, existing incremental analysis frameworks fail to deliver on this promise for whole-program lattice-based data-flow analyses. In particular, prior Datalog-based frameworks yield good incremental performance only for intra-procedural analyses.

In this paper, we first present a methodology to empirically test if a computation is amenable to incrementalization. Using this methodology, we find that incremental whole-program analysis may be possible. Second, we present a new incremental Datalog solver called LADDDER to eliminate the shortcomings of prior Datalog-based analysis frameworks. Our Datalog solver uses a non-standard aggregation semantics which allows us to loosen monotonicity requirements on analyses and to improve the performance of lattice aggregators considerably. Our evaluation on real-world Java code confirms that LADDDER provides up-to-date points-to, constant propagation, and interval information in milliseconds.

CCS Concepts: • **Software and its engineering → Automated static analysis**.

Keywords: Static Analysis, Incremental Computing, Datalog

ACM Reference Format:
Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. 2021. Incremental Whole-Program Analysis in Datalog with Lattices. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3453483.3454026

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06…$15.00
https://doi.org/10.1145/3453483.3454026

1 Introduction

Incremental static analyses provide up-to-date analysis results by processing code *changes*. They adapt previously computed analysis results to reflect the changed code. Compared to reanalyzing the entire code base from scratch, incremental analyses usually yield results orders of magnitudes faster, namely in time proportional to the size of the code change. This is why all modern IDEs include and continuously run incremental analyses: type checkers, code smell detectors, dead code analyses, and more.

The development of a correct incremental analysis is challenging, since it must yield the exact same results as a non-incremental analysis would. Specifically, the incremental analysis must precisely track dependencies between program elements such that it can invalidate and recompute all previous results affected by a code change. IDEs tend to use one-off algorithms to achieve this, which entails a significant development effort. Frameworks for incremental static analyses aim to relieve developers from this burden by automatically incrementalizing a non-incremental analysis specification. For example, Reviser [3] incrementally executes distributive analyses and Infer [8] incrementally executes compositional analyses. Both approaches rely on method summaries, exploiting distributivity and compositionality respectively. The goal of this paper is to provide automatic incrementalization for static analyses that are not necessarily distributive or compositional, and to make them fast enough to run inside IDEs.

Datalog-based analysis frameworks encode static analyses as Datalog rules. They rely on Datalog's relational fixpoint semantics rather than method summaries and therefore do not assume the analysis is distributive or compositional. For example, Doop [34] implements highly precise whole-program points-to analyses for Java in Datalog. However, Doop targets the non-incremental Datalog solver Soufflé [17]. In contrast, IncA [37] compiles analysis specifications to be solved by the incremental Datalog solver VIATRA QUERY [39]. This way, IncA has been used to realize incremental points-to analyses [37], interval and string analyses [36], and type analyses [27]. However, the literature on IncA only demonstrates millisecond update times for *intra-procedural* analyses. When we benchmarked IncA on one of Doop's whole-program points-to analyses, we discovered that IncA fails to

scale. Specifically, even small changes of the analyzed program regularly led to update times comparable to a complete reanalysis of the program.

In this paper, we tackle this problem in two ways. First, we develop an empirical methodology to help us estimate if a computation is inherently incompatible with incrementalization. For whole-program analyses, if a small program change invalidates most previous analysis results, efficient incrementalization must fail. However, we find that such high-impact changes are rare and efficient incrementalization thus may be possible, although IncA does not achieve it. Second, we investigate the performance issues of IncA and identify two problems in IncA's Datalog solver: unnecessary recomputations after deletions and overly strict monotonicity requirements for analysis specifications. Both impact performance significantly. To solve these problems, we design and implement a new incremental Datalog solver called LADDDER that can be used as a drop-in replacement in IncA. LADDDER is based on differential dataflow (DDF) [16, 22, 31], a generic computational model for incrementalizing dataflow-based fixpoint computations.

The key contribution of this paper is a novel aggregation semantics for DDF that supports incremental recursive aggregation, which is required for joins and meets in lattice-based data-flow analyses. What makes efficient incrementalization of recursive aggregators challenging in DDF is that DDF maintains a partially ordered set of intermediate computation states. Incremental aggregators must thus collect aggregands from and yield results at multiple such states. We designed data structures that enable efficient incrementalization in LADDDER by exploiting the algebraic properties of lattice-based aggregations. Moreover, we provide termination and correctness guarantees that are valid under relaxed monotonicity assumptions The implementation of LADDDER is available open source.[1]

We evaluate IncA with LADDDER for whole-program points-to, constant propagation, and interval analyses of Java code from the Qualitas Corpus [38]. We synthesize random program changes that affect the analysis results. While IncA's previous Datalog solver required tens of seconds to update analysis results, LADDDER responds to virtually all code changes within 10 ms. These results confirm that LADDDER provides a practical solution for incrementalizing whole-program lattice-based program analyses.

In summary, this paper makes the following contributions:

- We identify the challenges that come with the incrementalization of lattice-based whole-program analyses and discuss the limitations of the state of the art (Section 2).
- We present the methodology we used to verify that whole-program analyses are incrementalizable (Section 3).
- We introduce LADDDER, a novel DDF-based evaluator for Datalog with lattice-based recursive aggregation (Section 4).
- We design a novel aggregation architecture for LADDDER that exploits the algebraic properties of lattices to improve aggregation performance (Section 5).
- We formally define the semantics of LADDDER and provide correctness proofs (Section 6).
- We evaluate the performance of our approach with a lattice-based inter-procedural points-to analysis on real-world subject programs (Section 7).

[1] https://github.com/szabta89/IncA

2 Problem Statement

In this section, we review the state of the art of incremental Datalog-based analyses. While prior research has shown that incremental Datalog is effective for lattice-based intra-procedural analyses, we demonstrate that existing approaches do not scale to whole-program analyses. We discuss other non-Datalog-based approaches in Section 8.

In this section, we use a simple points-to analysis [33] as a running example to show that traditional incremental Datalog solvers do not scale. Points-to analysis is a fundamental analysis underpinning many other analyses, such as control-flow analysis or taint analysis. As demonstrated by the Doop framework [34], precise inter-procedural points-to analyses can be implemented in Datalog.

Datalog is a logic programming language [12]. Each Datalog a rule r has the form a_0 `:-` $a_1, \ldots, a_n$ with head a_0 and (possibly empty) body $a_1, \ldots, a_n$. Atoms a_i have the form $R(t_1, \ldots, t_k)$, where R is a relation name and t_i are terms. A rule is interpreted as a universally quantified implication: The substitutions of the variables in the body imply when the head holds. Given a head atom $R(t_1, \ldots, t_k)$, the valid substitutions of the terms $t_1, \ldots, t_k$ yield the tuples of relation R. Multiple Datalog rules can share the same head relation, thereby providing alternative ways to infer tuples for a single relation. A Datalog solver takes all rules of a Datalog program and computes their least fixpoint. A Datalog solver is incremental if it can update the fixpoint based on changes to the program's inputs.

As an example, consider the following subset of rules of a context-insensitive, flow-insensitive, yet inter-procedural points-to analysis for Java from the Doop framework [34]:

```
PT(var,obj) :– Reach(meth), Alloc(var,obj,meth).
PT(var,obj) :– Move(var,from), PT(from,obj).
PT(this,obj) :– Resolve(_,this,obj).
Resolve(meth,this,obj) :– PT(rcv,obj),
  VCall(rcv,sig,_,inMeth), Reach(inMeth),
  Type(obj,cls), Lookup(cls,sig,meth), ThisVar(meth, this).
Reach(meth) :– Resolve(meth,_,_).
Reach(meth) :– FuncName(meth, "main").
```

Doop uses a Datalog solver called Soufflé [17] to support the analysis of large code bases. However, Soufflé is not incremental and thus a complete reanalysis is necessary whenever any aspect of the analyzed program changes. To run Doop analyses incrementally, we tried to apply the incremental

Datalog solver provided by IncA. IncA is an incremental Datalog-based analysis framework that has been successfully used to incrementalize points-to [37], interval, and string analyses [36] in the past. However, the performance numbers in the literature only report on *intra-procedural* runs of IncA analyses; it is unclear if this approach scales to whole-program analyses.

We tested the scalability of IncA by running the inter-procedural points-to analysis from Doop on the source code of the MiniJava compiler.[2] While the code base only comprises 6.5 KLoC, inter-procedurality entails analyzing large parts of the JRE, so that the transitively reachable code size is considerable.

To our surprise, the performance of IncA on this benchmark was very poor. Specifically, deleting a single assignment from the analyzed code took up to 22 s until an updated analysis result was available, with a mean of 9 s for a sequence of random deletions. To put these numbers into perspective, the initial analysis took around 35 s. Even though, the initial analysis is a one-off cost, so its run time is acceptable, such high update times are incompatible with running whole-program analyses on-the-fly in IDEs. Indeed, we must conclude that IncA fails to efficiently incrementalize whole-program analyses, because the update time is not proportional to the change size.

Problem 1: Incrementalizability. We must pause and ask if we are trying to achieve an impossible task. Is it even possible to efficiently incrementalize whole-program analyses? A computational task is only incrementalizable if it adheres to the principle of inertia: Small input changes lead to small output changes. It is not clear if whole-program analyses adhere to this principle, because a small change in one part of the program may affect analysis results in many other places. When this happens, incremental processing of the code change will be inefficient. In Section 3, we present a simple yet effective empirical methodology to test if a computation is incrementalizable. We apply our method to test the incrementalizability of three inter-procedural analyses: points-to, constant propagation, and interval analyses. We find that all of the analyses are incrementalizable, and we suspect that this is true for many other analyses, as well. Therefore, we must tackle the second problem.

Problem 2: Scaling incremental analyses. While IncA efficiently executes intra-procedural analyses, it failed to yield satisfactory performance for whole-program analyses. We studied the implementation of IncA in detail and found that the performance problem is due to the underlying Datalog solver. IncA's Datalog solver is a variant of the famous DRed [13] algorithm. In response to a deletion, DRed invalidates *all* results that (transitively) depend on the deleted code, and then it re-derives results that are still valid after the deletion. This behavior in DRed is necessary to correctly update fixpoint results in face of *cyclic dependencies*. However it makes incremental handling of deletions prohibitively slow. To support incremental whole-program analyses, we must replace DRed with an incremental Datalog solver that scales better. However, in doing so, we must consider the specific requirements of static analyses.

[2] https://github.com/mtache/minijavac

```
PT(var,lat) :- Reach(meth),Alloc(var,obj,meth),lat=O(obj).
PT(var,lat) :- Move(var,from), PT_lub(from,lat).
PT(this,lat) :- Resolve(_,this,lat).
PT_lub(var,lub(lat)) :- PT(var, lat).
Resolve(meth,this,lat) :- PT_lub(rcv,lat),
  VCall(rcv,sig,_,inMeth), Reach(inMeth), lat = O(obj),
  Type(obj,cls), Lookup(cls,sig,meth), ThisVar(meth, this).
Resolve(meth,this,lat) :- PT_lub(rcv,lat),
  VCall(rcv,sig,_,inMeth), Reach(inMeth), lat = C(cls),
  LookupInSubclasses(cls,sig,meth), ThisVar(meth, this).
Reach(meth) :- Resolve(meth,_,_).
Reach(meth) :- FuncName(meth, "main").
```

Figure 1. A lattice-based points-to analysis in Datalog.

Problem 3: Lattice-based aggregation Static analyses routinely use lattices to abstract concrete values [26]. While this is unproblematic in monotone frameworks and abstract interpreters, recursive aggregation (e.g., least upper bound) is very challenging for Datalog solvers. Even non-incremental Datalog solvers often lack support for user-defined aggregation. For example, Soufflé only provides four built-in aggregators, so that Doop can only encode powerset-based analyses. Since we want to incrementalize a wide range of whole-program analyses, our new incremental Datalog solver must support recursive lattice-based aggregation.

For example, consider a lattice `Bot` $\sqsubseteq$ `O(obj)` $\sqsubseteq$ `C(cls)`, where `O` and `C` are singleton constructors for objects and class types, respectively. With this lattice, we precisely track singleton abstract objects `O(obj)`, but fall back to type-based method resolution on types `C(cls)` in other cases. Figure 1 shows a variant of the points-to analysis from above that uses this lattice. Importantly, we recursively aggregate the entries of relation `PT` in relation $\texttt{PT}_{lub}$, where we compute the least upper bound `lub` over a variable's points-to targets. The only previous Datalog solver to support this is a variant of DRed [36] in IncA, which our benchmark from above exposed as too slow for whole-program analyses. Moreover, this algorithm would diverge for our lattice-based analysis because it imposes per-rule *monotonicity requirements*, whereas our analysis only satisfies per-relation monotonicity for `Resolve`. We not only support incremental aggregation in our new Datalog solver, but also lift this overly strict requirement through a new aggregation semantics in Section 4.

3 Incrementalizability in Datalog

The core idea of incremental computing is that the processing of input changes is sometimes much faster than the reprocessing of the changed input. Yet, there is no good indicator

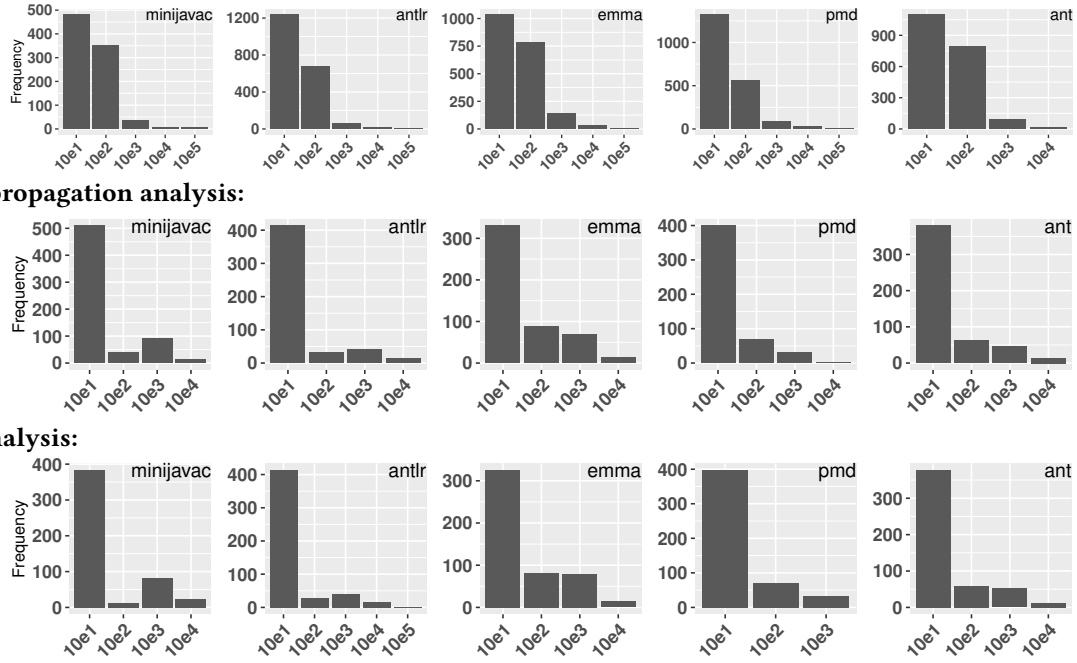


Figure 2. Whole-program analyses are incrementalizable because small input changes have low impact.

to determine which computations are amenable to such incrementalization. In particular, it is unclear if whole-program analyses can be efficiently incrementalized, or if all hope is lost because even small input changes require large amounts of recomputation. To answer this and similar questions, we propose a methodology that considers a necessary condition for a Datalog computation to be incrementalizable.

Datalog programs compute finite relations. This in contrast to other more general logic programming languages such as Prolog, which compute infinite relations. Most Datalog solvers exploit the finiteness of Datalog programs and exhaustively enumerate all derivable tuples. That is, they yield fully populated relations. We can use these relations to measure the impact of an input change:

Impact: The impact of an input change is the number of output tuples that are deleted or inserted because of it.

Importantly, we can measure the impact of an input change using a non-incremental Datalog solver: We run the computation once with the old input and once with the new input, and compute the difference of the output relation(s). For points-to analysis, the impact of a code change is the number of affected points-to tuples (relation PT). For the constant propagation and interval analyses, the impact of a code change is the number of affected value assignments to variables. Based on this notion of impact, we informally define incrementalizability.

Incrementalizability: A computation can only be incrementalizable if the vast majority of small input changes have low impact.

Equivalently, we can say that high-impact changes must be rare. Since we only restrict the effect on observable output tuples, our definition provides a necessary condition for incrementalizability. And we can test this condition empirically, as we shall see for whole-program analyses.

Whole-Program Analyses are Incrementalizable We apply our methodology to three whole-program analyses: points-to, constant propagation, and interval analyses. We benchmarked each analysis against the MiniJava compiler and against four real-world Java code bases from the Qualitas Corpus [38]: antlr, emma, pmd, and ant. We generated small input changes per code base that are likely to affect the analysis results. For points-to analysis, we delete or insert allocation sites; for the constant propagation and interval analyses, we replace numeric constants by zero. More details about the benchmark setup can be found in Section 7.

We measure the impact of each input change. We show the measurement results in Figure 2, where we group the data into exponentially growing buckets. For example, the third bucket 10e3 shows the number of input changes that affected between 10 and 100 tuples, the fourth bucket 10e4 shows the number of those that affected between 100 and 1000 tuples, and so on. As a frame of reference, note that the entire database contains millions of tuples for each analysis run. Our measurements clearly show that the vast majority of changes have low impact, although changes with higher impact do exist. This observation holds across both analyses and across all five code bases. Therefore, we conclude that whole-program analyses are incrementalizable.

4 LADDDER: An Incremental Datalog Solver

While Datalog extended with lattices is a good fit for a wide range of static analyses, Section 2 revealed two main obstacles to efficient incrementalization, especially in the face of high-impact changes: cyclic dependencies among partial analysis results and overly strict monotonicity requirements on the analysis definition. To address these challenges, we develop a new incremental Datalog solver called LADDDER. LADDDER uses a non-standard aggregation semantics to loosen the monotonicity requirements enforced by previous incremental solvers. This way, LADDDER supports a wider range of analysis definitions, including the lattice-based points-to analysis from Figure 1.

LADDDER builds on differential dataflow [22] (DDF), which is a generic computational model to incrementally maintain any iterative dataflow computation. LADDDER and other DDF-style approaches [16, 24, 31] track at which fixpoint iteration a tuple was derived. We call this the *timestamp* of the tuple. Since tuples can only recursively depend on tuples that were derived in earlier fixpoint iterations, we obtain fine-grained data dependencies that help incrementality. This is in contrast to other incremental Datalog solvers like DRed [13], which merge all tuples derived up until any iteration count into a single set.

4.1 Initial Analysis with LADDDER

The input to LADDDER is an analysis encoded as Datalog rules plus facts encoding the subject program. LADDDER repeatedly applies rules until a fixpoint is reached, thereby computing the tuples that the relations (defined by the rules) consist of. LADDDER follows a semi-naïve evaluation strategy [12]: In each iteration of the fixpoint computation, LADDDER only considers new tuples from the previous iteration instead of re-applying rules on the whole set of tuples computed thus far. LADDDER breaks up the analysis into *dependency components* (sets of mutually recursive rules, also called *strata* in Datalog) and applies rules according to a topological ordering of these components: Only after the fixpoint iterations finished in all *upstream* components, will LADDDER start evaluating a *downstream* component.

Consider again the singleton points-to analysis of Figure 1, where we used lattice `Bot` $\sqsubseteq$ `O(obj)` $\sqsubseteq$ `C(cls)` with singleton objects `O` and class types `C`. We illustrate the evaluation trace of LADDDER in Figure 4, where we analyze the example program from Figure 3. The trace reads from top to bottom along the fixpoint *iteration time axis*. An increasing *timestamp* (denoted by `T`) value is associated with every iteration, and the figure shows all tuples inferred at a specific timestamp. Fixpoint computation starts from the tuples produced by upstream dependency components, which, for our example, are all singleton components consisting of rules enumerating facts. These facts all appear at timestamp 0. Generally, we must increment timestamps (i.e. postpone for the next iteration) at least once as we go around a dependency cycle among rules to unroll the recursion. In the example, we chose a simple way to achieve this: Each inferred head tuple gets a timestamp that is one higher than the highest timestamp of the tuples used in the rule body.

Support counts LADDDER also maintains a support *count* for each tuple *per timestamp*: The count is equal to the number of alternative derivations a given tuple has at a specific timestamp. For example, `Reach(proc)` has count 2 at timestamp 7 (note the 2× symbol) because it can be inferred in two alternative ways by using the two `Resolve` tuples from timestamp 6. Support counts save time during incremental maintenance because they tell LADDDER if alternative derivations remain for a tuple after deletions. We demonstrate this in detail in Section 4.2.

Based on support counts, LADDDER considers different forms of *timelines* for each tuple. Figure 5 shows the timelines of tuple `Reach(proc)`; for now, ignore the dashed lines in the figure. Cumulative count shows the total number of alternative derivations as a function of timestamps: There are 2 alternative derivations at timestamp 7, and one more at timestamp 10. From the cumulative count timeline, a cumulative existence timeline can be inferred: Its value is 1 if the tuple exists at a specific timestamp, 0 otherwise. Ultimately, the cumulative existence is the important information for the fixpoint computation: a Datalog rule can use a tuple only when it "exists". This is straightforward during the initial semi-naïve evaluation because consecutive iterations infer new tuples based on the tuples from previous iterations, but will gain significance in an incremental setting (see Section 4.2). To maintain the cumulative timelines during semi-naïve evaluation, LADDDER uses differential timelines that reflect the changes in the cumulative ones. The differential count signals the changes in support counts at specific timestamps, while the differential existence is either +1 or -1 to signal the appearance or disappearance of a tuple.

Aggregation The aggregation semantics of LADDDER vastly differs from previous approaches, influencing how timelines are maintained. All existing DDF-based frameworks [16, 22, 31], as well as non-DDF approaches like IncA or Flix [21], use an aggregation semantics that was introduced by Ross and Sagiv [30]. In this semantics, a change in an aggregate result is represented with a deletion of the old and an insertion of the new result. In contrast, LADDDER uses a nonstandard *inflationary* semantics [14]. This means that LADDDER never retracts old aggregate results, it only ever inflates the set of aggregate results with the newly computed ones along the iteration time axis. Formally, for an aggregation group $\overline{g}$ (set of grouping variables), timestamp t, and aggregation operator α, LADDDER computes the set of tuples $\{(\overline{g}, \alpha(M[t])), (\overline{g}, \alpha(M[t-1])), \ldots (\overline{g}, \alpha(M[0]))\}$ where $M[T]$ yields the multiset of aggregands at timestamp T. This is also visible in Figure 4 because at timestamp 9 LADDDER infers $\texttt{PT}_{lub}\texttt{(f,O(F1))}$ (from

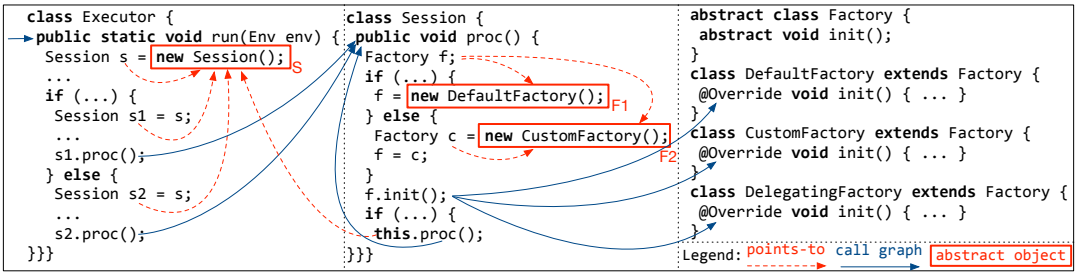


Figure 3. Example subject program used as input to the singleton points-to analysis.

T → Tuples produced at timestamp T
0 → *facts* (VCall, Move, Alloc, …)
1 → Reach(run)
2 → PT(s, O(S))
3 → PT_{lub}(s, O(S))
4 → PT(s1, O(S)), PT(s2, O(S))
5 → PT_{lub}(s1, O(S)), PT_{lub}(s2, O(S))
6 → Resolve(s1.proc(), proc, $\text{this}_{Session}$, O(S)), Resolve(s2.proc(), proc, $\text{this}_{Session}$, O(S))
7 → 2×PT($\text{this}_{Session}$, O(S)), 2×Reach(proc)
8 → PT_{lub}($\text{this}_{Session}$, O(S)), PT(f, O(F1)), PT(c, O(F2))
9 → Resolve(this.proc(), proc, $\text{this}_{Session}$, O(S)), PT_{lub}(f, O(F1)), PT_{lub}(c, O(F2))
10→ PT(f, O(F2)), PT($\text{this}_{Session}$, O(S)), Reach(proc), Resolve(f.init(), $\text{init}_{DefFactory}$, $\text{this}_{DefFactory}$, O(F1))
11→ PT_{lub}(f, C(*Factory*)), PT_{lub}($\text{this}_{Session}$, O(S)), Reach($\text{init}_{DefFactory}$)
12→ Resolve(f.init(), $\text{init}_{DefFactory}$, $\text{this}_{DefFactory}$, C(*Factory*)), Resolve(f.init(), $\text{init}_{CusFactory}$, $\text{this}_{CusFactory}$, C(*Factory*)), Resolve(f.init(), $\text{init}_{DelFactory}$, $\text{this}_{DelFactory}$, C(*Factory*))
13→ Reach($\text{init}_{CusFactory}$), Reach($\text{init}_{DelFactory}$)

Figure 4. LADDDER evaluation trace for the singleton points-to analysis on the subject program from Figure 3.

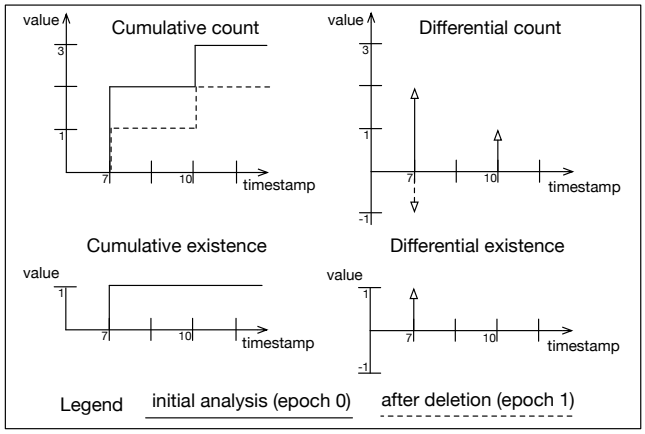


Figure 5. Timelines of `Reach(proc)` as maintained by LADDDER.

$M[9] = \{O(F1)\}$) and then PT_{lub}(`f,C(`*Factory*`)`) at timestamp 11 (from $M[11] = \{O(F1), O(F2)\}$), without retracting the former. As we detail in Section 4.3, this semantics allows us to loosen the monotonicity requirements on analysis definitions compared to the stricter requirements of existing solutions. With inflationary semantics, once a tuple gets inferred at a timestamp, it will exist at all subsequent timestamps along the iteration axis. In other words, the existential timelines can actually be represented by a single timestamp, which signals the first moment when a tuple appears during the fixpoint computation. For example, PT_{lub}(`f,O(F1)`) has a validity interval $T \in$ `[9,`∞`)` according to inflationary semantics, while it would have `[9,11)` with other DDF-based approaches.

However, with inflationary semantics, the relations of the analysis will contain additional, intermediate, aggregate results. Typically, this is unwanted by downstream components or analysis clients. Similarly, timestamps do not carry useful information for downstream components; they are important inside the current component to unroll recursion. To this end, LADDDER performs two steps of *postprocessing* on the output of each component. First, it provides a *timeless* view of the tuples by simply not writing timestamps to the output. Second, LADDDER filters out intermediate results and only writes the final aggregate result to the output for each aggregation group. The final result is either the largest or smallest value according to the aggregation direction.

4.2 Incremental Analysis with LADDDER

Orthogonal to the iteration time axis, LADDDER uses a separate time axis to represent epochs when the input changes. Assume that LADDDER already computed and stored the results of a semi-naïve evaluation based on the input at epoch e_1 and a change happens at e_2, where $e_2 > e_1$. The goal of incrementalization is to correct the states of the previous evaluation based on the diff between the inputs at e_1 and e_2, so that they become as if the input at e_2 was evaluated from scratch. LADDDER uses the input diff to trigger a new fixpoint evaluation that infers which tuples need to be inserted or deleted at higher timestamps. Let us look at an example.

Assume that we associate epoch 0 with the initial code in Figure 3, and we delete `s2.proc()` at epoch 1. The input diff is the deletion of a virtual call fact, and LADDDER uses this to update (compensate) its state to epoch 1:

T	Difference of tuples starting at timestamp T
0	-VCall(s2, proc, s2.proc(), run)
6	-Resolve(s2.proc(), proc, $\text{this}_{Session}$, O(S))
7	-PT($\text{this}_{Session}$, O(S)), -Reach(proc)

Concepts presented before, such as timestamps, support counts, and timelines, all come into play. The compensation starts with the input diff shown at timestamp 0 as a deletion, hence the minus (-) symbol. The deletion of this fact invalidates a `Resolve` tuple at timestamp 6, originally derived by the first `Resolve` rule in Figure 1. Among the tuples used in the rule body during the initial derivation, $\texttt{PT}_{lub}\texttt{(s2,O(S))}$ was inferred latest at 5, hence the timestamp 6 for the head. Propagating this, LADDDER deletes *one* derivation each of both $\texttt{PT(this}_{Session}\texttt{,O(S))}$ and `Reach(proc)` at timestamp 7. The support count of both tuples gets decremented to 1 (c.f. Figure 4 at timestamp 7), but this means that an alternative derivation still remains for each tuple. There is no existential diff, so the compensating propagation terminates.

The example is indicative of a scenario where IncA would not stop in only three steps, but would rather over-delete and re-derive much of the previous result. The underlying fixpoint algorithm DRed cannot tell apart the different derivations of the `Reach(proc)` tuples, having coalesced them into a single state. Therefore it must over-delete to tackle cyclic dependencies [36], as a positive support count remaining after deleting a derivation of a tuple is insufficient evidence for its continued existence. This is easily demonstrated by deleting the `s1.proc()` call in `run`; then the only justification for `proc` being reachable is the recursive call in itself, but that recursive call would not be executed at all if there is no other function calling `proc`.

While incrementalizing simple relational algebra operations in DDF is well-understood [22, 24], the efficient incrementalization of aggregation along two (epoch and iteration) time axes is challenging. In Section 5, we propose novel data structures that can efficiently maintain lattice aggregates in such a setting. But first, we review the assumptions of LADDDER on analysis definitions.

4.3 Monotonicity, Assumptions, and Guarantees

A novel aspect of our work is that LADDDER imposes looser monotonicity requirements on analysis definitions compared to prior approaches. We first discuss the restrictions in the state of the art by revisiting the Datalog code in Figure 1.

Recall that for the subject program from Figure 3 the points-to value associated with `f` is updated from `O(F1)` to `C(`*Factory*`)` in iteration 11 (c.f. Figure 4). The standard non-inflationary Ross and Sagiv semantics [30] would represent the change in the aggregate result as a deletion-insertion pair: $\texttt{-PT}_{lub}\texttt{(f,O(F1))}$ and $\texttt{+PT}_{lub}\texttt{(f,C(}\textit{Factory}\texttt{))}$. These tuples both fall in *aggregation group* `f`, and the inserted lattice value dominates the deleted one (`O(F1)` $\sqsubseteq$ `C(`*Factory*`)`). Ross and Sagiv call this a $\sqsubseteq$-*increasing change* of the entire relation.

Termination and minimality of the fixpoint computation is only guaranteed for $\sqsubseteq$-monotonic recursions: if recursive inputs only $\sqsubseteq$-increase, outputs can only $\sqsubseteq$-increase as well. If an output tuple is deleted at *some* stage of the fixpoint computation, then a $\sqsubseteq$-dominating insertion must also appear at the *same* stage. For instance in Figure 1, the first `Resolve` rule is problematic because it retracts previous inferences once a points-to set $\sqsubseteq$-increases to `C(`*Factory*`)`. As stated in Section 2, IncA failed to terminate, because the solver could not guarantee that the second `Resolve` rule would produce the $\sqsubseteq$-dominating insertion at the same stage of the fixpoint computation. The problem is not unique to DRed; Ross and Sagiv semantics can lead to a diverging fixpoint computation in DDF as well, when timestamps of the deletion and the $\sqsubseteq$-dominating insertion fail to line up.

Solving this non-termination challenge, if possible at all, would require either very clever preprocessing of the Datalog rules, or input from the analysis developer. The set of relations c where timestamps are to be incremented need to be selected carefully: While the easiest option would be to increment timestamps after each computation step or rule application, this could easily break the alignment between the two `Resolve` rules in the above example. Thus c needs to (i) avoid the above mentioned non-$\sqsubseteq$-monotonicity, and yet (ii) be a *cut* of the recursion, i.e. each dependency cycle goes through at least one timestamp increase, in order to avoid cyclic dependencies in the analysis result. Such a fixed choice of c will also restrict the ability of the solver to internally optimize the rules, e.g. by extracting common sub-rules into auxiliary relations that are separately maintained. However, the latter is a frequently used query optimization strategy (cf. higher-order view maintenance [2]). To this end, LADDDER uses the following assumption (looser that Ross and Sagiv), enabled by inflationary aggregation:

Eventual $\sqsubseteq$-monotonicity (ASM1) Our more relaxed assumption also requires $\sqsubseteq$-monotonicity, but only for one cut, and only in an *eventual* sense. If the relations in the cut only $\sqsubseteq$-increase in the recursive input, they must only $\sqsubseteq$-increase on the output. But the insertion of any $\sqsubseteq$-dominating tuple may be derived at an arbitrary timestamp, potentially later than the deletion it dominates.

Furthermore, the analysis developer does not need to inform the query engine of a cut, as in, the developer can just focus on defining the rules of the analysis. As long as an eventually monotonic cut exists, the solver can choose any other cut and still get correct results. This allows larger freedom for the analysis developer, but also for the solver to perform optimizations. In particular, the analysis developer only has to check that for each non-$\sqsubseteq$-monotonic rule, another rule exists that will eventually dominate the decrease, so that the end result is $\sqsubseteq$-monotonic. See in Figure 1; the first `Resolve` rule is non-$\sqsubseteq$-monotonic if `lat` $\sqsubseteq$-increases, but the second `Resolve` rule eventually $\sqsubseteq$-dominates it.

LADDDER imposes two further assumptions as well:

Well-behaving aggregators (ASM2) Each recursive aggregator must be well-behaving: (i) it is an associative and commutative binary operation, (ii) it respects a partial order $\sqsubseteq$, that is, when applied to a (multi)set of aggregands, the

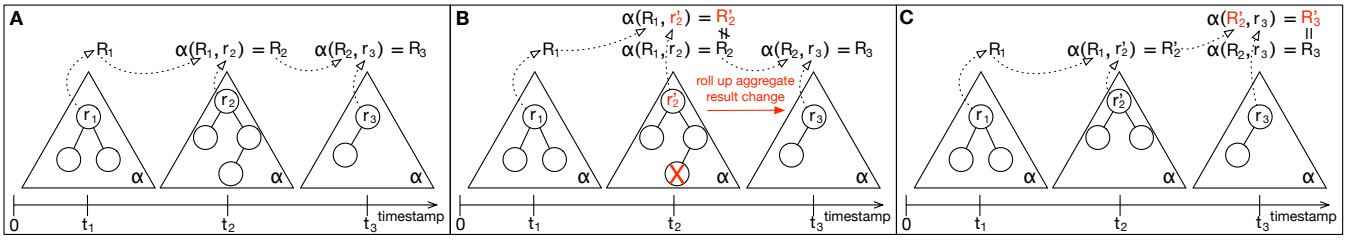


Figure 6. Sequential architecture. Triangles represent balanced trees maintaining (intermediate) aggregates.

result must $\sqsubseteq$-dominate the aggregands, (iii) it guarantees a stationary output in a finite number of repeated applications even in case of infinite lattices (i.e. is a widening [7]). For lattices with finite ascending chains, the lattice operations `lub`/`glb` immediately satisfy these criteria.

Stratified recursion (ASM3) Non-monotonic recursion is forbidden, which entails two constraints. First, stratified negation is required. Second, for each lattice that is *produced* in a dependency component, all aggregators applied on a specific lattice must be well-behaving, agree on the same $\sqsubseteq$ ordering direction, and form a cut. We emphasize that the requirement applies per lattice per component. Note the word *produced*, as it can happen that a lattice is produced in an upstream component, and the current component just treats is relationally without aggregating on it. Both of these requirements are standard in Datalog solvers, e.g. IncA also expects that analyses meet these conditions.

Guarantees If an analysis satisfies these requirements, then Laddder guarantees that the fixpoint computation terminates and yields the minimal model, i.e. the $\sqsubseteq$-smallest relations that are compatible with the rules. Laddder supports any lattice-based analysis that is eventually $\sqsubseteq$-monotonic, which also includes $\sqsubseteq$-monotonic analyses. We emphasise that, in case the analysis is $\sqsubseteq$-monotonic according to the Ross and Sagiv semantics, our eventually $\sqsubseteq$-monotonic semantics yields the *exact same* results as the traditional one. Note that in this paper we informally refer to our abstract domains as lattices (as is typical in program analysis), but technically we only require a partial order with a specific kind of aggregation operator. A formal treatment of the assumptions, semantics, correctness properties, and their proof sketches are available in Section 6.

4.4 Comparing Laddder to Standard Datalog

The perceptive reader may wonder at this point how is Laddder actually better than just using an incremental solver for standard stratified Datalog (without recursive aggregation or negation). Considering the fact that standard Datalog can already express certain lattices, this is a valid question.

First, any aggregator can be implemented by a standard Datalog rule `q(X) :- q(A),q(B),X is lub(A,B)`. Standard Datalog is already inflationary (unlike recursive aggregation [30], where we had to specifically apply inflationary aggregation), so it can be made functionally correct by the same *postprocessing* that we use in Laddder. But this rule may yield an exponential amount of intermediate values.

Second, while Laddder uses inflation, it never propagates *aggregands*, and it preserves a lot fewer intermediate *aggregate results* than an incremental solver for standard Datalog would do. Even for e.g. constant propagation, where there is no exponential blow-up, Laddder is a significant improvement, as an incremental solver for standard Datalog would propagate all potential constants for each variable, plus Top, while Laddder would propagate a single constant until a second constant is found, then it would propagate Top only.

Third, while standard Datalog can perform "set union" aggregations (powerset lattice) as one step, only certain *finite* lattices can be encoded so, and even then it is generally impractical (as argued by the authors of Flix [21, Section 1]).

5 Incremental Aggregation in Laddder

As said in Section 4, DDF is incremental along multiple time axes: input epochs and iteration rounds. Thus an aggregator data structure must yield the aggregate value (per each aggregation group) as a function of iteration timestamp, and then incrementally amend this function for each new epoch.

To aggregate lattices more efficiently than the standard aggregator architectures [22] originally proposed for DDF, we present a novel *sequential incremental architecture* that takes advantage of (i) the binary aggregation operator being associative and commutative as per **Well-behaving aggregators (ASM2)** and (ii) Laddder being inflationary so that all aggregands present at a given iteration round are also present in all subsequent iterations.

We maintain a (multi)set of aggregands plus their aggregate value for each iteration round, sparsely omitting timestamps where the multiset has not changed; these all are incrementally maintained upon each new epoch. The multiset of all aggregands valid at a timestamp can be split into a multiset of "new" aggregands inserted at that timestamp, and the multiset of "old" aggregands inserted at any previous timestamp (none of which are removed, due to inflation); the total aggregate would then be the result of the binary aggregation of these two collections. Since the aggregate of "old" values is known anyways (as the aggregate value at the previous timestamp), it is sufficient to maintain the "new" aggregands per each timestamp in an incremental aggregator data structure. We use the data structure introduced in

IncA [36] (applicable due to associativity and commutativity), which maintains (i) the multiset as a balanced binary tree (e.g. AVL tree [32, Chapter 3.3]), and (ii) at each node the aggregate of the subtree rooted there.

This architecture is depicted in Figure 6: At each timestamp, we maintain (as a tree) the aggregate of values inserted in that iteration, if there is any (**A**). α denotes the aggregation operator, and small r_i denotes the aggregate result at the root of a tree. Capital R_i depicts the total aggregate value, which is computed as a *sequential* roll up of all tree root aggregates r_i up to a specific timestamp. When a new epoch updates one of these trees (**B**), its local aggregate is maintained as usual, and the total aggregate at the timestamp is recomputed. Then the new total aggregate will roll up to each later timestamp to recompute the totals, stopping early if there is no more change at some point (**C**).

6 Formal Semantics of Laddder

We provide formal treatment of the theory behind Laddder. After introducing the necessary terminology, we provide a more detailed description of the assumptions of Laddder on input analyses. Then, we spell out correctness properties about Laddder, and we prove that Laddder satisfies them.

6.1 Concepts

For basic Datalog, we refer the reader to Green et al. [12]. We introduce advanced or non-standard terminology.

Predicates The predicates in a dependency component are divided into *exported* and *private* predicates, depending on whether they are directly used in downstream dependency components or user-facing results. The former group is denoted $Exp(D)$ for dependency component D. The Datalog solver is free to introduce new private predicate symbols for storing auxiliary results and rearrange rules in a way that leaves the meaning of exported predicates intact. A *cut* of a dependency component is a subset of its predicates with the property that all recursive dependency loops in the component must intersect the cut. An *interpretation* is an assignment of actual relations to Datalog predicate symbols.

Immediate consequence Fixing the interpretation of the predicates in a cut (as well as any upstream predicates) allows the non-recursive evaluation of all rules in the component. For a cut c of a component D, the *immediate consequence* operator $T_c(I, J_c)$ takes I as an interpretation of upstream dependency components and J_c an interpretation of predicates in the cut, and directly applies the Datalog rules in the component to derive a collection of tuples. These tuples will form a new interpretation of all predicates in D. By recursion, this includes the relations in c; we denote by $T_c(I, J_c)[c]$ the restriction of the results to the predicates in the cut.

In addition to the standard Datalog immediate consequence operator $T_c(I, J_c)$, Laddder also makes use of an alternative immediate consequence operator $\hat{T}_c(I, J_c)$ referred to as *inflationary consequence*, which is obtained by modifying the behavior of aggregators. In addition to the aggregate value of the current iteration, $\hat{T}$ also returns (derives as additional tuples in the aggregating relation) each aggregate result obtained at any earlier iteration timestamp.

Laddder iteratively applies the inflationary consequence operator. We denote as $\hat{T}_c^{(k)}$ the effect of k iterations, with $\hat{T}_c^{(0)}(I, J_c) = J_c$ and $\hat{T}_c^{(k+1)}(I, J_c) := \hat{T}_c(I, \hat{T}_c^{(k)}(I, J_c)[c])$., while $\hat{T}_c^{\omega}(I, J_c)$ consists of all tuples derived in any number of iterations: $\hat{T}_c^{\omega}(I, J_c) := \hat{T}_c^{(1)}(I, J_c) \bigcup \hat{T}_c^{(2)}(I, J_c) \bigcup \ldots$.

Lattices and ordering Datalog rules with aggregation or expression evaluation can compute new values beyond those present in the input (extensional) relations. These values belong to appropriate *abstract domain*s, which are often (practically) infinite. In our use case of program analysis, such domains are typically lattices. Note, however, that for Laddder to work correctly, we only actually require partial orders equipped with binary operators having certain properties (see Section 6.2).

Given that Datalog rules can simply use lattice values produced in upstream dependency components without aggregating them, we explicitly say that a component *produces a lattice* if the value gets derived in the component by expression evaluation or aggregation.

Taking any one of the two partial orderings $\sqsubseteq$ for each *produced lattice* of a dependency component, they can be naturally extended [30] to: (i) tuples derived by a Datalog rule as $t \sqsubseteq t'$, if t and t' are $\sqsubseteq$-related on all variables of computed lattices, and agree elsewhere; (ii) entire interpretations $I \sqsubseteq I'$ if all $t \in I$ have a $t' \in I'$ with $t \sqsubseteq t'$. The latter relationship is a *preorder* (transitive but not antisymmetric), as it permits $I \sqsubseteq I' \sqsubseteq I$, denoted as $I \simeq I'$, even if $I \neq I'$; for finite interpretations this means agreement in a subset of tuples that $\sqsubseteq$-dominate all differences.

6.2 Refined Assumptions on Datalog Rules

We refer to **Well-behaving aggregators (ASM2)** and **Stratified recursion (ASM3)** from Section 4.3. However, **Eventual $\sqsubseteq$-monotonicity (ASM1)** was introduced in Section 4.3 only informally, so a more formal treatment is needed here. We introduce the notion of a *well-cut recursion*.

Well-cut recursion (ASM1): We require that each dependency component D has at least one such cut c that is *well-cut*, i.e. has the following properties:

Aggregated cut (ASM1.1): The cut is placed after aggregations. More precisely, each predicate in c is the aggregation of a collecting relation along all of its produced lattice variables.

Observable monotonicity (ASM1.2): c contains all the exported predicates of D, i.e. $Exp(D) \subseteq c$. In other words, predicates that may permanently $\sqsubseteq$-decrease (when $\sqsubseteq$-increasing the input), and hence not part of the well-cut, must not be directly externally observable, only through their effect on

the well-cut. Note that this also implies that only aggregated predicates are visible externally.

Eventually $\sqsubseteq$-monotonic cut (ASM1.3): For each lattice produced in it, we require that the dependency component D be associated with one of the ordering directions $\sqsubseteq$ of the lattice such that for the above mentioned cut also demonstrates *eventually* $\sqsubseteq$-monotonic recursion for the inflationary consequence operator of any other aggregated cut: All tuples that will eventually be derived for the predicates in the well-cut, in any number of iterations, must be $\sqsubseteq$-dominated by at least one tuple eventually derivable from each possible interpretation that $\sqsubseteq$-dominates the original starting interpretation; but the two tuples need not be derived in the same iteration. Formally, for well-cut c and any aggregated cut d, $J_d \sqsubseteq J'_d$ implies $\hat{T}_d^{\omega}(I, J_d)[c] \sqsubseteq \hat{T}_d^{\omega}(I, J'_d)[c]$.

The above definition of **Well-cut recursion (ASM1)** supersedes the informally described **Eventual $\sqsubseteq$-monotonicity (ASM1)** from here on. Clarifying **Stratified recursion (ASM3)**, the aggregators of the dependency component are expected to agree with the order $\sqsubseteq$ of the corresponding lattice as mentioned in **Eventually $\sqsubseteq$-monotonic cut (ASM1.3)**.

6.3 Semantics

Note while the existence of a well-cut is assumed, the semantics and the implementation are based on a more general kind of cuts: an *eligible cut* is any aggregated cut (as in **Aggregated cut (ASM1.1)**) that contains all of $Exp(D)$ (as in **Observable monotonicity (ASM1.2)**). As eventual monotonicity is not required, it is not necessarily a well-cut. Laddder performs an iterative fixpoint computation that computes the analysis result according to the following semantics. For a dependency component D, take an arbitrary eligible cut and start at an empty J_c to compute $D_c^{\text{raw}}(I) := \hat{T}_c^{\omega}(I, \emptyset)[c]$ the least fixpoint of $\hat{T}_c$. Since the inflationary variant of the immediate consequence operator was used, the result may contain aggregate results from older iterations as well. They can be discarded by pruning the results at the cut by taking the $\sqsubseteq$-maximal (or, equivalently, latest result from each aggregation group). The result of this pruning is denoted $D_c^{\text{prune}}(I) := Prn(D_c^{\text{raw}}(I))$. The Laddder semantics of the dependency component, as far as downstream components or the end user is concerned, is the interpretations of the exported predicates $Exp(D)$ (all contained in c by **Observable monotonicity (ASM1.2)**), i.e. $D^{\text{exp}}(I) := D_c^{\text{prune}}(I)[Exp(D)]$.

6.4 Correctness Properties

Under the above assumptions, Laddder satisfies:

Termination (P1): The fixpoint computation of $\hat{T}_c$ completes in a finite number of iterations for any aggregating cut c.

Stability (P2): For any well-cut c, the results are stable under the consequence operations: both the raw results $D_c^{\text{raw}}(I)$ and their pruned form $D_c^{\text{prune}}(I)$ are fixpoints of the inflationary consequence operator $\hat{T}_c$; while the latter is also a fixpoint of the conventional immediate consequence operator T_c. Informally, this means that the Datalog rules are satisfied in the final state.

Minimal model (P3): For any well-cut c, among all possible $\hat{T}_c$-stable interpretations, $D_c^{\text{raw}}(I)$ and $D_c^{\text{prune}}(I)$ are both $\sqsubseteq$-minimal (which is only unique up to $\simeq$, due to the preorder property). Moreover among such $\sqsubseteq$-minimal interpretations, $D_c^{\text{prune}}(I)$ is set-minimal. In practice, this implies the absence of recursively self-reinforcing false tuples in the results.

Well-defined semantics (P4): The semantics $D^{\text{exp}}(I)$ is independent from the choice of the eligible cut used to compute it. In fact, as long as a well-cut exists, any other cut that satisfies **Aggregated cut (ASM1.1)** and **Observable monotonicity (ASM1.2)** (but not necessarily **Eventually $\sqsubseteq$-monotonic cut (ASM1.3)**) will yield the same final result as the well-cut. As eligible cuts are easy to find algorithmically (e.g. just include all aggregations), there is no need for the user to point out a cut to the evaluator, merely promise that a well-cut exists. Furthermore, as any eligible cut is acceptable, the evaluator may freely choose one based on performance considerations.

Compatible semantics (P5): If the standard aggregation semantics (Ross and Sagiv [30]) is also defined for a well-cut c of the component (and thus T_c is immediately monotonic), its least fixpoint (minimal model) is $D^{\text{exp}}(I)$.

We formally prove the assumptions in our technical report: https://github.com/szabta89/IncA/blob/master/papers/inca-pldi2021-extended.pdf.

7 Evaluation

In Section 3, we showed that Datalog-based whole-program analyses are amenable to incrementalization, although state-of-the-art Datalog solvers are incapable of delivering efficient incremental updates. In this section, we empirically evaluate if Laddder achieves efficient incrementalization of whole-program analyses by answering the following two questions:

Run time (RQ1) Can Laddder provide quick feedback for lattice-based inter-procedural analyses?
Memory (RQ2) Is the extra memory consumption of Laddder acceptable for IDE usage?

We implemented Laddder in Java as open source software, and we integrated it into the Viatra Query [39] incrementalization library. Given that IncA also uses Viatra Query as the solver, this allowed us to reuse the front end of IncA. Below we describe the detailed evaluation setup.

Analyses We evaluate Laddder on three lattice-based whole-program analyses. First, we use an inter-procedural k-update points-to analysis for Java that over-approximates to Top only if a points-to set grows beyond a fixed size k. A low k makes the analysis cheaper to compute while still enabling

certain compiler optimizations (e.g., inlining of virtual calls). In the examples earlier in this paper we used $k = 1$ to model a singleton points-to analysis, but for our measurements we selected $k = 5$ in our measurements, which made the analysis infer concrete points-to sets for around 40% of the program variables. The analysis is based on a points-to analysis from Doop [34], which we imported into IncA and to which we added the lattice aggregation. We emphasise that the design of the analysis is not a contribution of this paper; analysis design is orthogonal to incrementalization.

We also implemented constant propagation and interval analyses for Java. Both of these analyses are flow-sensitive and inter-procedural, the only difference is in the lattice abstraction used to track values of integer-typed variables. These analyses operate on the Jimple representation of Java programs that we extract using Soot [19].

Subject programs We benchmark the analyses against four real-world Java code bases from the Qualitas Corpus [38]: antlr (22k LOC), emma (26k LOC), pmd (61k LOC), and ant (105k LOC). To compare with the state of the art, we added a smaller code base: minijavac (6.5k LOC). For the points-to analysis, we used the Doop fact extractor to produce facts that describe the program elements (e.g. function signatures) and their relationships (e.g. parameters of function) from the subject program and *the JRE*. For the constant propagation and interval analyses, we use Soot to extract the Jimple AST and the ICFG of the programs. We use the ICFG as input to these analyses.

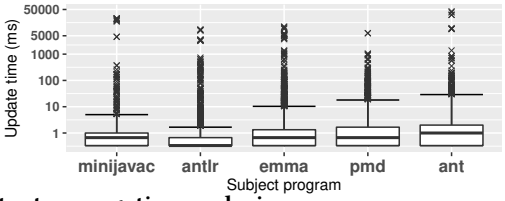
Program changes We run the analyses on the entire code bases to initialize the incremental algorithm. Unfortunately, there is no standard benchmark for incremental program changes available, so we chose to synthesize program changes that are likely to affect the analysis results. For the points-to analysis, we randomly delete and re-insert 1000 object allocation sites. We chose to focus on allocation sites because these are simple atomic changes that directly affect the results of the points-to analysis, whereas many other changes are no-ops for a points-to analysis. For the constant propagation and interval analyses, we randomly replace 1000 numeric literals and field reads with the zero literal. Again, these kinds of program changes are "worst case" for these analyses in terms of the workload because there is a high chance that many assignments to variables will be affected due to inter-procedurality. We acknowledge that the way how we synthesize these changes is a threat to validity, and future work should consider more realistic editing scenarios with source code-level changes (in contrast to low-level fact changes in input relations).

We consider every one of the synthesized changes individually. After each change, we run the incremental analysis until fixpoint and measure the update time of the analyses. We ran each benchmark 4 times, dropped the result of the first run to account for JVM warmup, and report the average times of the remaining three runs. We ran the benchmarks on an Intel Core i7-6820HQ at 2.7 GHz with 16 GB of RAM, running 64-bit OSX 10.12.6 and Java 1.8.0_121.

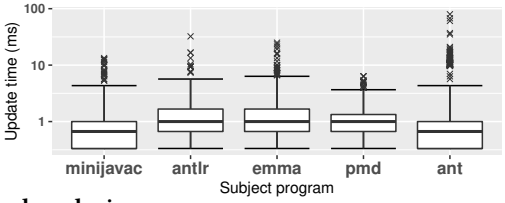
7.1 Evaluating Run time (RQ1)

The range of the initialization times of the analyses on the 5 code bases are as follows: points-to analysis 57–172 s, constant propagation analysis 5–23 s, and interval analysis 3–23 s. These delays are acceptable because they are (i) one-off costs only and (ii) possibly can be precomputed. The following boxplots show the incremental update times of LADDDER for the various analyses:

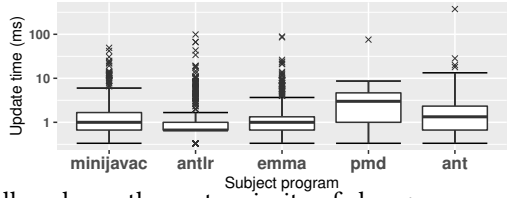
k-update points-to analysis:



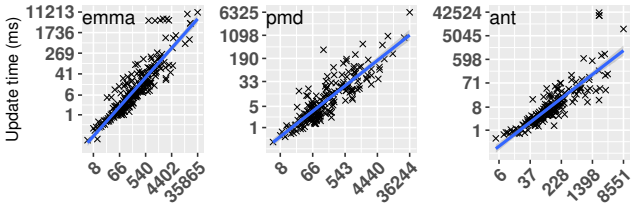
Constant propagation analysis:



Interval analysis:



For all analyses, the vast majority of changes are processed in less than 100 ms. However, the points-to analysis has a few outliers requiring up to 50 s to update the analysis result, although 99% of them are still faster than 1000 ms. To better understand outliers, we relate the update times of the points-to analysis to the impact of the program change:



For space reasons, we only show the diagram for the three largest code bases, but when we fit a linear regression on the log-log plots, we found that the relationship $\texttt{time} \sim \mathit{impact}^{1.5}$ approximately holds for *all* subject programs. We believe that dealing with rare cases of high update times is acceptable in practice, given that the vast majority of changes are handled in milliseconds.

7.2 Evaluating Memory (RQ2)

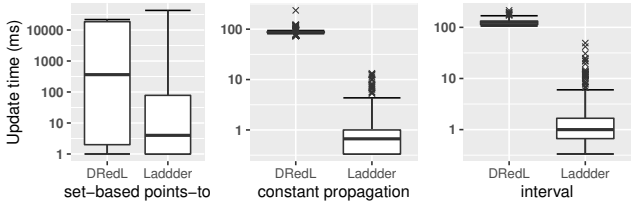
We measured the memory use of Laddder by taking the reachable JVM heap size after initializing the analysis and subtracting the size from before the analysis. Throughout the program changes, the memory use of Laddder remained roughly the same. The range of the memory use of the analyses on the 5 code bases are as follows: points-to analysis 3.7–8.7 GB, constant propagation analysis 0.6–2.3 GB, and interval analysis 0.8–2.9 GB. MPS used around 2 GB before running the analysis, which means the largest overhead is ~4.5 times larger than that. These values may seem high, but we emphasize that the analyses are inter-procedural, also analyzing parts of the *JRE*. We have concrete plans to reduce the memory overhead of Laddder; for example, by supporting n-ary joins [25] instead of binary ones; or by using larger-grained caching instead of caching every relational algebra operation [11] (e.g. joins, selections).

7.3 Comparison with DRed_L

Many incremental Datalog solvers are in principle compatible with recursive aggregation, but only IncA and its solver DRed_L actually support it [36]. Therefore, we compare the expressiveness and running times of DRed_L and Laddder.

Recall that our new aggregation semantics is the first to relax the standard per-rule monotonicity requirement. Thus, many analyses supported by Laddder are incompatible with prior incremental Datalog solvers. In particular, the k-update points-to analysis relies on this relaxation and cannot be run with DRed_L. The constant propagation and interval analyses, on the other hand, are compatible with DRed_L because there are no conditional rules based on previously derived constants. However, the relaxed semantics can quickly become necessary: For example, DRed_L fails if we use derived constants to improve the control-flow (e.g., constant if-conditions).

We compare the performance of DRed_L and Laddder on the minijavac benchmark. We emphasise that the measurements of DRed_L and Laddder use the same analysis specification and back end library, except that we configured different fixpoint algorithms. Instead of the k-update points-to analysis, we revert to a standard powerset-based points-to analysis, so that DRed_L can run it.



Compared to DRed_L, Laddder achieves faster update times and it does so more consistently. We conclude that the increased expressiveness of Laddder is relevant in practice and that its update times improve the state of the art.

Finally, initializing the powerset-based points-to analysis took 35.2 s for DRed_L and 62.8 s for Laddder, constant propagation analysis took 1.4 s for DRed_L and 2.6 s for Laddder, and interval analysis took 4.7 s for DRed_L and 5.4 for Laddder. These numbers show that the initialization time of DRed_L is consistently better, the overhead of Laddder ranges between 15% up to 86%. As DRed_L does not maintain timelines, its from-scratch initialization phase is essentially a standard bottom-up Datalog fixpoint evaluation. We can therefore regard DRed_L initialization times as stand-ins for non-incremental algorithms[3], and conclude that the overheads of incrementalization are manageable.

8 Related Work

Analysis frameworks Several approaches incrementalize analyses that use the powerset lattice [10, 18, 28, 41], which can express many interesting (inter-procedural) data-flow analyses (e.g. set-based points-to or initialized variables). But as argued by Section 4.4 and by Madsen et al. [21, Section 1], custom lattices are more powerful. A restriction to the powerset lattice rules out important analysis domains, including those reasoning about strings [6], intervals [40], distributed systems [5], or even our singleton points-to analysis.

Several frameworks analyze procedures in isolation to derive summaries that describe their effect. Such frameworks can usually deal with code changes efficiently because many previously computed summaries can be reused and only need to be re-composed with the newly derived summaries of changed procedures. This way, Reviser [3] can handle the entire Maven repository, and Infer [8] can deal with Facebook-scale code bases. These tools are primarily used at code reviewing time to deliver feedback within several minutes. However, summary-based approaches work efficiently if summaries only encode procedure-local information. For example, it was believed for a long time that reasoning about points-to information of the heap cannot be efficiently summarized [4]. Only recently did Späth et al. show how to summarize certain subtasks of a powerset-based points-to analysis [35]. It is unclear if this approach also works for our singleton points-to analysis, or if other analyses using custom lattices and aggregation can be efficiently summarized.

DDF has a reference implementation in Rust available open source.[4] Apart from Laddder, there are other frameworks that also build on DDF. For example, Differential Datalog (DDLog) is a Datalog-based logic programming language [31]. DDLog extends Datalog in several ways; e.g. with an expression language, rich type system, and module system. DDLog directly uses the DDF reference implementation for incrementalization and experiments show that it can handle

[3] Actual non-incremental tools such as Flix [21] may of course implement additional optimizations, but these are mostly orthogonal to incrementalization, and hence not relevant for supporting our claims in this paper.
[4] https://github.com/TimelyDataflow/differential-dataflow

Doop analyses efficiently [29]. 3DF [16] follows a similar approach, as it also compiles Datalog programs into DDF. The primary application area of 3DF is to provide live updates in streaming analytics systems. A key difference between these approaches and LADDDER is in the aggregation support. First, LADDDER uses an inflationary aggregation semantics which enables looser requirements on analysis definitions (see Section 4.3). Second, while the traditional DDF algorithm is compatible with lattice-based aggregation, these approaches do not actually provide a practical implementation for aggregation over custom lattices. As the comparison to DRed_L in in Section 7 showed, both the aggregation semantics and the sequential architecture play a crucial role in efficiently scaling to lattice-based whole-program analyses.

Flix [21] is a non-incremental analysis framework that uses Datalog and lattices. Analysis developers in Flix use a Scala-based core functional language to define lattice operations. Flix puts emphasis on safe and sound analysis definitions in the front end because its compiler automatically verifies the mathematical properties of lattice operations, for example, if a `lub` operation is indeed monotone wrt. the partial order of the lattice [20]. Currently, LADDDER requires analysis developers to reason about their code and ensure e.g. eventual $\sqsubseteq$-monotonicity (Section 4.3). Flix also uses the standard Ross and Sagiv aggregation semantics just like the previous DDF-based approaches.

Demand-driven evaluation LADDDER eagerly updates all derived relations after a program change. Coupled with high-impact changes, this strategy can (rarely) lead to update times exceeding 10 sec (Section 7). Demand-driven evaluation, i.e. the idea to only compute information that is needed, may help here. For example, Boomerang is a highly-precise points-to analysis that allows clients to query points-to information for a given context, which can be answered much faster than analyses on the whole program [35]. Do et al. [9] prioritizes the computation of analysis results that are relevant for feedback around the actively edited code parts.

Datalog semantics Inflationary semantics [14] is a well-known Datalog concept, but it generally lacks minimal model guarantees and is often associated with nondeterminism [1, 15]. Our application of inflationary semantics is novel in providing, under specific assumptions, important correctness guarantees, including both minimality and termination. The equivalence between LADDDER and conventional semantics [30], if the latter exists, can be considered a reverse application of the concept *pre-mappability* [42].

Comparison of DDF and DRed performance DRed suffers from an over-deletion problem, where a large amount of tuples could be deleted only to be later re-derived. This shows up especially when frequently used library functions are affected. DDF does not have this specific problem, but it may also do unnecessary steps if e.g. a program change leads to a new derivation of an existing tuple at an earlier iteration round: Potentially all consequences of that tuple will have to be computed again, at a shifted timestamp. Motik et al. concludes [24] that neither algorithm[5] is universally superior to the other: It is possible to construct inputs that force either solution to do significantly more work than necessary.

Regarding memory, DDF clearly needs more space than DRed: LADDDER associates each tuple with (a sparsely stored) differential count timeline, as opposed to a single support count (as in [36]). DDF would generally also require a differential existence timeline, but due to its inflationary nature, LADDDER can represent it as a single timestamp of appearance. This overhead is highly dependent on the number of different timestamps a tuple is derived at. In both cases, practical implications can only be determined empirically. Experiments in Section 7 reveal LADDDER to be significantly faster than DRed (IncA) with an acceptable memory cost.

9 Conclusions

We presented an approach for the efficient incrementalization of lattice-based whole-program analyses. We used Datalog and lattices in the front end to enable a wide range of static analyses. Our new solver LADDDER uses inflationary aggregation semantics to loosen the monotonicity requirements on analysis definitions compared to prior approaches. LADDDER also combines the DDF computation model with efficient aggregator architecture for improved performance. In our evaluation, we first verified that lattice-based whole-program analyses are amenable to incrementalization because high-impact changes happen rarely throughout a random series of changes on real-world code bases. Then, we showed that LADDDER delivers the performance interactive applications in IDEs need as it updates results in sub-second time for more than 99 % of all changes, typically in a few milliseconds. We pay the price for the fast updates with memory: The overhead can get large, but not prohibitive.

Acknowledgments

We thank the anonymous reviewers for their useful feedback on this paper. We thank Yannis Smaragdakis for the idea of the k-update points-to analysis. The third author performed some of his research as a member of the MTA-BME Lendület Research Group on Cyber-Physical Systems, and was partially supported by the János Bolyai Research Scholarship (http://mta.hu/bolyai-osztondij) of the Hungarian Academy of Sciences; ÚNKP-18-4 New National Excellence Program (http://www.unkp.gov.hu) of the Ministry of Human Capacities; as well as the ÚNKP-19-4 New National Excellence Program (http://www.unkp.gov.hu) and the charter of bolster at BME both issued by the NRDI Office under the auspices of the Ministry For Innovation and Technology.

[5]The study makes no reference to DDF, but their algorithm "Recursive Counting" is apparently an independent rediscovery of the core idea of DDF, as adapted for non-aggregating Datalog. A third method [23] is included as well, which is not relevant here due to the lack of aggregation support.

References

[1] Serge Abiteboul, Daniel Deutch, and Victor Vianu. 2014. Deduction with Contradictions in Datalog. In *International Conference on Database Theory*. Athens, Greece. https://hal.inria.fr/hal-00923265

[2] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. 2012. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. *Proc. VLDB Endow.* 5, 10 (June 2012), 968–979. https://doi.org/10.14778/2336664.2336670

[3] Steven Arzt and Eric Bodden. 2014. Reviser: Efficiently Updating IDE-/IFDS-based Data-flow Analyses in Response to Incremental Program Changes. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. ACM, New York, NY, USA, 288–298. https://doi.org/10.1145/2568225.2568243

[4] Eric Bodden. 2018. The Secret Sauce in Efficient and Precise Static Analysis: The Beauty of Distributive, Summary-based Static Analyses (and How to Master Them). In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops* (Amsterdam, Netherlands) *(ISSTA '18)*. ACM, New York, NY, USA, 85–93. https://doi.org/10.1145/3236454.3236500

[5] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. 2012. Logic and Lattices for Distributed Programming. In *Proceedings of the Third ACM Symposium on Cloud Computing* (San Jose, California) *(SoCC '12)*. ACM, New York, NY, USA, Article 1, 14 pages. https://doi.org/10.1145/2391229.2391230

[6] Giulia Costantini, Pietro Ferrara, and Agostino Cortesi. 2011. Static Analysis of String Values. In *Proceedings of the 13th International Conference on Formal Methods and Software Engineering* (Durham, UK) *(ICFEM'11)*. Springer-Verlag, Berlin, Heidelberg, 505–521.

[7] Patrick Cousot and Radhia Cousot. 2004. *Basic Concepts of Abstract Interpretation*. Springer US, Boston, MA, 359–366. https://doi.org/10.1007/978-1-4020-8157-6_27

[8] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling Static Analyses at Facebook. *Commun. ACM* 62, 8 (July 2019), 62–70. https://doi.org/10.1145/3338112

[9] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. 2017. Just-in-time Static Analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. ACM, New York, NY, USA, 307–317. https://doi.org/10.1145/3092703.3092705

[10] Michael Eichberg, Matthias Kahl, Diptikalyan Saha, Mira Mezini, and Klaus Ostermann. 2007. Automatic Incrementalization of Prolog Based Static Analyses. In *Proceedings of the 9th International Conference on Practical Aspects of Declarative Languages* (Nice, France) *(PADL'07)*. Springer-Verlag, Berlin, Heidelberg, 109–123. https://doi.org/10.1007/978-3-540-69611-7_7

[11] Charles L. Forgy. 1982. Rete: A fast algorithm for the many pattern/-many object pattern match problem. *Artificial Intelligence* 19, 1 (1982), 17 – 37. https://doi.org/10.1016/0004-3702(82)90020-0

[12] Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. 2013. Datalog and Recursive Query Processing. *Found. Trends databases* 5, 2 (Nov. 2013), 105–195. https://doi.org/10.1561/1900000017

[13] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. 1993. Maintaining Views Incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data* (Washington, D.C., USA) *(SIGMOD '93)*. ACM, New York, NY, USA, 157–166. https://doi.org/10.1145/170035.170066

[14] Y. Gurevich and S. Shelah. 1985. Fixed-point extensions of first-order logic. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*. 346–353. https://doi.org/10.1109/SFCS.1985.27

[15] Antonella Guzzo and Domenico Saccà. 2005. Semi-Inflationary DATALOG: A Declarative Database Language with Procedural Features. *AI Commun.* 18, 2 (April 2005), 79–92. http://dl.acm.org/citation.cfm?id=1218852.1218854

[16] Nikolas Göbel. 2018. Incremental Datalog with Differential Dataflows. Retrieved 2019-10-11 from https://www.nikolasgoebel.com/2018/09/13/incremental-datalog.html.

[17] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 422–430.

[18] Uday Khedker. 1995. *A Generalised Theory of Bit Vector Data Flow Analysis*. Ph.D. Dissertation. Department of Computer Science and Engineering, IIT Bombay.

[19] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, Vol. 15. 35.

[20] Magnus Madsen and Ondřej Lhoták. 2018. Safe and Sound Program Analysis with Flix. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. ACM, New York, NY, USA, 38–48. https://doi.org/10.1145/3213846.3213847

[21] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. From Datalog to Flix: A Declarative Language for Fixed Points on Lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. ACM, New York, NY, USA, 194–208. https://doi.org/10.1145/2908080.2908096

[22] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *CIDR*.

[23] Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks. 2015. Incremental Update of Datalog Materialisation: The Backward/Forward Algorithm. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence* (Austin, Texas) *(AAAI'15)*. AAAI Press, 1560–1568. http://dl.acm.org/citation.cfm?id=2886521.2886537

[24] Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks. 2019. Maintenance of Datalog Materialisations Revisited. *Artificial Intelligence* 269 (04 2019). https://doi.org/10.1016/j.artint.2018.12.004

[25] Hung Q. Ngo. 2018. Worst-Case Optimal Join Algorithms: Techniques, Results, and Open Problems. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Houston, TX, USA) *(SIGMOD/PODS '18)*. ACM, New York, NY, USA, 111–124. https://doi.org/10.1145/3196959.3196990

[26] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of program analysis*. Springer.

[27] André Pacak, Sebastian Erdweg, and Tamás Szabó. 2020. A Systematic Approach to Deriving Incremental Type Checkers. *Proc. ACM Program. Lang.* 4, OOPSLA (2020).

[28] L. L. Pollock and M. L. Soffa. 1989. An Incremental Version of Iterative Data Flow Analysis. *IEEE Trans. Softw. Eng.* 15, 12 (Dec. 1989), 1537–1549. https://doi.org/10.1109/32.58766

[29] Argyro A. Ritsogianni. 2019. Incremental Static Analysis with Differential Datalog.

[30] Kenneth A. Ross and Yehoshua Sagiv. 1992. Monotonic Aggregation in Deductive Databases. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (San Diego, California, USA) *(PODS '92)*. ACM, New York, NY, USA, 114–126. https://doi.org/10.1145/137097.137852

[31] Leonid Ryzhyk and Mihai Budiu. 2019. Differential Datalog. In *Datalog 2.0 2019 - 3rd International Workshop on the Resurgence of Datalog in Academia and Industry co-located with the 15th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2019) at the Philadelphia Logic Week 2019, Philadelphia, PA (USA), June 4-5, 2019*. 56–67.

[32] Robert Sedgewick and Kevin Wayne. 2011. *Algorithms* (4th ed.). Addison-Wesley Professional.

[33] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. Trends Program. Lang.* 2, 1 (April 2015), 1–69. https://doi.org/

10.1561/2500000014

[34] Yannis Smaragdakis and Martin Bravenboer. 2011. Using Datalog for Fast and Easy Program Analysis. In *Proceedings of the First International Conference on Datalog Reloaded* (Oxford, UK) *(Datalog'10)*. Springer-Verlag, Berlin, Heidelberg, 245–251. https://doi.org/10.1007/978-3-642-24206-9_14

[35] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 22:1–22:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.22

[36] Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. 2018. Incrementalizing Lattice-based Program Analyses in Datalog. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 139 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276509

[37] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: A DSL for the Definition of Incremental Program Analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. ACM, New York, NY, USA, 320–331. https://doi.org/10.1145/2970276.2970298

[38] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. 2010. The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *2010 Asia Pacific Software Engineering Conference*. 336–345. https://doi.org/10.1109/APSEC.2010.46

[39] Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, and Zoltán Ujhelyi. 2016. Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Software & Systems Modeling* 15, 3 (01 Jul 2016), 609–629. https://doi.org/10.1007/s10270-016-0530-4

[40] Yawen Wang, Yunzhan Gong, Junliang Chen, Qing Xiao, and Zhaohong Yang. 2008. An application of interval analysis in software static analysis. In *Embedded and Ubiquitous Computing, 2008. EUC'08. IEEE/IFIP International Conference on*, Vol. 2. IEEE, 367–372.

[41] Frank Kenneth Zadeck. 1984. *Incremental data flow analysis in a structured program editor*. Vol. 19. ACM.

[42] Carlo Zaniolo, Mohan Yang, Ariyam Das, Alexander Shkapsky, Tyson Condie, and Matteo Interlandi. 2017. Fixpoint semantics and optimization of recursive Datalog programs with aggregates. *Theory and Practice of Logic Programming* 17, 5-6 (2017), 1048–1065. https://doi.org/10.1017/S1471068417000436