Incrementalizing Lattice-Based Program Analyses

TAMÁS SZABÓ, itemis, Germany and Delft University of Technology, Netherlands
GÁBOR BERGMANN, Budapest University of Technology and Economics / MTA-BME Lendület Research Group on Cyber-Physical Systems, Hungary
SEBASTIAN ERDWEG, Delft University of Technology, Netherlands
MARKUS VOELTER, independent and itemis, Germany

Program analyses detect errors in code but have to trade off precision, recall, and performance. However, when code changes frequently as in an IDE, repeated re-analysis from-scratch is unnecessary and leads to poor performance. Incremental program analysis promises to deliver fast feedback after a code change by deriving a new analysis result from the previous one, and prior work has shown that order-of-magnitude performance improvements are possible. However, existing frameworks for incremental program analysis only support Datalog-style relational analysis, but not lattice-based analyses that derive and aggregate lattice values. To solve this problem, we take our existing IncA incremental program analysis framework that supports relational analyses, and we extend it with lattice-based computations. Our extension is based on a novel algorithm that enables the incremental maintenance of recursive lattice-value aggregation, which occurs when analyzing code with cyclic control flow by fixpoint iteration. To demonstrate our approach, we realized strong-update points-to analysis and string analyses for Java in IncA and present performance measurements that demonstrate incremental analysis updates within milliseconds.

Additional Key Words and Phrases: Static Analysis, Incremental Computing, Domain-Specific Language, Language Workbench, Datalog, Lattice

1 INTRODUCTION

Static program analyses are fundamental for the software development pipeline: They form the basis of compiler optimizations, enable refactorings in IDEs, and detect a wide range of errors. For program analyses to be useful in practice, they must balance multiple requirements, most importantly precision, recall, and performance. In this work, we study a novel approach to improving the performance of program analyses through *incrementality*, without affecting precision or recall. After a change to the subject program (program under analysis), an incremental analysis updates a previous analysis result instead of re-analyzing the code from scratch. Incrementality can provide order-of-magnitude speedups [Bhatotia et al. 2011; Sumer et al. 2008] because small input changes only trigger small output changes, with correspondingly small computational cost. Prior research has shown that incrementality is particularly useful for program analysis. For example, incrementality was reported to speed up FindBugs checks 65x [Mitschke et al. 2014] and C points-to analysis 243x [Szabó et al. 2016]. Unfortunately, existing approaches for efficiently incrementalizing static analyses are limited in expressiveness due to limitations of the Datalog solvers they rely on.

Datalog is a logic programming language that is frequently used for program analyses [Avgustinov et al. 2016; Green et al. 2013; Smaragdakis and Bravenboer 2011; Whaley and Lam 2004]. Incremental Datalog solvers efficiently update query results based on code changes [Gupta and Mumick 1995]. However, existing incremental solvers do not support recursive user-defined aggregation, which means that an analysis can only compute fixpoints over sets but not over any other data structure such as intervals. The lack of support for recursive aggregation severely limits the

Authors' addresses: Tamás Szabó, itemis, Germany , Delft University of Technology, Netherlands; Gábor Bergmann, Budapest University of Technology and Economics / MTA-BME Lendület Research Group on Cyber-Physical Systems, Hungary; Sebastian Erdweg, Delft University of Technology, Netherlands; Markus Voelter, independent , itemis, Germany.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

applicability to program analyses, which often aggregate over *lattices* using least upper bound or greatest lower bound.

In this paper, we present a new algorithm DRed_L that incrementally solves recursive Datalog queries containing user-defined aggregations. However, in general, recursive user-defined aggregations may require complete unrolling of previous computations when aggregation inputs change. Instead, DRed_L solves a subproblem and requires that (i) aggregations operate on lattices and that (ii) recursive aggregations are monotonic; both requirements are readily satisfied by program analyses. The key insight of DRed_L is that the monotonicity of recursive aggregations allows for efficient handling of monotonic input changes. This is necessary for correctness and essential for efficiency. We have formally verified that DRed_L is correct and yields the exact same result as computing the query from scratch.

We built a whole analysis framework around DRed_L. The starting point was our prior IncA incremental analysis framework [Szabó et al. 2016]. IncA provides a domain-specific language (DSL) for the definition of program analysis which relies on relations, *but no lattice operations*. IncA represents subject programs as base relations, caches the results of derived relations, and propagates program changes bottom-up from base relations to derived relations, which yield the analysis result in the end. Importantly, the runtime system of IncA also did not support aggregations over lattices in IncA. We extended IncA in the following way. We added lattices and aggregations over lattices to the DSL. Then, we integrated DRed_L into the runtime system of IncA, which enable analysis developers to incrementally compute fixpoints over user-defined lattices. Finally, we extended the compiler and integrated the framework into the MPS language workbench[1]. Our implementation is available open source.[2]

To evaluate the applicability and performance of IncA, we have implemented two Java analyses in IncA adapted from the literature: Strong update points-to analysis [Lhoták and Chung 2011] as well as character-inclusion and prefix-suffix string analysis [Costantini et al. 2011]. We ran performance measurements for both analyses on four real-world Java projects with up to 70 KLOC in size. We measured the initial non-incremental run time and the incremental update time after random as well as idiomatic code changes. Our measurements reveal that incremental update times are up to four orders of magnitudes faster than a non-incremental run. Additionally, we also benchmarked the memory requirement of the system because incrementalization relies on extensive caching. Our evaluation shows that the dominating factor in the memory usage is the precision of the analysis. For example, a flow-sensitive strong update points-to analysis in IncA requires two times the IDE memory usage, which is large but not prohibitive.

In summary, we make the following contributions:

- We identify and describe the key challenge for incremental lattice-based program analysis: cyclic reinforcement of lattice values (Section 2).
- We present the IncA approach for incremental lattice-based program analysis by embedding in Datalog, requiring recursive aggregation (Section 3).
- We develop and verify DRed_L, the first algorithm to incrementally solve recursive Datalog rules containing user-defined aggregations (Section 4).
- We implemented DRed_L and IncA as open-source software (Section 5).
- We demonstrate the applicability and evaluate the performance of DRed_L and IncA on two existing Java analyses (Section 6).

[1] https://www.jetbrains.com/mps
[2] https://github.com/szabta89/IncA

Interval relies on the CFlow relation to deduce control flow

A

CFlow	
src	trg
N1	N2
N2	N3
N3	N4
N4	N5
N5	N2

B C D E

IntervalAfter					
stmt	var	iv	iv	iv	iv
(N1) x = 7,y = 0; ① -1 ③ 0 ② cond ? -1 : 0	x	[7,7]	[7,7]	[7,7]	[7,7]
	y	[0,0]	[-1,-1]	[-1,0]	[0,0]
(N2) while (read()) {	x	[7,11]	[7,11]	[7,11]	[7,11]
	y	[0,∞)	[-1,∞)	[-1,∞)	[0,∞)
(N3) x = 9;	x	[9,9]	[9,9]	[9,9]	[9,9]
	y	[0,∞)	[-1,∞)	[-1,∞)	[0,∞)
(N4) x = x + 2;	x	[11,11]	[11,11]	[11,11]	[11,11]
	y	[0,∞)	[-1,∞)	[-1,∞)	[0,∞)
(N5) y = cond ? y : y + 1; }	x	[11,11]	[11,11]	[11,11]	[11,11]
	y	[0,∞)	[-1,∞)	[-1,∞)	[0,∞)

Fig. 1. Relations of the running example. (A) CFlow relation encoding the CFG of the subject program. (B) Interval relation showing the value ranges of variables per CFG node. (C) - (E) Updates in the Interval relation after changing the initializer of y through the series `0` → `-1` → `cond ? -1 : 0` → `0`. Red color shows the changes compared to previous values.

2 CHALLENGES OF INCREMENTAL PROGRAM ANALYSIS

In this section we introduce a running example to illustrate the challenge of incremental lattice-based static program analysis. The example is a flow-sensitive interval analysis for Java, which reports the possible value ranges of program variables. As a starting point, we choose logic programming in Datalog because the use of Datalog for program analysis is well-documented [Green et al. 2013; Smaragdakis and Bravenboer 2011], and there exist incremental Datalog solvers [Gupta et al. 1993; Mitschke et al. 2014; Szabó et al. 2016]. It will become clear in this section that incremental program analysis requires incremental recursive aggregation, which existing solvers fail to support.

Interval Analysis with Datalog. A flow-sensitive interval analysis keeps track of the value ranges of variables at each control flow location. We first analyze a program's control flow by constructing a relation `CFlow(src,trg)`, where each `(src,trg)` tuple represents an edge of the control flow graph (CFG). We define `CFlow(src,trg)` through Datalog rules.

A Datalog rule r has form `h :- a1,...,an` where `h` is the *head* of the rule and $a_1, \ldots, a_n$ are the (possibly negated) *subgoals*. In standard Datalog (as opposed to some extensions of it [Green et al. 2013]), the rule head and subgoals are of the form $R(t_1, \ldots, t_k)$ where R is the name of a relation and t_i is a term that is either a variable or a constant value. Each rule is interpreted as a universally quantified implication: For each substitution of variables that satisfies all subgoals a_i, the head `h` must also hold. Given a rule head $R(t_1, \ldots, t_k)$, the satisfying substitutions of the terms $t_1, \ldots, t_k$ yield the tuples of relation `R`. We refer the reader to [Green et al. 2013] for further literature on Datalog. The following rules compute `CFlow`:

```
CFlow(from, to)      :- Next(from, to), ¬IfElseStmt(from).
CFlow(while, start) :- WhileStmt(while), FirstChild(while, start).
CFlow(last, while)   :- WhileStmt(while), LastChild(while, last).
```

First, control flows from a statement to its successor statement if the former is not an `IfElseStmt` (or any other construct that redirects control flow). Second, control flows from a `while` head to the first child in its body. And third, control flows from the last statement of a loop body to the loop head. Figure 1B shows an example subject program with statements `N1` to `N5`; its CFG as computed by `CFlow` is shown in Figure 1A.

We now construct a flow-sensitive interval analysis on top of this CFG. We define a relation `IntervalAfter(stmt,var,iv)`, where a tuple `(s,v,i)`∈`IntervalAfter` describes that after execution of statement `s`, variable `v` must take a value in interval `i`. Terms `s` and `v` are existing program elements, and `i` is a computed value from the *interval lattice*. To compute the interval `i` of variable

v after statement s, the analysis must consider two cases. If s (re)assigns v, we compute v's interval based on the assignment expression. Otherwise v's interval does not change, and we propagate the interval associated with v from before the statement:

```
IntervalAfter(stmt, var, iv) :- AssignsNewValue(stmt, var), AssignedInterval(stmt, var, iv).
IntervalAfter(stmt, var, iv) :- ¬AssignsNewValue(stmt, var), IntervalBefore(stmt, var, iv).
```

In order to figure out this previous interval, we query the CFG to find all control-flow predecessors of the current statement and collect the variable's `IntervalAfter` for each of the predecessors in a relation `PredecessorIntervals`. As the interval containment *partial order* forms a *lattice*, we can obtain the smallest interval containing all predecessor intervals of the current statement. It is computed by aggregating such predecessors using the *least upper bound* (`lub`) lattice operation:

```
PredecessorIntervals(stmt, var, pred, iv) :- CFlow(pred, stmt), IntervalAfter(pred, var, iv).
IntervalBefore(stmt, var, lub(iv)) :- PredecessorIntervals(stmt, var, _pred, iv).
```

Note that `IntervalAfter` and `IntervalBefore` are *mutually recursive* and also induce *cyclic tuple dependencies* whenever `CFlow` is cyclic. This is typical for flow-sensitive program analyses. We obtain the final analysis result by computing the (least) fixpoint of the Datalog rules. The computation always terminates because the rules in our analysis are monotonic wrt. the partial order of the interval lattice, and because we use widening to ensure that the partial order does not have infinite ascending chains [Cousot and Cousot 2004]. For example, as shown in Figure 1B, the interval analysis computes tuples `(N2,x,[7,11])` and `(N2,y,[0,∞])` for the loop head. Tuple `(N2,x,[7,11])` is the result of joining `(N1,x,[7,7])` and `(N5,x,[11,11])`. Tuple `(N2,y,[0,∞])` is the result of the fixpoint computation, joining all intermediate intervals `lub([0,0], [0,1], [0,2],...)` of `y` before `N2`.

Incremental Interval Analysis. The analysis presented up to here is standard; we now look at incrementalization. Existing incremental Datalog solvers cannot handle this analysis because they do not support recursive aggregation. The root cause of the limitation is that realistic subject programs typically have loops or recursive data structures. Analyzing them requires fixpoint iteration over cyclic control flow or data-flow graphs, which, in turn, requires recursive aggregation. Indeed, there is no known algorithm for incrementally updating aggregation results over recursive relations with cyclic tuple dependencies.

The goal of an *incremental* analysis is to update its results in response to the subject program's evolution over time with *minimal computational effort*. For illustrative purposes, we consider three changes ①, ②, and ③ of the initializer of `y` in Figure 1B, and then we review what is expected from an incremental analysis in response to them. First, we change it from `0` to `-1`. Figure 1C shows the updated `iv` column of relation `IntervalAfter`. While this change does not affect previous results of `x`, the intervals assigned to `y` require an update at all CFG nodes. After a subsequent change from `-1` to `(cond ? -1 : 0)` we expect that only the interval at `N1` gets updated (see Figure 1D) because the analysis of the loop already considered `y=0` before. Third, consider a final change from `(cond ? -1 : 0)` to `0`. Now the analysis does not have to consider `y=-1` anymore, and we expect an update to all lower bounds of the intervals assigned to `y` as shown in Figure 1E.

How can we update a previous result with minimal computational effort in response to changes ①, ②, and ③? Fundamentally, we propagate program changes bottom-up through the analysis' Datalog rules, starting at the changed program element. For example, a change to a variable's initializer directly affects the result of `AssignedInterval`, which is then used in the first rule of `IntervalAfter`. If the new initializer has a different interval than the old initializer, `AssignedInterval` propagates a derived change to `IntervalAfter`. A derived change leads to the deletion

of the old interval and the insertion of the new interval into the relevant relation. In turn, `IntervalAfter` propagates such a change to `PredecessorIntervals`, which propagates it to the aggregation in `IntervalBefore`, which propagates it to the second rule of `IntervalAfter`, and so on. This way we can handle change ①, where we delete `(N1,y,[0,0])`, insert `(N1,y,[-1,-1])`, and propagate these changes.

Handling change ② efficiently is more challenging: We must augment our strategy to avoid deleting `(N1,y,[-1,-1])` before inserting `(N1,y,[-1,0])`, so that we can reuse previous results. Failing to do that would result in re-computing the exact same result, which is unnecessary excess work degrading efficiency.

The toughest problem is change ③, where we delete `(N1,y,[-1,0])` and insert `(N1,y,[0,0])`. The problem manifests at CFG node `N2` and is due to the loop in the CFG and the `lub` aggregation over it. Before change ③, `N2`'s CFG predecessors `N1` and `N5` report intervals `[-1,0]` and `[-1,∞)` for `y`, as shown in Figure 1D. When propagating deletion `(N1,y,[-1,0])` and insertion `(N1,y,[0,0])` to `N2`, we expect to replace `(N2,y,[-1,∞))` by `(N2,y,[0,∞))`. However, the aggregation in `IntervalBefore` for `N2` also has to consider interval `[-1,∞)` from `N5` and `lub([0,0], [-1,∞)) = [-1,∞)`. However, note how this yields the *wrong* result for `N2` because the very reason for `[-1,∞)` at `N5` is the old initializer at `N1`, which just got deleted. We call this situation *cyclic reinforcement* of lattice values. An incremental analysis must carefully handle cyclic reinforcements to avoid computing incorrect analysis results. The design of an algorithm for executing program analyses incrementally and correctly is the central contribution of this paper.

Problem Statement. Our goal is to incrementalize program analyses with recursive, user-defined aggregations. Our solution must satisfy the following requirements:

Correctness (R1): An incremental analysis must produce the exact same result as re-running the analysis from scratch. For example, it must rule out cyclic reinforcements.

Efficiency (R2): An incremental analysis must update its results with minimal computational effort. We expect from an incremental analysis to be significantly faster than the corresponding non-incremental analysis, so that we can use analyses in interactive editing scenarios.

Expressiveness (R3): Our solutions should apply systematically to any program analyses that use (one or more) user-defined lattices and user-defined aggregation operators.

Note that we do not mention *precision* or *recall* because these are the responsibility of the analysis developer when designing the lattices and their operations. There is no limitation on the design of those by our solution. However, a more precise analysis might take more time to compute (e.g. because we reach a fixpoint in more iterations), so it has an impact on performance, just like in a non-incremental analysis.

3 INCREMENTAL LATTICE-BASED PROGRAM ANALYSIS WITH IncA

IncA is a framework for specifying and incrementally executing program analyses that perform recursive lattice-based aggregations. IncA provides a domain-specific language (DSL) for the specification of lattices and program analyses. As we describe in Section 5, the DSL compiles to Datalog-style rules similar to the ones we have shown in Section 2. In the present section, we focus on the runtime system of IncA and how it executes analyses incrementally.

As pointed out in Section 1, this work extends our prior work on incremental program analysis. The prior version of IncA could only incrementally analyze relations between program elements; for example, we used it for an incremental points-to analysis for C. However, because of fundamental limitations in the runtime system, there was no support for lattices nor aggregation over them. In this paper, we solve these limitations by addig support for lattice-based aggregations, which, as we have argued above and validate in Section 6, enables a much wider range of program analyses.

3.1 Incremental Execution of Non-Recursive Analyses

The runtime system of IncA incrementally computes relations described by Datalog-style rules. We briefly review program representation, handling of program changes, and non-recursive aggregations first. Then we discuss recursive analyses in Section 3.2.

Program representation and change. We represent the abstract syntax tree (AST) of a program in *base relations* (extensional predicates in Datalog). We use a separate base relation for each AST node type, for example, `IfElseStmt` and `AssignStmt`. We use additional base relations to connect AST nodes, for example, `Next(prev,next)` for subsequent nodes in a list and `Parent(child,parent)` to represent the containment hierarchy. We rely on the IDE to incrementally maintain the base relations for the subject program, either by incremental parsing [Ramalingam and Reps 1993], non-incremental parsing with tree diffing [Dotzler and Philippsen 2016], or projectional editing [Voelter et al. 2014]; we use the latter.

Program analyses and analysis results. Traditionally, a program analysis traverses the AST of a program to extract analysis results. Correspondingly, in IncA, program analyses query the base relations. That is, program analyses are Datalog-style rules that define *derived relations*. For example, in Section 2 we showed rules that define the derived relation `CFlow` based on `IfElseStmt`, `Next`, and others. Derived relations can also make use of the contents of other derived relations, which is useful for decomposing an analysis into multiple smaller parts. For example, we defined `PredecessorIntervals` based on `CFlow`.

IncA allows updates to derived relations to be observed. For example, we can extend our interval analysis to define a derived relation `BoundedArrayAccess` that contains a tuple for each array access where the interval of the index is guaranteed to be within bounds of the array. By observing this relation, an IDE can incrementally highlight safe and (possibly) unsafe array accesses.

Change propagation. IncA ensures that derived relations are up-to-date; as tuples are inserted and deleted from base relations, IncA propagates these changes to dependent derived relations. Effectively, IncA maintains a dependency graph between relations and caches their tuples. The base relations are sources of changes, and the IDE-observed derived relations are sinks of changes.

IncA propagates tuples from sources to sinks, and it skips the propagation where a relation is not affected by an incoming change. This can happen in two cases: (i) The cache of the derived relation already accounts for the tuple, or (ii) a Datalog rule ignores the tuple entirely. The first case occurs when the tuple was already derived before through an alternative derivation, which can happen because there can be alternative bodies for the same Datalog rule head. For example, change ② in Figure 1B was skipped at statement `N2` because the previous analysis result remained valid. The second case for skipping occurs when an incoming tuple simply could not affect the contents of a relation. For example, `boolean` variables are irrelevant for our interval analysis. This manifests in those Datalog rules that inspect the right-hand sides of assignments, but never query relations such as `AndExpr` that represent `boolean` expressions. The IncA compiler optimizes the subject analyses to avoid the propagation of irrelevant changes, see more details in [Szabó et al. 2016].

Finally, the occurrence of negated subgoals in a rule's body leads to a stratification of change propagation. That is, the relation that occurs negated must be fully up-to-date before we can draw conclusions about the absence of a tuple. For example, in our control-flow analysis, we must fully update relation `IfElseStmt` before we can derive changes to `CFlow` based on it. Moreover, when `CFlow` uses ¬`IfElseStmt`, an insertion into `IfElseStmt` triggers deletion(s) from `CFlow`, and a deletion from `IfElseStmt` triggers insertion(s) into `CFlow`. IncA's change propagation respects negation. Stratification requires that there is no negation through recursion because then we could not fully compute the negated relations before computing the non-negated ones.

Non-recursive aggregation. In contrast to other incremental systems [Gupta and Mumick 1995], IncA supports aggregations over *recursive* relations. Here, however, we first review how IncA handles *non*-recursive aggregations, which, while much simpler, is an important stepping stone.

Formally, an aggregating Datalog rule has the following form:

$$\texttt{Agg}(\texttt{t}_1,\ldots,\texttt{t}_k,\alpha(\texttt{v})) \texttt{ :- } \texttt{Coll}(\texttt{t}_1,\ldots,\texttt{t}_k,\texttt{\_x}_1,\ldots,\texttt{\_x}_l,\texttt{v}).$$

Here the *aggregand* column v and *aggregate result* column α(v) are both lattice-valued, and α is an *aggregation operator*, i.e. a mapping from a *multiset* of lattice values to a single lattice value. Without loss of generality (as shown below) we assume this is the only rule with Agg in the head. We call Agg the aggregating relation and Coll the *collecting* relation; $\texttt{t}_1, \ldots, \texttt{t}_k$ are the *grouping* variables, while $\texttt{\_x}_1, \ldots, \texttt{\_x}_l$ are *auxiliary* variables.

For example, in the interval analysis from Section 2, PredecessorIntervals served the role of Coll whereas IntervalBefore served the role of Agg, while α was lub. We used stmt and var as grouping variables (the value of var after statement stmt), and we used pred as an auxiliary variable to enumerate the interval values of all CFG predecessors.

For each substitution of the grouping columns, the aggregand values v in Coll, if any, are mapped by α into the aggregate result column of the single corresponding tuple in Agg. Given a set of grouping values $\texttt{t}_1, \ldots, \texttt{t}_k$, different sets of values $\texttt{\_x}_1, \ldots, \texttt{\_x}_l$ can have the same aggregand v associated with them, so α aggregates over a *multiset* of values (instead of just a simple set).

Note that the above form of the aggregating rule does not restrict expressiveness (**R3**): the IncA language actually allows multiple alternative rules for Agg, each with several subgoals. The compiler then introduces a helper relation Coll that is derived using these multiple rule bodies, and then aggregates Agg from Coll.

We can incrementalize non-recursive aggregations by (i) incrementally maintaining Coll as usual and (ii) incrementally maintaining the aggregate result α(v) of each group whenever an aggregand v is inserted or deleted from the collecting relation. IncA specifically supports (see Section 4.4) the latter kind of incrementality for aggregation operators that are induced by associative and commutative binary operations, e.g. lub (least upper bound) or glb (greatest lower bound).

The above form of aggregating rules is independent of whether the aggregation is recursive or not. An aggregation is recursive if Coll also depends on Agg (e.g. see dependencies between PredecessorIntervals, IntervalAfter, and IntervalBefore in Section 2). This leads us to the next part where we discuss recursive analyses.

3.2 Incremental Execution of Recursive Analyses

Our strategy for incrementalizing *non*-recursive analyses is relatively simple once the analysis is formulated as a relational computation (e.g., through our DSL, Section 5): We propagate changes through the dependency graph, we are careful about negations, and we incrementally maintain aggregate results. Adding recursion, the process is severely complicated because of cyclic reinforcement, as shown in Section 2.

To deal with recursion, we modify our approach in two steps. First, we add *recursive relations* but the aggregations must remain non-recursive. This enables the incrementalization of control flow analysis as well as set-based data-flow analyses and represents the current state of the art of incremental Datalog solvers (including the prior version of IncA as documented in [Szabó et al. 2016]). Second, we add *recursive aggregations* on values of lattices with monotonic lattice operators. This enables the incrementalization of lattice-based data-flow analyses and goes beyond the state of the art. We carefully engineer our solution to handle cyclic reinforcement of lattice values correctly, thus satisfying **Correctness (R1)**.

Recursive relations. In contrast to non-recursive relations, recursive relations require a fixpoint computation. That is, an insertion into a recursive relation can trigger subsequent insertions, which can trigger subsequent insertions, and so on. Consider the following recursive relation that computes the transitive closure of `CFlow`:

```
CFlowReach(from, to) :- CFlow(from, to).
CFlowReach(from, to) :- CFlow(from, step), CFlowReach(step, to).
```

When adding a statement `stmt` to the program, we trigger an insertion of `(pred,stmt)` into `CFlow` for `stmt`'s predecessor. This triggers subsequent insertions such as `(predpred,stmt)` into `CFlowReach` for `pred`'s predecessor, but eventually `CFlowReach` stabilizes when no new tuples can be derived.

Incrementally deleting tuples from a recursive relation is more difficult because of cyclic reinforcement of tuples. If a tuple was derived multiple times within a cycle, deleting one derivation does not necessarily invalidate the other derivations, but invalidating all derivations is sometimes necessary to obtain correct results. This is similar to the cyclic reinforcement of lattice values we discussed in Section 2, with an important difference: The cyclic reinforcement of tuples does not involve aggregations.

There is a generic solution called Delete and Re-derive (DRed) [Gupta et al. 1993] for supporting deletions in spite of cyclic reinforcement of tuples. The basic strategy of DRed is to first delete the tuple and everything that was derived from it, ignoring alternative derivations of that tuple for now. This is an over-approximation and, in general, will delete too much. After the deletion reaches a fixpoint, DRed starts a re-derive phase to insert back all those tuples that can be derived from the remaining ones that were left intact during the delete phase.

Recursive aggregation. The key technical contribution of our paper is to develop a novel algorithm for incremental recursive aggregation in Datalog. As we explained in Section 2, the main challenge is the cyclic reinforcement of lattice values (in contrast to cyclic reinforcement of *tuples*, as discussed above). Specifically, change ③ in Figure 1B induced the deletion of tuple `(N1,y,[-1,0])`, which should have triggered the deletion of `(N2,y,[-1,∞))`. However, because `N2` occurred in a loop, there was cyclic reinforcement between the lattice values: `N5` still reported the interval `[-1,∞)`, even though the very reason for that value is that the initial interval was `[-1,0]` previously. At this point, we did not know how to proceed.

Why can we not just apply DRed here? How is cyclic reinforcement of lattice values different from cyclic reinforcement of tuples? The main difference is that updating an aggregate value induces both a deletion of the old value and an insertion of the new value. However, DRed first processes all deletions and postpones insertions until the re-derive phase. Hence, one issue is that lattice values require an interleaving of the delete and re-derive phases, which violates the contract of DRed. A second issue is more subtle. Let's say we allow interleaving. When deleting the old aggregate value, DRed's delete phase will delete all tuples derived from it. In particular, it will delete the new aggregate value, which we were just about to insert.

To resolve these issues, we developed a novel algorithm called DRed_L. It supports incremental computation of recursive *monotonic* aggregations over lattices. Given a partially ordered set $(\mathcal{M}, \sqsubseteq)$, an aggregation operator α is $\sqsubseteq$-monotonic if $\alpha(\texttt{M}) \sqsubseteq \alpha(\texttt{M} \cup \{\texttt{m}\}) \sqsubseteq \alpha(\texttt{M} \cup \{\texttt{m'}\})$ for all multisets of values $\texttt{M} \subseteq \mathcal{M}$ and all values $\texttt{m},\texttt{m'} \in \mathcal{M}$ with $\texttt{m} \sqsubseteq \texttt{m'}$. That is, the aggregate result of a monotonic aggregation increases with the insertion of any new tuple or with the increasing replacement of any existing tuple in the collecting relation.

Crucially, our algorithm recognizes monotonicity at runtime when handling the update of an aggregate result. Given a $\sqsubseteq$-monotonic aggregator α, whenever DRed_L sees a deletion $(\texttt{t}_1, \ldots,$

393 t_k,old) from a relation and an insertion (t_1, ..., t_k,new) to the same relation with old $\sqsubseteq$ new,
394 it recognizes that together they represent an increasing replacement of a lattice value. We call such
395 a change pair a $\sqsubseteq$-*increasing replacement*, and deletions that are part of a $\sqsubseteq$-increasing replacement
396 do not need to go through a full delete phase. This way, deletions of old aggregate results will not
397 invalidate cyclically dependent lattice values and, in particular, the new aggregate values. This
398 allows DRed_L to perform correct incremental maintenance even in presence of recursive aggregation.
399 We describe the details of DRed_L in Section 4, with a focus on correctness and efficiency; its novelty
400 over the state-of-the-art (esp. DRed) will be discussed in Section 7.
401 While in Section 4 we explain DRed_L in its full generality, its main application is the incremen-
402 talization of program analyses that use custom lattices. Program analyses routinely use lattices
403 to approximate program behavior [Nielson et al. 2010]. Without recursive aggregation, only the
404 power set lattice can be represented because recursive tuple insertion and deletion can model
405 set union and intersection. DRed_L's support for recursive aggregation lifts this limitation and
406 enables the incrementalization of program analyses over any user-defined lattice. In Section 6,
407 we show how DRed_L performs on strong-update points-to [Lhoták and Chung 2011] and string
408 analyses [Costantini et al. 2011].
409
410 ## 4 INCREMENTAL RECURSIVE AGGREGATION WITH DRED_L
411 We propose DRed_L, an incremental algorithm for solving Datalog rules that use recursive aggrega-
412 tion over lattices. Given a set of Datalog rules, DRed_L efficiently and transitively updates derived
413 relations upon changes to base relations. Changes are propagated to derived relations in four steps:
414
415 (1) *Change splitting*: Split incoming changes into monotonic changes (increasing replacements
416 and insertions) and anti-monotonic changes (deletions that are not part of increasing replace-
417 ments). The crucial novelty over the older DRed algorithm is the lattice-aware recognition of
418 increasing replacements at runtime, which heavily impacts the other three steps as well.
419 (2) *Anti-monotonic phase*: Interleaved with the previous step, we process anti-monotonic changes
420 by transitively deleting the relevant tuples and everything derived from them. This is an
421 over-approximation and, in general, will delete too much, but, importantly, over-deletions
422 guarantee that we compute correct results in the face of cyclic reinforcements.
423 (3) *Re-derivation*: Fix the over-approximation of the previous step by re-deriving deleted tuples
424 from the remaining tuples.
425 (4) *Monotonic phase:* Process monotonic changes. For insertions, we insert the new tuple and
426 transitively propagate the effects. For increasing replacements, we simultaneously delete the
427 old tuple and insert the new tuple and transitively propagate their effects. By propagating
428 deletions and insertions of increasing replacements together, we ensure that dependent
429 relations will in turn recognize them as increasing replacements and handle them accordingly.
430 In the remainder of this section, we summarize the assumptions of DRed_L on Datalog rules,
431 introduce necessary data structures for DRed_L, present DRed_L as pseudocode, explain how DRed_L
432 incrementalizes aggregations, and present a proof sketch of our algorithm.
433
434 ### 4.1 Assumptions of DRed_L on the Input Datalog Rules
435 In order to guarantee that DRed_L computes a correct (**R1**) analysis result efficiently (**R2**), the input
436 Datalog rules must meet the following assumptions:
437 **Monotonic recursion (A1):** We call a set of mutually recursive Datalog rules a *dependency*
438 *component*. DRed_L assumes that each dependency component respects a partial order $\sqsubseteq$ of
439 each used lattice (either the natural order of the lattice, or its inverse). Given this order, the
440 relations represented by the rules must only recurs $\sqsubseteq$-monotonically: If they are updated by
441

insertions or $\sqsubseteq$-increasing replacements, then *recursively* derived results may only change by insertions or $\sqsubseteq$-increasing replacements. This assumption has important implications: Abstract interpretation operators and aggregations must be monotonic within a dependency component wrt. the chosen partial order. The analysis developer needs to ensure operators are monotonic. Additionally, negation through recursion is not allowed: IncA automatically rejects analyses that do not conform to this requirement (Section 5).

Aggregation exclusivity (A2) Alternative rules deriving the same (collecting) relation must produce mutually disjoint results, if the relation is in a dependency component that uses aggregation at all. This is important for ruling out cyclic reinforcements.

Cost consistency (A3) Given a non-aggregating relation $\texttt{R}(\texttt{t}_1, \ldots, \texttt{t}_k, \texttt{v})$ with lattice-typed column v, we require that columns $(\texttt{t}_1, \ldots, \texttt{t}_k)$ uniquely determine column v.

No infinite ascending chains (A4) DRed_L computes the least fixpoint of the Datalog rules. To ensure termination and as is standard in program analysis frameworks, DRed_L requires that the used lattices do not contain infinite $\sqsubseteq$ -ascending chains [Atiyah and Macdonald 1994, Section 6] for any $\sqsubseteq$ that is chosen as part of **A1**. Fulfilling this requirement may require widening [Cousot and Cousot 2004], as shown in our interval analysis in Section 2.

Assumption **A1** allows a dependency component to use, and aggregated over, several lattices. For each of those, the component must be monotonic; this is the responsibility of the analysis developer.

In typical lattice analyses, where aggregators are idempotent (e.g. `lub`, `glb`) and all transitive dependency chains involve at least one aggregating relation for any lattice-typed column, the IncA compiler automatically guarantees **A2** and **A3** by transforming IncA analysis code (using the idea in [Ross and Sagiv 1992]). In other cases, they are the responsibility of the analysis developer (though for purely non-aggregating standard Datalog recursion, **A2** trivially holds).

We validated that these assumptions suffice for correctness (**R1**) by constructing a proof sketch (Section 4.5). We also validated that these assumptions do not inhibit expressiveness (**R3**) for incremental program analyses by construction of IncA (Section 5) and case studies (Section 6).

4.2 Support Data Structures

The anti-monotonic step of DRed_L performs an over-deletion, after which some tuples may need to be re-derived; while the monotonic phase performs monotonic deletions that do not always have to be propagated. To make these decisions, we adapt support counts [Gupta et al. 1993] for tuples, and we design a new data structure called support multisets for aggregate results:

- *Support count*: The support count of a tuple is equal to the number of alternative derivations a tuple has within a relation (across all alternative rules and local variables). Given a tuple t in relation R, $\texttt{Support}^{\#}_{R}(\texttt{t})$ represents the support count of t.
- *Support multiset*: The support multiset of an aggregate result contains the individual aggregands that contribute to it. Formally, given an aggregating rule as in Section 3.1, the support multiset $\text{Support}^{MS}_{\text{Agg}}$ associates, to each substitution of grouping variables of aggregating relation Agg, the aggregands in the group from Coll. In other words, $\texttt{Support}^{MS}_{\text{Agg}}(\texttt{t}_1, \ldots, \texttt{t}_k)$ is the multiset of values v that satisfy $\texttt{Coll}(\texttt{t}_1, \ldots, \texttt{t}_k, \_\texttt{x}_1, \ldots, \_\texttt{x}_l, \texttt{v})$ for some $\_\texttt{x}_1, \ldots, \_\texttt{x}_l$.

These support data structures are used and incrementally maintained throughout DRed_L. As $\texttt{Support}^{\#}_{R}$ is no larger than R and $\text{Support}^{MS}_{\text{Agg}}$ is no larger than Coll, there is no asymptotic overhead.

4.3 DRed_L Algorithm

We present the DRed_L algorithm as pseudocode in Figure 2. The entry point is procedure `maintainIncrementally`, which takes a changeset *all* as input and updates affected relations. In the code, we use italic font exclusively for variables that store changesets.

```
1  procedure maintainIncrementally(all) {
2   for C in top. order of dep. components {
3    effect:= immediateConsequences(C, all)
4    (anti, mon):= changeSplitting(effect)
5    deleted:= ∅
6    while (anti != ∅) {//ANTI-MONO - fixpoint
7     new:= updateAnti(anti); deleted ∪= new
8     newEffect:= immediateConsequences(C, new)
9     (anti, mon):=changeSplitting(newEffect ∪ mon)
10   }
11   red:= directRederive(deleted) // RE-DERIVE
12   mon ∪= immediateConsequences(C, red)
13   all ∪= deleted ∪ red
14   while (mon != ∅) { // MONO - fixpoint
15    new:= updateMon(mon); all ∪= new
16    mon:= immediateConsequences(C, new)
17 }}}
```

```
36 procedure directRederive(deleted) {
37  red:= ∅
38  foreach change in deleted  {
39   R:= rel(change)
40   if (R is non-aggregating) {
41    if (Support#R(|change|) > 0)
42     red ∪= { +|change| }
43   } else { // aggregating: R(t̄,α(v)):-...
44    let R(t̄,v) = |change|
45    if (SupportMS_R(t̄) != ∅) {
46     red ∪= { +R(t̄,α(SupportMS_R(t̄))) }
47    }
48   }
49  }
50  update stored relation contents by red
51  return red
52 }
```

```
18 procedure updateAnti(body) {
19  head:= ∅
20  foreach change in body  {
21   R:= rel(change); h:= π_R(change)
22   if (R is non-aggregating) {
23    Support#R(|h|) -= 1
24    if (|h| ∈ R)
25     head ∪= {h}
26   } else { // aggregating: R(t̄,α(v))
27    let R(t̄, v) = |h|
28    SupportMS_R(t̄) -= {v}
29    if (∃w.  (t̄, w) ∈ R)
30     head ∪= {−R(t̄, w)}
31   }
32  }
33  update stored relation contents by head
34  return head
35 }
```

```
53 procedure updateMon(body) {
54  head:= ∅
55  foreach change in body  {
56   R:= rel(change); h := π_R(change)
57   if (R is non-aggregating) {
58    Support#R(|h|) += sign(h)
59    if (support changed to or from 0)
60     head ∪= {h}
61   } else { // aggregating: R(t̄,α(v))
62    let R(t̄, v) = |h|
63    head ∪= {-R(t̄,α(SupportMS_R(t̄)))}
64    if (sign(h) == -1) { SupportMS_R(t̄) -= {v} }
65    else { SupportMS_R(t̄) += {v} }
66    head ∪= {+R(t̄,α(SupportMS_R(t̄)))}
67  }}
68  update stored relation contents by head
69  return head
70 }
```

Fig. 2. The DRed$_L$ algorithm in pseudocode. The entry point is procedure `maintainIncrementally`.

As usual for incremental Datalog solvers, DRed$_L$ iterates over the dependency components of the analysis in a topological order (Line 2). We exploit that recursive changes within each component are required to be monotonic (Assumption **A1**). We start in Line 3 by computing the *immediate consequences* of changes in *all* on the bodies of the Datalog rules in the current component `C`. That is, for a Datalog rule `h :- a`$_1$`,...,a`$_n$ in `C`, we compute the consequences of the changes on `a`$_1$`,...,a`$_n$ first. We omit the details of `immediateConsequences`, but the implementation relies on straightforward *algebraic differencing* from the Datalog literature [Gupta and Mumick 1995]. Technically, the interim result `effect` is expressed on rule bodies and not yet projected to rule heads like `h` (the actual relations to be maintained). This allows us to maintain support counts and support multisets for alternative body derivations.

If component `C` uses aggregation, in Line 4 we perform **change splitting** of changeset *effect*, according to the $\sqsubseteq$ of Assumption **A1**. We compute the set of *monotonic changes mon* from *effect*

by collecting all insertions and those deletions that are part of an increasing replacement:

$$\begin{aligned} mon \;=\; & \{+r(t_1,\ldots,t_k) \mid +r(t_1,\ldots,t_k) \in effect\} \\ & \cup\; \{-r(t_1,\ldots,t_k,c_{old}) \mid -r(t_1,\ldots,t_k,c_{old}) \in effect, +r(t_1,\ldots,t_k,c_{new}) \in effect, c_{old} \sqsubseteq c_{new}\} \end{aligned}$$

Here and below we write $+r(t_1,\ldots,t_k)$ for a tuple insertion and $-r(t_1,\ldots,t_k)$ for a deletion. The set of *anti-monotonic changes* consists of all deletions not in *mon*. Note that for a component `C` that does not use aggregation, *mon* simply consists of all insertions and *anti* consists of all deletions.

Next, we perform the **anti-monotonic phase** on changeset *anti* iteratively until reaching a fixpoint (Line 6-10). In each iteration, we first use procedure `updateAnti` (discussed below) to update the affected support data structures and relations (Line 7). This yields changes *new* to rule heads defined in `C`. We propagate the *new* changes to *deleted* because they are candidates for a later re-derivation. But we also propagate the *new* changes to `C` to handle recursive effects (Line 8), yielding recursive feedback in *newEffect*. By design, the anti-monotonic phase within a dependency component only produces further deletions and never insertions. We merge the *newEffect* changes with the monotonic changes *mon* and split them again because a new change may cancel out an insertion or form an increasing replacement. Note that this can be done efficiently by indexing the changesets for aggregating relations over the grouping variables. This way we can efficiently query relevant lattice values when deciding if a pair needs to be split up or formed.

Procedure `updateAnti` processes anti-monotonic changes of rule bodies and projects them to changes of the rule heads while keeping support counts and support mutlisets up-to-date. We iterate over the anti-monotonic body changes, all of which are deletions by definition. For each change, we obtain the changed relation symbol `R`, and we project with π_R the body change to the corresponding change of the relation's head `h` (Line 21). While `h` is a change, we write `|h|` to obtain the change's absolute value, that is, the tuple being deleted or inserted. If `R` does not aggregate, we decrease the support count of `h` and we propagate a deletion of `h` if it is currently derivable in `R`. If instead `R` aggregates, we decompose `h` to obtain the grouping terms $\bar{t}$ and the aggregand `v`. We delete `v` from the support multiset of $\bar{t}$. Furthermore, if `R` currently associates an aggregation result `w` to $\bar{t}$, we propagate the deletion of said association from `R`. Note that the associated aggregation result `w` is unique by Assumption **A3**, and we delete it as soon as any of the aggregands is deleted. It is important to point out that a positive support count or non-empty support multiset after deletions is no evidence for the tuple being present in the relation, due to the possibility of a to-be-deleted tuple falsely reinforcing itself through cyclic dependencies. We will put back tuples that still have valid alternative derivations, and we will put back aggregate results computable from remaining aggregands in the re-derivation step of DRed_L. Finally, we update the stored relations and return `head` for recursive propagation in the main procedure `maintainIncrementally`.

Back in `maintainIncrementally`, we proceed with **re-derivation** to fix the over-deletion from the anti-monotonic phase (Line 11-12). To this end, we use procedure `directRederive` to re-derive tuples that were deleted during the anti-monotonic phase but still have support. The input to the procedure is `deleted`, and we iterate over the deletions in the changeset. If `R` does not aggregate, we use the support count: A positive support count indicates the tuple is still derivable, and we propagate a re-insertion (Line 41-42). If instead `R` aggregates, we use the support multiset: A non-empty support multiset indicates that some aggregand values are left (Line 45). In this case, we recompute the aggregation result by applying the aggregation operator α and propagating a re-insertion (Line 46). Due to the support multiset, we do not need to re-collect aggregand values, saving precious time. In Section 4.4, we explain how we further incrementalize the aggregation computation (blue highlighting in the code). We return to procedure `maintainIncrementally` by storing the re-derived tuples in `red`. Because these tuples, together with the previously deleted ones, can trigger transitive changes in downstream dependency components, we add `deleted` and

mon to `all` (Line 13). We perform a signed union, so ultimately if a tuple was deleted but then we could re-derive it, then that tuple will not change `all`. Note that re-derivation triggers insertions only and hence only entails monotonic changes that we handle in the final step of DRed_L.

Finally, DRed_L runs the **monotonic phase** until a fixpoint is reached (Line 14-16). In each iteration, we use procedure `updateMon` to compute the effect of the monotonic changes. Procedure `updateMon` is similar to `updateAnti`, but `updateMon` handles deletions as well as insertions due to increasing replacements. If `R` does not aggregate (Line 57-60), we update the support count of `h` according to the tuple being an insertion (`sign(h)=1`) or a deletion (`sign(h)=-1`). If instead `R` aggregates (Line 61-66), we delete the old aggregate result and insert the new one. To this end, we compute the aggregate result over the support multiset before and after the change to the support multiset. We collect all tuple deltas and return them to `maintainIncrementally`, which continues with the next fixpoint iteration. The implementation also checks if the produced tuple deltas may cancel each other out, which is quite common with idempotent aggregation functions such as `lub`.

Procedure `maintainIncrementally` executes the four steps of DRed_L for each dependency component `C` until all of them are up-to-date. While it may seem that a change requires excessive work, in practice many changes have a sparse effect and only trigger relatively little subsequent changes. We evaluate the performance of DRed_L in detail in Section 6. One potential source of inefficiency in DRed_L is computing the aggregation result over the support multiset (highlighted in blue in Figure 2). We eliminate this inefficiency through further incrementalization.

4.4 Incremental Aggregator Function

Procedures `updateAnti` and `updateMon` recompute the aggregate results based on support multisets. A straightforward implementation, that reapplies aggregation operator α on the multiset contents, will require $O(N)$ steps to recompute the aggregate result from a multiset of N values. For example, a flow-insensitive interval analysis on a large subject program may write to the same variable N times. For large N, this can degrade incremental performance as computational effort will linearly depend on input size N (instead of the change size), which contradicts **Efficiency (R2)**.

Given associative and commutative aggregation operators (like `glb` or `lub`), our idea is to incrementalize the aggregator functions themselves using the following approach. Independently from the partial order of the lattice we aggregate over, we take an arbitrary *total* order of the lattice values (e.g., the order of memory addresses). Using this order, we build a balanced search tree (e.g. AVL [Sedgewick and Wayne 2011, Chapter 3.3]) from the aggregands. At each node, we store additionally the aggregate result of all aggregands at or below that node. The final aggregate result is available at the root node. Upon insertion or deletion, we proceed with the usual search tree manipulation. Then we locally recompute the aggregate results of affected nodes and their ancestors in the tree. At each node along the path of length $O(logN)$, the re-computation consist of aggregating in $O(1)$ time the locally stored aggregand with intermediate results. In sum, this way we can incrementally update aggregate results in $O(logN)$ steps, while using $O(N)$ memory.

4.5 DRed_L Semantics and Correctness Proof

Our approach relies on the formal semantics of recursive monotonic aggregation given by Ross and Sagiv [Ross and Sagiv 1992]. We recap here the most important semantical aspects (taken from [Ross and Sagiv 1992]), and then present a (novel) proof sketch for the correctness of DRed_L.

4.5.1 Semantics of Recursive Aggregation. A *database* or *interpretation* assigns actual instance relations to the relation names (*predicates*) appearing in the Datalog rules. A database is *cost consistent* if it satisfies the condition in Assumption **A3**, i.e. lattice columns are functionally determined by non-lattice columns in all relations. Let the set of variables that appear in the body (in any of

the subgoals) of a Datalog rule r be $Vars$. Then, given a concrete database, one can evaluate the body of a rule r to find all substitutions to $Vars$ that satisfy all subgoals in the body.

A *model* is a cost-consistent database that satisfies all Datalog rules, i.e. any given derived relation is perfectly reproduced by collecting or aggregating the derivations of all those rules that have this relation in the head. A *Datalog semantics* assigns a unique model to a set of Datalog rules (where, technically, base relations are also encoded as rules that only use constants, so called *extensional rules*). In the following, we present the semantics of Ross and Sagiv by induction on dependency components, i.e. when considering the relations defined by a given component, we assume that we already know the semantics of all other components it depends on, so they can be equivalently substituted by constants and considered as base relations.

For such a single dependency component, by Assumption **A1** we have a partial order $\sqsubseteq$ for each lattice used. We can thus lift this notation to define a partial order (shown to be a lattice as well) on databases; we say that $D_1 \sqsubseteq D_2$ iff for each tuple $t_1 \in D_1$, there is a tuple $t_2 \in D_2$ in the corresponding relation such that $t_1 \sqsubseteq t_2$. For cost-consistent databases, this is equivalent to saying D_2 can be reached from D_1 by tuple insertions and $\sqsubseteq$-increasing replacements.

It has been shown [Ross and Sagiv 1992] that if a set of Datalog rules satisfy Assumption **A1** and **A3**, then there is a unique *minimal model* M_{min}, i.e. all models M have $M_{min} \sqsubseteq M$. Thus the minimal model is considered as the semantics of the Datalog rules with recursive aggregation.

4.5.2 Correctness of the Algorithm - Proof Sketch. Following [Green et al. 2013; Motik et al. 2015; Ross and Sagiv 1992], we give an informal sketch to prove the proposed DRed_L algorithm correct (**R1**), given the assumptions[3] from Section 4.1. We do not include proofs for the correctness of well-known techniques used, such as algebraic differencing [Gupta and Mumick 1995] or semi-naïve evaluation [Green et al. 2013].

Specifically, we will show that DRed_L terminates and produces the minimal model, while keeping its support data structures consistent, as well. As usual, we will conduct the proof using (i) induction by update history, i.e. we assume that DRed_L correctly computed the results before given input changes were applied, and now it merely has to maintain its output and support data structures in face of the change; as well as (ii) componentwise induction, i.e. we assume that the result for all lower dependency components have already been correctly computed (incrementally maintained), and we consider them base relations.

Preliminaries. Let B^{old} be the base relations (input) before a changeset Δ, inducing minimal model M^{old}_{min}, which DRed_L correctly computed by the induction hypothesis; let $B = B^{old} \cup \Delta$ be the new input and M_{min} the new minimal model, which DRed_L shall compute. Let $D^{\cap}$ be the $\sqsubseteq$-greatest lower bound of the two minimal models (exists since databases themselves form a lattice [Ross and Sagiv 1992]); essentially it is the effect of applying the anti-monotonic changes only.

Let we denote by $D_{init} = D^0_{anti}$ the current database (state of base and derived relations) when DRed_L starts to process the dependency component (with changes in base relations already applied), by D^i_{anti} after iteration i of the anti-monotonic phase, by D^{final}_{anti} at the end of the anti-monotonic phase, by $D_{red} = D^0_{mon}$ after the immediate re-derivations, by D^i_{mon} after iteration i of the monotonic phase, and finally by D^{final}_{mon} after the monotonic phase.

Monotonicity. Iterations of the anti-monotonic phase only delete tuples, so $D_{init} \sqsupseteq D^i_{anti} \sqsupseteq D^{final}_{anti}$. Re-derivations are always insertions, and each re-derived tuple t_{red} compensates for a tuple $t_{anti} \sqsupseteq$

[3] To prove general fixpoint convergence, Ross and Sagiv required each lattice to be a *complete lattice*. We have dropped this assumption in favor of Assumption **A4**, which will also suffice for our goals. Note that we require the finite height of *ascending* chains only, thus strictly speaking our assumption is neither stronger nor weaker than the original one.

t_{red} ($t_{anti} = t_{red}$ for non-aggregating relations) that was deleted during the anti-monotonic phase; therefore we also have $D^{final}_{anti} \sqsubseteq D_{red} \sqsubseteq D_{init}$. Finally, by induction on i and by Assumption **A1**, each iteration i of the monotonic phase performs monotonic changes, thus $D_{red} \sqsubseteq D^{i}_{mon} \sqsubseteq D^{final}_{mon}$.

Termination. The anti-monotonic phase must terminate in finite time as there are finite number of tuples to delete. The immediate re-derivation affects at most as many tuples as the anti-monotonic phase, thus it terminates. The monotonic phase must reach its convergence limit in a finite number of steps as well, for the following reasons. Given a finite database of base relations, a finite amount of non-lattice values are available. Due to Assumption **A3** it follows that the output of the component is also finite. As the monotonic phase never deletes tuples, it can only perform a finite number of insertions to reach this finite output size. Finally, it may only perform a finite number of $\sqsubseteq$-increasing replacements due to Assumption **A4**.

Upper bound and support consistency. We now show that after the anti-monotonic phase, intermediate database states are upper bounded by the desired output, and that support data structures are consistent: A tuple t is (i) contained in the database or (ii) has a positive support count or nonempty support multiset only if $\exists t' \in M_{min}$ with $t \sqsubseteq t'$.

The anti-monotonic phase deleted all tuples that, in any way, depended transitively on the anti-monotonic part of input deletions, therefore those that remained must have had support not impacted by the deletions, thus $D^{final}_{anti} \sqsubseteq D^{\cap} \sqsubseteq M_{min}$. As rules are monotonic by Assumption **A1**, and the support data structures are correctly maintained (using algebraic differencing [Gupta and Mumick 1995] by `immediateConsequences`), any tuple t that has support in state D^{final}_{anti} must have a t' that has support in (and thus contained in) M_{min}. This means that re-derivation only inserts tuples upper bounded by the minimal model, so $D_{red} \sqsubseteq M_{min}$.

A similar argument applies during the monotonic phase. Assume by induction that $D^{i-1}_{mon} \sqsubseteq M_{min}$. All tuples t that are to be inserted in iteration i due to having support (as computed by `immediateConsequences` and reflected in the support data structures by `updateMon`), must have (by Assumption **A1** and the induction hypothesis) a corresponding $t' \in M_{min}$ with $t \sqsubseteq t'$, thus $D^{i}_{mon} \sqsubseteq M_{min}$ still holds. Eventually, $D^{final}_{mon} \sqsubseteq M_{min}$.

Cost consistency. We will show that the output D^{final}_{mon} is cost consistent. Base relations are considered cost consistent due to Assumption **A3** and the assumed correctness of DRed_{L} for lower dependency components. Therefore it is sufficient to check the cost consistency of derived relations, which we do by indirect proof. By the induction hypothesis, the state of the algorithm before the change was the cost-consistent model M^{old}_{min}. The change in base relations did not directly affect derived relations and the anti-monotonic phase only deleted tuples, so D^{final}_{anti} is cost-consistent as well. During the re-derive and monotonic phase, tuples are only inserted if they have support, so cost-consistency is only violated if tuples $t \neq t'$, agreeing on all non-lattice columns, both had support. Let us assume such a situation arises, and we prove that it can actually never happen.

Due to Assumption **A3**, the Datalog rules themselves are specified to obey cost consistency, so t and t' both can't be transitively derivable from base relations at the same time. Thus one of them, say t, must be an error in the support data structure, cyclically reinforcing itself. Such cyclic reinforcement can form for t if it was previously present, then one of its derivations is deleted (which is possible in the monotonic phase via an increasing replacement), but a positive support count still remained, so it was left in the database.

Because of the way how change splitting is performed, increasing replacements in the monotonic phase are only permitted for dependency components that use aggregation. By Assumption **A2**, alternative rules for the same relation must produce disjoint results, so all derivations of t must

stem from the same rule, and may only differ in local variables of the rule. Therefore an increasing replacement that has removed one such derivation must have removed all of them, decreasing the support count of t to zero. This contradiction concludes the indirect proof.

Model. The monotonic phase acts as a standard bottom-up Datalog evaluation using semi-naïve (differential) iterations [Green et al. 2013]. Each iteration i takes the Datalog rules that can be applied to D^{i-1}_{mon} in order to derive new facts or increase aggregate results to obtain D^{i}_{mon}. If a rule can be applied to a database, its result will contribute to the computed immediate consequence due to the correctness of differential evaluation. At the final fixpoint, the immediate consequence is empty; this means that no more rules can be applied to D^{final}_{mon}, therefore the state thus reached (shown above to be cost-consistent) is a model.

Minimality. By definition, $M_{min} \sqsubseteq M$ for any model M over the same base relations. However, we have shown above that $D^{final}_{mon} \sqsubseteq M_{min}$ which implies that $D^{final}_{mon} = M_{min}$, i.e. the algorithm is correct.

5 THE IncA FRAMEWORK

We build a complete analysis framework as an extension of our prior IncA incremental analysis framework [Szabó et al. 2016]. Our extensions span four layers: DSL, compiler, runtime system, and IDE integration. We briefly review each of these extensions. IncA is open source software.[4]

DSL. IncA provides a DSL for the specification of lattices and analyses using these lattices. The DSL is subject language independent, that is, one can use it to define program analyses for any language. For brevity, we highlight the key features of the language here, while Appendix A shows the complete syntax definition. To illustrate IncA, let us re-implement the Datalog rules `IntervalAfter` and `IntervalBefore` from the interval analysis in Section 2 using IncA.

```
def getIntervalAfter(stmt : Stmt, var : Var) : Interval = {
  assert assignsNewValue(stmt, var)
  initializer := getInitializerFor(stmt, var)
  return assignedInterval(stmt, initializer)
} alt {
  assert undef assignsNewValue(stmt, var)
  return getIntervalBefore(stmt, var)
}

def getIntervalBefore(stmt : Stmt, var : Var) : Interval/lub = {
  pred := getCFlowPred(stmt)
  return getIntervalAfter(pred, var)
}
```

In IncA, analysis developers write analysis functions that are similar to Datalog rules in the sense that they also represent relations. However, they are *directional*: They have input and output parameters, and alternative bodies of functions encode alternative ways of obtaining output(s) from input(s), naturally following the style of a forward or backward analysis. These features are inherited from the DSL of the prior version of IncA. Our extension allows the definition of custom lattices and aggregations over lattices. For example, `getIntervalBefore` specifies in it's return type that the returned values are aggregated using `Interval`'s `lub` operator. If an operator (`glb`, `lub`, or others) is specified, IncA aggregates lattice values; otherwise IncA just collects the lattice values. In contrast to Section 3.1, there is no need to define separate functions that compute

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

the `Coll` and `Agg` relations; those are automatically generated by the compiler based on the type annotations. Appendix A shows a more detailed implementation of the interval analysis in IncA including the `Interval` lattice, and Section 6 shows more examples through the implementation of our case studies.

Compiler. IncA translates the DSL code into graph patterns [Rozenberg 1997; Ujhelyi et al. 2015] encoded in Java. Our first extension is to allow relations to carry lattice values. We translate a lattice definition into a Java class, where lattice constructors become class constructors and lattice operations become static methods. We allow relations to use instances of these classes as values. For each aggregation operation `L.n` used in the IncA program under compilation, we generate an AVL tree implementation $\tau_{L.n}$ specialized for `L.n`, as described in Section 4.4.

Our second extension is to introduce a new aggregation graph pattern `Agg(p,c,`$\tau_{L.n}$`)`, where `p` is a graph pattern, `c` is a lattice-typed column of `p`, and $\tau_{L.n}$ is the AVL tree for the aggregation of `c`. We use this new graph pattern for any IncA function `f` that takes or yields an aggregated lattice value. Specifically, we generate two graph patterns: An auxiliary pattern $\mathtt{f}'_p$ that is the result of compiling `f` without any aggregation annotation (analogous to `Coll` of Section 3.1), as well as the main pattern $\mathtt{f}_p$ that aggregates the tuples of $\mathtt{f}'_p$ using our new `Agg` graph pattern (analogous to the relation `Agg`). This translation naturally extends to more than one aggregating columns.

The IncA compiler also performs static analyses on IncA subject analyses to ensure that the assumptions from Section 4.1 are met. These analyses include (1) checking the consistent usage of aggregations, (2) enforcing stratifiable negation, and (3) aggregation exclusivity. These analyses are all implemented in IncA itself and they all rely on analyzing the strongly connected components in the call graph of analysis functions. The results of the analyses are used by the compiler, which reports errors to the analysis developer. Finally, the IncA compiler performs a transformation (written in Java) on the analysis code to ensure cost consistency (Section 4.1). The main idea is to automatically lift up local variables from the bodies of analysis functions to the signatures of analysis functions to ensure that the lattice-typed parameters are uniquely determined by the non-lattice typed parameters.

Runtime system. The low-level incremental solver of IncA is the VIATRA QUERY system (formerly INCQUERY [Ujhelyi et al. 2015]), which is an incremental graph pattern matching engine that did not support incremental recursive aggregation prior to this work. We extended VIATRA QUERY with our new aggregation graph pattern `Agg` and our DRed_L algorithm that incrementally maintains this pattern. Every component of this architecture is available as open source software.[5]

IDE integration. Our primary goal with IncA is to enable the application of program analyses in interactive use cases in an IDE. To this end, we integrated IncA into the MPS language workbench: Our integration comes with typical IDE features, such as syntax highlighting, type system, generators, and validations for analyses expressed with IncA's DSL. MPS relies on a projectional editor [Voelter et al. 2016], where user edits directly lead to AST modifications, without parsing. This aligns perfectly with incremental program analysis because edits directly correspond to fine-grained incremental AST changes.

6 EVALUATION

To validate that our solution satisfies **Efficiency (R2)** and **Expressiveness (R3)**, we used IncA to implement two program analyses with aggregation over lattices that were designed by program analysis researchers, and we ran these analyses on four open source code bases. For an *incremental*

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

analysis framework, performance is crucial, which includes both the run-time (initialization and update) and the memory required for caching. This leads to two evaluation criteria:

Run-time (RQ1) Is an incremental IncA analysis significantly faster than its non-incremental counterpart or other non-incremental analysis frameworks?

Memory (RQ2) Is the extra memory requirement induced by incrementalization acceptable?

Note that we cannot compare to another incremental framework because to the best of our knowledge none exists that supports recursive aggregation over lattices.

6.1 Evaluation Setup

The subject language for our evaluation is Jimple, an intermediate Java representation adopted from the Soot framework [Lam et al. 2011]. It only uses `if`s and `goto`s to express control, which relieves us from reasoning about more complex control flow. It helps with the construction of the CFG, but it does not simplify the data-flow analyses themselves. Implementing control flow analyses with relations is itself an interesting research problem; IncA has been successfully used for C control flow analysis [Szabó et al. 2016], but addressing Java is beyond the scope of this paper.

Our first analysis adapts the strong update points-to analysis for C introduced in [Lhoták and Chung 2011] to Jimple. We chose this analysis because of its practical relevance; points-to analyses are the basis of many other analyses. Our second analysis addresses character inclusion and prefix/suffix analyses for string values [Costantini et al. 2011]; we chose these because they rely on non-trivial lattices and lattice computations. Our string analyses are flow-*in*sensitive in order to show that flow sensitivity is the dominating factor: We hypothesize that despite the more complex lattice, the ballpark of the run time is lower. All of our analyses rely on recursive aggregation over lattices. Our implementations in IncA are available open source.[6]

We analyze the following Java code bases: (1) Google Truth, an assertion framework (9 KLOC Java code), (2) Google Gson, a JSON serialization library (14 KLOC), PGSQL JDBC, a PostgreSQL-Java binding (45 KLOC), and (4) BerkeleyDB, an embedded database (70 KLOC). We selected these because they represent widely used real world applications. We are not aware of a standard benchmark for incremental analyses that comes with a particular code base and sequences of code changes. We imported the code into MPS and transformed it to Jimple, which is functionally equivalent to the original Java programs.

The actual evaluation is as follows. For each code base, we start the analysis with an initial, non-incremental run. We then introduce 1000 incremental program changes on the Jimple code, simulating the edits of a developer. We use two kinds of changes. First, we randomly perform generic changes such as copying and deleting expressions, statements, and methods, or renaming variables. Second, we perform random changes tailored to each analysis: We change assignments for the points-to analysis and modify string values for the string analysis. In all cases, we measure the wall clock time of the initial run and the run-time of the incremental updates to answer **Run-time (RQ1)**. To measure memory consumption (**Memory (RQ2)**), we subtract MPS' total memory use before the initialization of the analyses from the value after the analyses are initialized. For the points-to analysis, we vary the subject code base when running the measurements. For the string analyses, we also vary the number of tracked string properties to find out how much more expensive it is to track not only one string property (e.g. only prefix) but two or three at the same time. We do this in order to see if a developer can simply turn on and off tracked properties to control the amount of information the analysis provides without incurring excessive performance penalty.

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

To position our work relative to the state-of-the-art, we also implemented[7] the points-to analysis with Flix [Madsen et al. 2016], which is a *non*-incremental analysis framework that also supports recursive aggregation over lattices. Unfortunately, we failed to scale our Flix analysis to any of the Java code bases we use even after receiving tips for optimization from the authors of Flix. We set a 5-minute timeout for all measurements, a small multiple of the longest IncA initialization time. There is no other out-of-the-box baseline that we could use for comparison because Flix was used originally to implement the strong-update points-to analysis for C, and we are not aware of another implementation of the string analyses.

To validate **Correctness (R1)**, we compared the results of the incremental analysis with the results of a full re-analysis after every code manipulation, and we verified that they are the same.

We ran the benchmarks on an Intel Core i7 at 2.7 GHz with 16 GB of RAM, running 64-bit OSX 10.12.6, Java 1.8.0_121 and MPS version 2017.3.5. The raw data and the sequence of code changes are available online.[8] We now show interesting implementation details of the analyses, and then present our measurement results to answer our research questions.

6.2 Strong Update Points-to Analysis

The strong update points-to analysis is at the sweet spot between the cheap flow-insensitive and the precise flow-sensitive analysis: It is flow-sensitive for a variable or field (given that we analyze Java/Jimple) as long as the target points-to set consists of a singleton object, and then gives up on flow-sensitivity immediately when the target set has more than one object. This makes sense because dereferencing a non-singleton points-to set does not allow strong updates (that would overwrite the previous results) anyway, and so in most of the cases the precision gained through flow-sensitivity is limited. The original paper [Lhoták and Chung 2011] provides details on the corner cases and precision characteristics. Here, we focus on the implementation in IncA.

Implementation. Figure 3 shows the main building blocks of the analysis. The first one is the flow-insensitive points-to analysis. Figure 3A shows the signatures of the IncA analysis functions. `varPT` returns the points-to set of variable `v`, while `fieldPT` returns the points-to set of object `o` through field `f`. Similar to the interval analysis, these functions collect assignments and interpret the left and right hand sides to collect the pointer variables and the pointed objects. The functions recursively call each other, but they do not use a custom lattice or aggregation.

Next, we define the flow-sensitive analysis that uses the singleton-set lattice to decide when to give up on tracking the points-to set of a variable. Figure 3C shows the lattice definition in IncA, and Figure 3B shows the signatures of the flow-sensitive analysis functions. The implementation of these functions follows the same approach as the interval analysis in Section 2 in that they recursively depend on each other to propagate the points-to information along the subject program's CFG, using a forward style analysis. The current implementation builds on a control flow analysis for Jimple. The functions in Figure 3B all annotate the `SObj` lattice type with `lub`, making sure that the analysis gives up on tracking non-singleton sets (see first `case` in the body of `lub` in Figure 3C).

The final ingredient of the strong update analysis is responsible for combining the results of the previous two (Figure 3D). The analysis returns the result of the flow-sensitive analysis in the first alternative, given that it yields an exact result in the form of a singleton set. The second alternative turns to the flow-insensitive analysis. Additionally, these two functions use a power set lattice for the output parameter (`PSObj`) to aggregate the target objects into one set. The reason for this is that the flow-sensitive analysis gives up on tracking non-singleton targets to save memory, so we must not store separate tuples per target objects here either, otherwise we would lose the memory saving.

[7] https://github.com/szabta89/flix-analyses
[8] https://github.com/szabta89/IncA

A

```
def varPT(v:Var) : Obj
def fieldPT(o:Obj, f:Field) : Obj
```

B

```
def varPTBefore(s:Stmt, v:Var) : SObj/lub
def varPTAfter(s:Stmt, v:Var) : SObj/lub
def fieldPTBefore(s:Stmt, o:Obj, f:Field): SObj/lub
def fieldPTAfter(s:Stmt, o:Obj, f:Field): SObj/lub
```

C

```
lattice SObj {
 constructors { Bot | S(Obj) | Top }
 def leq(l : SObj, r : SObj) : bool =
  return match (l, r) with {
   case (S(o1), S(o2)) => o1 == o2
   case ...
  }
 def lub(l : SObj, r : SObj) : SObj =
  return match (l, r) with {
   case (S(o1), S(o2)) =>
     o1 == o2 ? l : Top
   case ...
} }
```

D

```
def varPT_SU(s:Stmt,v:Var): PSObj/lub = {
 trg := varPTAfter(s, v)
 assert trg != Top
 return PSObj.fromSObj(trg)
} alt {
 assert varPTAfter(s, v) == Top
 trg := varPT(v)
 return PSObj.fromObj(trg)
}

def fieldPT_SU(s:Stmt,o:Obj,f:Field): PSObj/lub = {
 ...
}
```

Fig. 3. Building blocks of the IncA strong update points-to analysis.

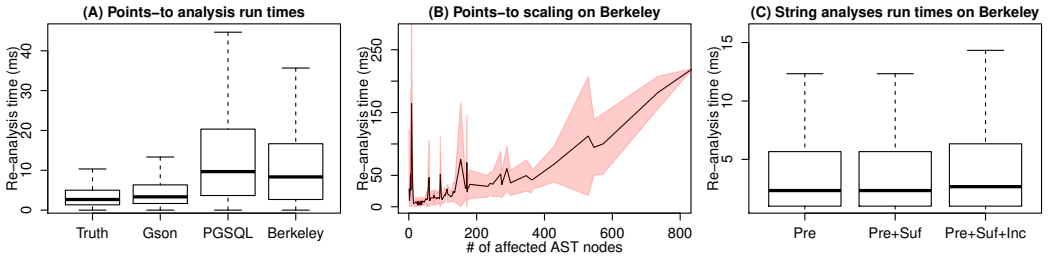

Fig. 4. Runtime measurement results. Box plot whiskers cut off outliers beyond 1.5 IQR.

To query the points-to information, clients would interact with the `varPT_SU` and `fieldPT_SU` functions, whose results are maintained incrementally.

Run-time (RQ1) The *initialization times* of the points-to analysis are as follows: Google Truth - 6.5s, Google Gson - 9.4s, PostgreSQL JDBC - 57.8s, and BerkeleyDB - 64.3s. In contrast to these numbers, loading a larger project in MPS easily takes a few tens of seconds, so even the initialization on BerkeleyDB is not prohibitively long. We compare the numbers to the MPS project loading time because it is better to initialize the analyses during start-up, otherwise developers would observe a longer pause when an analysis is queried first. Figure 4A captures the *incremental update times* of IncA for the four code bases in a box plot (outliers removed). These times are very fast, exactly the kind of numbers we want in interactive applications. Figure 4B reinforces this observation because it shows the scaling of the system on BerkeleyDB in terms of update time wrt. the size of the program change. We had multiple data points per change sizes: The line connects the mean values, and we obtained the red area by subtracting and adding the standard deviation to the mean. We can see also on this graph that the update times are fast, even for larger changes. For example, changes affecting ~800 AST nodes were typically the duplication or removal of complete methods.

Memory (RQ2) The memory consumption of IncA is as follows: Google Truth - 670MB, Google Gson - 1GB, PostgreSQL JDBC - 4.5GB, and BerkeleyDB - 5GB (these numbers essentially remain constant as the program changes). To put these numbers into perspective: (1) At the time of measurements MPS used around 2GB of memory, so the analysis on BerkeleyDB uses 2.5X of that, (2) Flix uses around 1.2GB of memory on a code base that comprises 1 KLOC (which we used

only for testing of our implementation) where IncA uses 100MB, and (3) the points-to analysis is a flow-sensitive analysis, and the Jimple representation of BerkeleyDB has a few hundred thousand CFG nodes. These are high numbers, and they are because of extensive caching, but, in return, we have fast incremental updates.

One of the directions for future work is reducing this memory overhead. We already know concrete steps. Likely a great source of overhead is the strategy that the incremental backend of VIATRA QUERY breaks down a large analysis function into subqueries, each performing an elementary relational algebra operation. For instance, a rule `h :- a1,a2,a3,a4` may be automatically substituted with `h :- b1,b2` and `b1 :- a1,a2` and `b2 :- a3,a4`, so that the rule body is decomposed into a tree of three simple relational joins. The results of these subqueries are each stored, maintained and indexed separately (the Rete [Forgy 1982] approach). This reduces runtime of change propagation (when a_3 changes, b_1 can be looked up from the index instead of recomputed) at the expense of additional memory (both b_1 and b_2 are stored and indexed). The TREAT system [Miranker 1987] is on the other side of this spectrum, it consumes less memory by using more coarse-grained operations, but it loses efficiency in maintenance run-time. Gator networks [Hanson and Hasan 1993] form a continuum between Rete and TREAT by selecting subqueries of various granularities for caching. We will experiment with finding the right trade-off between these approaches and either make it configurable or employ a dynamic solution that tunes the granularity.

6.3 String Analyses

Our second case study concerns string values. Costantini et al. proposed several abstract domains to keep track of the properties of string values [Costantini et al. 2011]. We implemented with IncA three distinct string analyses. First, the character inclusion analysis that keeps track of the definitely-contained and maybe-contained set of characters for a string variable. This, in turn, can be used to decide whether the return value of operations like `indexOf(s)` can be negative or not, which is interesting because a negative value causes operations like `split` or `substring` to throw an exception. We also implemented analyses that compute the longest common prefix and suffix of the strings a variable take; this is useful to check well-formedness, for example, to ensure that an SQL statement always starts with a keyword and terminates with a semicolon.

Implementation. Due to space reasons, we only show a few operations of the prefix lattice:

```
lattice Prefix {
 constructors { Bot | Pre(string) }
 def top() : Prefix = Pre("")
 def leq(l : Prefix, r : Prefix) : boolean =
  return match (l,r) with case (Pre(p1),Pre(p2)) => p1.startsWith(p2)
 def lub(l : Prefix, r : Prefix) : Prefix =
  return match (l, r) with case (Pre(p1), Pre(p2)) => Pre(lcp(p1, p2))
 def interpretSubstring(v:Prefix, s:int, e:int) : Prefix =
  return match v with case Pre(p) => {
    final int l = p.length();
    if (s <= e && e <= l) return Pre(p.substring(s, e));
    else if (s <= e && s < l && l < e) return Pre(p.substring(s, l));
    else return top();
} ... }
```

The `Prefix` lattice [9] has two kinds of elements: `Bot`, the bottom of the lattice, is never used by the analysis, it marks a "failure". The `Pre` element wraps a prefix string. The `leq` operator performs

[9] This lattice is an interesting example, as it has infinite descending chains (any prefix can be continued to an even longer prefix), but no ascending ones (any given prefix has a finite number of truncations), hence Assumption **A4** is still satisfied.

a prefix-check using Java's `String.startsWith` method, while `lub` uses a helper function, `lcp`, that computes the longest common prefix of two strings. The pattern matching syntax allows us to encode the lattice structure in just a few lines of code.

For example, function `interpretSubstring` interprets the result of `substring` calls on strings, in the `Prefix` abstract domain. It performs various checks using the start and end indices and returns a slice of the prefix accordingly, or resorts to the lattice top in case of an empty result.

Run-time (RQ1) We use the notation `Pre` to refer to the string analysis that keeps track of the longest common prefix, `Pre+Suf` tracks the suffix in addition, while `Pre+Suf+Inc` also tracks character inclusion. The initialization times for BerkeleyDB are as follows: `Pre` - 13.5s, `Pre+Suf` - 13.8s, and `Pre+Suf+Inc` - 20.4s. Figure 4 (C) shows the incremental update times. We observe: (1) as expected, a flow-insensitive analysis is cheaper to compute, even incrementally, (2) these run times meet the requirements of interactive applications, (3) activating one (`Suf`) or two more string properties (`Suf+Inc`) has only very minor impact on incremental performance, (4) compared to the non-incremental run, IncA achieves up to four orders of magnitude speedups.

Memory (RQ2) The memory requirement of the analyses are as follows: `Pre` - 1.6GB, `Pre+Suf` - 1.7GB, `Pre+Suf+Inc` - 2.2GB. The overall lower memory use compared to the points-to analysis is because of flow-insensitivity. An interesting observation is that maintaining the longest common suffix in addition to the prefix requires only a 100MB extra memory, but activating the character inclusion induces an extra 500 MB compared to `Pre-Suf`. This is because the character inclusion lattice wraps two sets of characters, which is expensive in terms of memory for the large code base.

Run-time (RQ1) IncA delivers on the run-time performance. It requires an acceptable amount of initialization time even on our largest benchmark code base, and its update times are a few milliseconds on average. In the best case, we achieve a four orders of magnitude speedup with an incremental IncA analysis compared to the non-incremental counterpart.
Memory (RQ2) We pay the price for the fast update times with memory. The memory requirement of IncA can grow large, but it is not prohibitive. In practice, developers would typically work with smaller parts of projects, reducing the absolute amount of memory. In addition, we have concrete plans for reducing the memory use in the future.

7 RELATED WORK

We discuss the relation of this work to individual incremental algorithms, to analysis frameworks, and to incremental Datalog solvers.

Single-analysis algorithms. While IncA is a program analysis *framework*, other researchers have studied how to incrementalize individual analyses. Lu et al. encode points-to analysis as a reachability problem over an auxiliary data structure that encodes the points-to information and selectively re-computes affected paths after a program change [Lu et al. 2013]. Yur et al. incrementalizes a flow- and context-sensitive points-to analysis using worklists [Nielson et al. 2010, Chapter 6.1.1][Yur et al. 1999]. Saha and Ramakrishnan define points-to analysis as a logic program using Datalog [Ceri et al. 1989], and they build on top of the DRed algorithm [Gupta et al. 1993] to incrementalize its execution [Saha and Ramakrishnan 2005]. Their algorithm is lazy, meaning that it only computes points-to information when demanded by clients. Souter and Pollock [Souter and Pollock 2001] and Ismail [Ismail 2009] incrementalize call graph construction with specialized algorithms targeted towards in-IDE use. Instead of incrementalizing individual analyses, we have aimed for a framework that supports the incrementalization of a wide range of analysis.

Analysis frameworks. Many frameworks facilitate the implementation of various classes of analyses. For example, Cooper and Kennedy [Cooper and Kennedy 1984], Pollock and Soffa [Pollock and Soffa 1989], Zadeck [Zadeck 1984], and Eichberg et al. [Eichberg et al. 2007] design algorithms that incrementalize the evaluation of traditional kill-gen data flow analyses (e.g. liveness, very busy expressions, uninitialized variables analysis), which is essentially the same as recursive relational analyses wrt. expressive power; many of the proposed maintenance algorithms coincide with the behavior DRed would have on these analyses. However, this class of analysis does not support aggregation over custom lattices, needed for more sophisticated data-flow analysis. While Khedker [Khedker 1995] provides incremental maintenance for (semi)lattice analyses, only *bitvector-represented* (Boolean) lattices are elaborated, and it is not possible to efficiently represent arbitrary lattices as bitvectors (c.f. the string prefix lattice of Section 6.3). None of these approaches handle flow functions that are monotonic in the lattice but not set-theoretically (bitvector-) monotonic.

Hermenegildo et al. implement an incremental solver for constraint logic programs [Hermenegildo et al. 2000; Puebla and Hermenegildo 1996], but does neither support negation nor aggregation.

Arzt and Bodden's Reviser framework incrementalizes IFDS data-flow analyses [Arzt and Bodden 2014; Reps et al. 1995], which associates semantic information (data flow facts) with nodes in the inter-procedural CFG of the subject program and uses flow functions to propagate the information along the CFG. Upon a change at a CFG node, Reviser computes the transitively reachable affected nodes and re-executes the flow functions on them to compute the updated analysis result. In contrast, IncA provides a more fine-grained incrementalization, and it also exploits monotonic lattice value updates; this would not be possible with Reviser's selective re-computation.

In terms of expressivity, the most closely related framework to IncA is Flix [Madsen et al. 2016]. Flix combines Datalog and lattices and thus, from the frontend's perspective, is similar to IncA. However, Flix is not incremental. In our performance evaluation, we also failed to scale Flix to the subject program sizes that we efficiently support with IncA. In contrast to IncA, Flix uses a verifier to automatically prove the monotonicity and other properties of lattice operations at compile time; in IncA, we only check that a program analysis is free of inconsistent lattice orderings and negation through recursion (Section 5). It should be possible to integrate Flix's verifier in IncA since the expressivity is similar.

Like Flix, Conway et al. also combine lattices and logic programming in Bloom^L [2012]. Bloom^L offers many built-in lattices (e.g. bool, max, map) in its standard library, and developers can also define their own. In contrast to IncA, Bloom^L does not support full incrementality: only insertions are allowed. However, deletions are not only much more difficult to handle (requiring anti-monotonic propagation and re-derivation), deletions are also crucial in an IDE setting where developers freely edit code. Bloom^L targets distributed application development, and Conway et al. argue that supporting insertions is sufficient for this use case.

Incremental Datalog evaluation. Datalog [Ceri et al. 1989] forms the theoretical underpinning for our declarative analyses in mathematical logic. Non-recursive Datalog is straightforward to maintain incrementally (see *Counting* [Gupta et al. 1993]). For stratified recursive Datalog programs, incremental maintenance algorithms (see survey [Gupta and Mumick 1995]) include DRed [Gupta et al. 1993]; improvements over DRed that reduce work in some cases (e.g. *Backward/Forward* [Motik et al. 2015]); and various *provenance*-based approaches [Liu et al. 2009] that maintain, for each tuple, logical expressions as conditions of validity. In contrast to our solution, none of these algorithms can handle recursive aggregation, as it isn't monotonic in the conventional set-theoretic sense.

Our proposed solution DRed$_L$ shares many features with DRed; the following notable differences stem from aggregation support:

- For aggregations, tuple insertions may result in increasing replacements, which DRed cannot correctly handle as they are not conventionally monotonic.
- As its key novelty, DRed_L solves this problem by splitting tuples at runtime according to $\sqsubseteq$-monotonicity as opposed to simply separating deletions and insertions. This has far-reaching consequences on all aspects of the algorithm:
- The monotonic phase of DRed_L must thus incrementally process monotonic deletions (in addition to insertions). This requires a mechanism to determine how to propagate such deletions that were not present in DRed. For this purpose we have adapted support counts (from *Counting*) and introduced support multisets, neither of which were part of DRed where only insertions were considered monotonic.
- The maintenance of these support data structures required DRed_L to enumerate each derivation exactly once; this is achieved (following [Motik et al. 2015]) through semi-naïve evaluation (algebraic differencing).
- As a side effect, such support data structures also enabled the rederive phase of DRed_L to avoid recomputation in contrast to DRed.

An alternative to maintaining recursive aggregations would be to rewrite them [Mazuran et al. 2013; Shkapsky et al. 2015] into a form supported by DRed: replicate tuple t with lattice value c to tuples t' with c' for each $c' \sqsubseteq c$, thus mapping $\sqsubseteq$-monotonic changes, aggregations, and recursions into their classical monotonic counterparts. Depending on the lattice, this transformation may be prohibitively expensive or even impossible. For example, the (integer) interval analysis would require a quadratic amount of additional interval values (one for each sub-interval); while an *infinite* amount of new tuples would have to be produced for an occurrence of `Top` in the points-to analysis. First-class support of lattice values avoids this blowup in DRed_L.

8 CONCLUSIONS

We presented IncA, a framework that efficiently incrementalizes the evaluation of program analyses with user-defined lattices. To this end, we developed the DRed_L algorithm that incrementalizes recursive aggregations over lattice values. We then integrated DRed_L into the runtime system of IncA and developed an analysis DSL to hide the technical details of incrementalization, allowing developers to focus on the essentials of the analysis definition. Our evaluation with practically relevant program analyses on real-world code bases shows that IncA can efficiently support interactive applications in IDEs through fast incremental update time. Our future work is centered around reducing the memory consumption of IncA based on approaches like on-demand evaluation and tuning the granularity of caching and indexing, potentially dynamically as the analysis runs.

REFERENCES

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 (ICSE 2014)*. ACM, New York, NY, USA, 288–298. https://doi.org/10.1145/2568225.2568243

Michael Francis Atiyah and Ian Grant Macdonald. 1994. *Introduction to commutative algebra*. Westview press.

Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 56. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquin. 2011. Incoop: MapReduce for Incremental Computations. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing (SOCC '11)*. ACM, New York, NY, USA, Article 7, 14 pages. https://doi.org/10.1145/2038916.2038923

S. Ceri, G. Gottlob, and L. Tanca. 1989. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Trans. on Knowl. and Data Eng.* 1, 1 (March 1989), 146–166. https://doi.org/10.1109/69.43410

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 (SoCC '12)*. ACM, New York, NY, USA, Article 1, 14 pages. https://doi.org/10.1145/2391229.2391230

Keith D. Cooper and Ken Kennedy. 1984. Efficient Computation of Flow Insensitive Interprocedural Summary Information. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction (SIGPLAN '84)*. ACM, New York, NY, USA, 247–258. https://doi.org/10.1145/502874.502898

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 (ICFEM'11)*. Springer-Verlag, Berlin, Heidelberg, 505–521. http://dl.acm.org/citation.cfm?id=2075089.2075132

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

Georg Dotzler and Michael Philippsen. 2016. Move-optimized Source Code Tree Differencing. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 660–671. https://doi.org/10.1145/2970276.2970315

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 (PADL'07)*. Springer-Verlag, Berlin, Heidelberg, 109–123. https://doi.org/10.1007/978-3-540-69611-7_7

Charles L. Forgy. 1982. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artif. Intell.* 19, 1 (Sept. 1982), 17–37. https://doi.org/10.1016/0004-3702(82)90020-0

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

Ashish Gupta and Inderpal Singh Mumick. 1995. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Eng. Bull.* 18, 2 (1995), 3–18. http://sites.computer.org/debull/95JUN-CD.pdf

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 (SIGMOD '93)*. ACM, New York, NY, USA, 157–166. https://doi.org/10.1145/170035.170066

Eric N. Hanson and Mohammed S. Hasan. 1993. *Gator: An Optimized Discrimination Network for Active Database Rule Condition Testing*. Technical Report TR93-036. Univ. of Florida.

Manuel Hermenegildo, German Puebla, Kim Marriott, and Peter J. Stuckey. 2000. Incremental Analysis of Constraint Logic Programs. *ACM Trans. Program. Lang. Syst.* 22, 2 (March 2000), 187–223. https://doi.org/10.1145/349214.349216

Usman Ismail. 2009. *Incremental call graph construction for the Eclipse IDE*. Technical Report CS-2009-07. University of Waterloo.

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

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.

Ondrej Lhoták and Kwok-Chiang Andrew Chung. 2011. Points-to Analysis with Efficient Strong Updates. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 3–16. https://doi.org/10.1145/1926385.1926389

M. Liu, N. E. Taylor, W. Zhou, Z. G. Ives, and B. T. Loo. 2009. Recursive Computation of Regions and Connectivity in Networks. In *2009 IEEE 25th International Conference on Data Engineering*. 1108–1119. https://doi.org/10.1109/ICDE.2009.36

Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. 2013. An Incremental Points-to Analysis with CFL-Reachability. In *Proceedings of the 22Nd International Conference on Compiler Construction (CC'13)*. Springer-Verlag, Berlin, Heidelberg, 61–81. https://doi.org/10.1007/978-3-642-37051-9_4

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 (PLDI '16)*. ACM, New York, NY, USA, 194–208. https://doi.org/10.1145/2908080.2908096

Mirjana Mazuran, Edoardo Serra, and Carlo Zaniolo. 2013. Extending the power of datalog recursion. 22 (08 2013).

Daniel P. Miranker. 1987. TREAT: A Better Match Algorithm for AI Production Systems; Long Version. (1987).

Ralf Mitschke, Sebastian Erdweg, Mirko Köhler, Mira Mezini, and Guido Salvaneschi. 2014. i3QL: language-integrated live data views. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 417–432. https://doi.org/10.1145/2660193.2660242

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 (AAAI'15)*. AAAI Press, 1560–1568. http://dl.acm.org/citation.cfm?id=2886521.2886537

Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 2010. *Principles of Program Analysis*. Springer Publishing Company, Incorporated.

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

Germán Puebla and Manuel Hermenegildo. 1996. Optimized algorithms for incremental analysis of logic programs. In *Static Analysis: Third International Symposium, SAS '96 Aachen, Germany, September 24–26, 1996 Proceedings*, Radhia Cousot and David A. Schmidt (Eds.). Springer Berlin Heidelberg, 270–284. https://doi.org/10.1007/3-540-61739-6_47

G. Ramalingam and Thomas Reps. 1993. A Categorized Bibliography on Incremental Computation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '93)*. ACM, New York, NY, USA, 502–510. https://doi.org/10.1145/158511.158710

Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. ACM, New York, NY, USA, 49–61. https://doi.org/10.1145/199448.199462

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 (PODS '92)*. ACM, New York, NY, USA, 114–126. https://doi.org/10.1145/137097.137852

Grzegorz Rozenberg (Ed.). 1997. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA.

Diptikalyan Saha and C. R. Ramakrishnan. 2005. Incremental and Demand-driven Points-to Analysis Using Logic Programming. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '05)*. ACM, New York, NY, USA, 117–128. https://doi.org/10.1145/1069774.1069785

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

A. Shkapsky, M. Yang, and C. Zaniolo. 2015. Optimizing recursive queries with monotonic aggregates in DeALS. In *2015 IEEE 31st International Conference on Data Engineering*. 867–878. https://doi.org/10.1109/ICDE.2015.7113340

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

Amie L. Souter and Lori L. Pollock. 2001. Incremental Call Graph Reanalysis for Object-Oriented Software Maintenance. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01) (ICSM '01)*. IEEE Computer Society, Washington, DC, USA, 682–. https://doi.org/10.1109/ICSM.2001.972787

Ozgur Sumer, Umut Acar, Alexander T Ihler, and Ramgopal R Mettu. 2008. Efficient bayesian inference for dynamically changing graphs. In *Advances in Neural Information Processing Systems*. 1441–1448.

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 (ASE 2016)*. ACM, New York, NY, USA, 320–331. https://doi.org/10.1145/2970276.2970298

Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. 2015. EMF-IncQuery: An integrated development environment for live model queries. *Science of Computer Programming* 98, Part 1, 0 (2015), 80 – 99. https://doi.org/10.1016/j.scico.2014.01.004 Fifth issue of Experimental Software and Toolkits (EST): A special issue on Academics Modelling with Eclipse (ACME2012).

Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. 2014. Towards User-Friendly Projectional Editors. In *Software Language Engineering*, Benoît Combemale, DavidJ. Pearce, Olivier Barais, and JurgenJ. Vinju (Eds.). Lecture Notes in Computer Science, Vol. 8706. Springer International Publishing, 41–61. https://doi.org/10.1007/978-3-319-11245-9_3

Markus Voelter, Tamás Szabó, Sascha Lisson, Bernd Kolb, Sebastian Erdweg, and Thorsten Berger. 2016. Efficient Development of Consistent Projectional Editors Using Grammar Cells. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering (SLE 2016)*. ACM, New York, NY, USA, 28–40. https://doi.org/10.1145/2997364.2997365

John Whaley and Monica S. Lam. 2004. Cloning-based Context-sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*. ACM, New York, NY, USA, 131–144. https://doi.org/10.1145/996841.996859

Jyh-shiarn Yur, Barbara G. Ryder, and William A. Landi. 1999. An Incremental Flow- and Context-sensitive Pointer Aliasing Analysis. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*. ACM, New York, NY, USA, 442–451. https://doi.org/10.1145/302405.302676

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

(module)	m	::= **module** n **import** $\overline{n}$ $\{\overline{mc}\}$
(module content)	mc	::= f | l
(lattice)	l	::= **lattice** L{**constructors**$\{\overline{c}\}$ $\overline{lop}\}$
(lattice name)	L	::= name
(type in lat. def.)	T_L	::= AST node type | Java base type | String | L
(constructor)	c	::= $n(\overline{T_L})$
(lattice op)	lop	::= **def** $n(\overline{n : T_L}) : \overline{T_L} = lopb$
(lattice op body)	$lopb$	::= Java code + lattice constructors and operations
(visibility)	vis	::= **private** | **public**
(function)	f	::= vis **def** $n(\overline{n : T_R}) : \overline{T_R} = \overline{alt}$
(type in rel. code)	T_R	::= AST node type | L | L/lop
(alternative)	alt	::= $\overline{s}$
(statement)	s	::= $\overline{n} := e$ | **assert** $cond$ | **return** e
(condition)	$cond$	::= $e == e$ | e != e | e **instanceOf** T_R | e **not instanceOf** T_R | **undef** e
(expression)	e	::= n | lit | $e.n$ | $n(\overline{e})$ | $n^+(e)$ | $L.n(\overline{e})$
(literal)	lit	::= number | string | enum | boolean
(name)	n	::= name

Fig. 5. Syntax of IncA. The highlighted parts show the additions in comparison to the prior version of the IncA DSL [Szabó et al. 2016].

A IncA DSL THROUGH EXAMPLES

In this section, we first give a high level overview over the IncA DSL, and then implement the interval analysis from Section 2 using this DSL.

Syntax. We extend the syntax of our prior IncA DSL with the constructs that are highlighted in Figure 5. A lattice definition l declares the name of the lattice, algebraic data-type constructors for lattice values, and lattice operations. Each lattice operation has a name and is implemented in Java extended with calls to lattice constructors and operations. Each lattice must implement at least `leq`, `lub`, and `glb` operations; other user-defined operations for abstract interpretation of expressions, or statements are optional.

Functions f define IncA analyses. Internally, IncA treats functions as relations, but the distinction of input parameters and output values seems useful for analysis developers [Szabó et al. 2016]. We extended type annotations T_R such that functions can take and produce lattice values. The type L simply means that a function uses lattice values, but does not aggregate them. In contrast, L/lop means lattice-based aggregation using lattice operation lop. In this case, IncA only allows to use operations with the right signature: An aggregation operation must take two lattice values and produce one lattice value. As explained in Section 5, IncA does not verify that the aggregator operator is indeed monotonic. Finally, expressions within a function body can invoke lattice constructors and operations `L.n` using the lattice name as a qualifier.

Interval Analysis in the IncA DSL. The interval analysis in IncA consists of two parts: the interval lattice definition and the main relational analysis. First, Figure 6 shows the implementation of the standard interval lattice in the Java-like language of IncA. Lattice constructors represent the possible values of the interval lattice, and the mandatory operations `leq`, `top`, `bot`, `lub`, and `glb` operate on these values. Additionally, the `subtract` method demonstrates how we can abstractly interpret subtraction in the interval domain.

```
lattice Interval {
 constructors { Bot | Interval(int,int) }

 def leq(l:Interval, r:Interval):boolean =
  return match (l, r) with {
   case (Bot, _) => true
   case (Interval(l1, h1), Interval(l2, h2)) =>
        l2 <= l1 && h1 <= h2
   case _ => false
  }
 def top:Interval = Interval(Int.Min, Int.Max)
 def bot:Interval = Bot
 def lub(l:Interval, r:Interval):Interval = { ... }
 def glb(l:Interval, r:Interval):Interval = { ... }
 def subtract(l:Interval, r:Interval):Interval =
  return match (l, r) with {
   case (Interval(l1, h1), Interval(l2, h2)) =>
    Interval(l1 - h2, h1 - l2)
   case ...
  }
}
```

Fig. 6. Interval lattice definition in IncA.

Figure 7 shows the second part of the interval analysis, using IncA analysis functions, each one of them computing tuples of relations. Conceptually, the analysis follows the idea of the Datalog implementation from Section 2, but there are some important differences:

- Analysis functions are directional; they define a computation from input to output, which naturally follows the design of a forward data flow analysis.
- Aggregation happens through annotations on lattice types. Functions which do not use an annotation on `Interval` simply collect the interval values without aggregating them.
- In contrast to the Datalog implementation in Section 2, we do not use a helper function that would compute a relation similar to `PredecessorIntervals`. Beyond simplifying the explanation, the purpose of that relation was to take care of the cost consistency assumption **A3** (Section 4.1). Even though, the `pred` variable is not used by `IntervalBefore`, we needed to lift it up to the head of `PredecessorIntervals` because this way the columns `(stmt,var,pred)` uniquely determine column `iv` even if the same interval is propagated from multiple CFG predecessors. IncA analysis developers do not need to care about this assumption because the IncA compiler automatically performs the necessary rewritings to satisfy the assumption.

```
def getIntervalAfter(stmt : Stmt, var : Var) : Interval = {
  assert assignsNewValue(stmt, var)
  initializer := getInitializerFor(stmt, var)
  return assignedInterval(stmt, initializer)
} alt {
  assert undef assignsNewValue(stmt, var)
  return getIntervalBefore(stmt, var)
}

def getIntervalBefore(stmt : Stmt, var : Var) : Interval/lub = {
  pred := getCFlowPred(stmt)
  return getIntervalAfter(pred, var)
}

private def assignedInterval(stmt : Stmt, var : Var) : Interval = {
  assert stmt instanceOf Assignment
  left := stmt.left
  assert left instanceOf VarRef
  assert var == left.var
  return ainterpretExp(stmt.right, stmt)
} alt { ... }

private def ainterpretExp(exp : Exp, stmt : Stmt) : Interval = {
  assert exp instanceOf NumberLiteral
  value := exp.value
  return Interval.singleton(value)
} alt {
  assert exp instanceOf VarRef
  return getIntervalBefore(stmt, exp.var)
} alt {
  assert exp instanceOf PlusExpression
  left := ainterpretExp(exp.left, stmt)
  right := ainterpretExp(exp.right, stmt)
  return Interval.binaryOp(left, right, Operation.ADD)
} alt {
  assert exp instanceOf MinusExpression
  left := ainterpretExp(exp.left, stmt)
  right := ainterpretExp(exp.right, stmt)
  return Interval.binaryOp(left, right, Operation.SUB)
} alt {
  assert exp instanceOf PostIncrementExpression
  base := ainterpretExp(exp.expression, stmt)
  return Interval.unaryOp(base, Operation.ADD, 1)
} alt {
  assert exp instanceOf DirectMinusAssignmentExpression
  base := ainterpretExp(exp.left, stmt)
  value := ainterpretExp(exp.right, stmt)
  return Interval.binaryOp(base, value, Operation.SUB)
} alt { ... }
```

Fig. 7. Excerpt of the interval analysis in IncA.