Incrementalizing inter-procedural program analyses with recursive aggregation in Datalog

Tamás Szabó
itemis, Germany /
JGU Mainz, Germany
tamas.szabo@itemis.de

Gábor Bergmann
Budapest University of
Technology and Economics,
Hungary
bergmann@mit.bme.hu

Sebastian Erdweg
JGU Mainz, Germany
erdweg@uni-mainz.de

1. MOTIVATION

Static program analyses play an important role in integrated development environments (IDEs) because they help software engineers spot potential runtime errors (e.g. null pointer dereference, array out of bounds indexing) already at compile time. However, for an analysis to be useful in IDEs, it needs to be both *efficient* and *precise*. Precision is important because false positive results hinder the adoption of static analyses by developers, while efficiency guarantees that analyses can update their results after a program change without incurring a break in the development flow.

In this paper, we study how to use *incrementality* to speed up program analyses without affecting precision. Compared to a from-scratch re-computation, an incremental analysis reuses previous results and updates them based on the changes in the subject program. This pays off at runtime because small program changes typically result in small changes in the analysis result, so an incremental analysis can achieve significant speedups over its non-incremental counterpart. There is a vast amount of research showing how to build increasingly precise and efficient program analyses; ranging from sophisticated inter-procedural points-to analyses that run non-incrementally [1], through incremental type checkers [2] and data-flow analyses [3], to incremental but intra-procedural program analyses with recursive aggregation over custom lattices [4]. In our work, we incrementalize inter-procedural lattice-based program analyses with recursive aggregation. We believe that this is a useful combination when it comes to applications in IDEs: static analyses build on custom lattices, recursive aggregation is important when analyzing cyclic control flow or recursive data structures, while incrementality plays a key role in providing real-time feedback in IDEs.

2. THE IncA ANALYSIS FRAMEWORK

Datalog is a widely used logic programming language to express static analyses [1]. For our work, an important aspect of Datalog is that it provides a good basis for incrementalization. In prior work, we developed the IncA analysis framework [4] (Fig 1), which can efficiently incrementalize recursive Datalog programs. Then, we built IncA_L [5], which supports recursive aggregation over custom lattices. However, our experiments showed that IncA_L does not have the performance required for interactive applications in IDEs when running inter-procedural analyses: We observed that IncA_L is often no better than a from-scratch re-computation in response to a change in the subject program. This behavior is inherent because of the underlying Datalog solver algorithm DRed. The essence of the problem is that DRed invalidates too much of the previous analysis result in response to a deletion in the subject program, and then it needs to do excess work to correct the analysis result. We show this problem with a concrete example in Sec 3.

Framework/Backend	Key feature
IncA [4]/DRed [8]	Recursive analyses
IncA_L [5]/DRed_L [5]	Recursive aggregation over lattices
IncA_D/inspired by DDF [6, 7]	Inter-procedural analyses

Figure 1: Consecutive versions of IncA.

In this paper, we present IncA_D (Fig 1), which is an extension of IncA_L. IncA_D supports lattice-based program analysis with recursive aggregation like IncA_L, but IncA_D also scales to inter-procedural analysis at the same time. The backend of IncA_D builds on differential dataflow (DDF) [6] instead of DRed_L. DDF was introduced independently by McSherry et al. [6] and Motik et al. [7], and we extend DDF to efficiently support recursive aggregation in order to preserve the expressive power of IncA_L. Unlike DRed_L, DDF maintains provenance information about the analysis result. This plays a key role in avoiding excess work in face of deletions in the subject program. We briefly present our extensions to DDF to support recursive aggregation and detail our performance evaluation in Sec 4.

3. RECURSIVE DATALOG: DRED VS DDF

In our example, we use a simple points-to analysis encoded as a Datalog rule in Fig 2 (A). The `PT(v, o)` rule has two alternative bodies: The first one computes all those `(v, o)` tuples that originate from a store instruction, while the second one transitively collects the points-to targets of a variable through the assignments between variables. Transitivity plays a key role in our example because it requires a recursive Datalog program, and this is exactly what makes it challenging to incrementalize the analysis.

We use a graphical representation of the subject program in Fig 2 (B). The graph consists of two kinds of edges: Solid lines represent assignments between variables, and dotted lines represent the assignment of a heap allocated object to a variable. We use the notation `Assign(v, w)` to represent when variable `w` is assigned to variable `v` and `Store(v, o)` when heap object `o` is assigned to variable `v`. Next, we compare the inner workings of two algorithms used to compute the result of recursive Datalog programs.

Delete and Re-derive (DRed) [8] is a well-known algorithm to incrementally maintain the result of recursive Datalog pro-

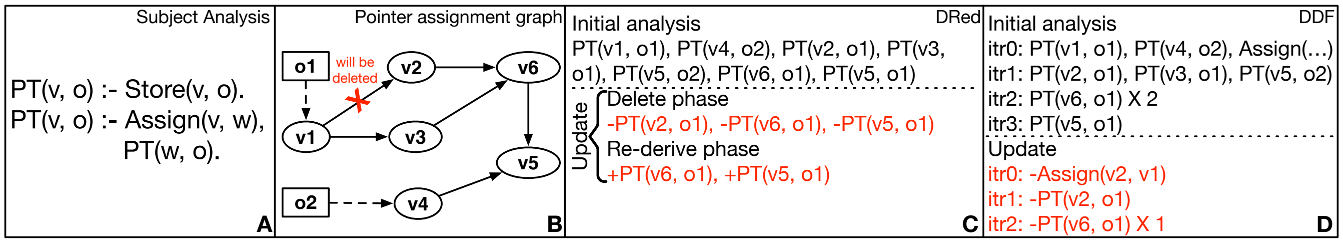

Figure 2: Running example that demonstrates the key differences between DRed and DDF.

grams in face of changing input. The top part of Fig 2 (C) shows the initial analysis result computed by DRed. Assume that we delete `Assign(v2, v1)` from the subject program. DRed computes the analysis result in two fixpoint phases. First, the *delete* phase invalidates everything that (transitively) depends on a deleted input fact. This phase marks tuples `PT(v2, o1)`, `PT(v6, o1)`, and `PT(v5, o1)` for deletion. During this phase, DRed ignores alternative derivations: for example, it does not matter that `v6` is reachable from `o1` also through `v3`, `PT(v6, o1)` is still marked for deletion. For here, suffice it to say that DRed does this because this way it can guarantee correct results in the presence of cyclic dependencies between tuples. DRed fixes the over-deletion in a *re-derive* phase, where it puts back all those tuples that have alternative derivations after the deletion. Note that DRed does excess work for our example because it needs to put back both `PT(v6, o1)` and `PT(v5, o1)` which were victims of over-deletion. Inter-procedural analyses exacerbate this behavior because there are far reaching transitive dependencies when analyzing across procedure calls.

DDF is another algorithm for incrementally maintaining recursive Datalog programs. Compared to DRed, DDF keeps track of two lightweight forms of provenance: *timestamp* and *count*. Timestamp is the iteration number in the fixpoint computation where a tuple was derived. Count is the number of alternative derivations of a tuple within a timestamp. For example, `itr2: PT(v6, o1)`$\times 2$ in Fig 2 (D) means that at timestamp 2, `PT(v6, o1)` has 2 alternative derivations. This extra bookkeeping pays off when handling an input change. DDF computes the consequences of `-Assign(v2, v1)` in three iterations. The fixpoint computation stops at timestamp 2 because DDF realizes that the count of `PT(v6, o1)` only decreases to 1, meaning that the tuple still has an alternative derivation. DDF does not run a re-derive phase because there was no over-deletion that would need to be corrected. This is a promising behavior when aiming to support inter-procedural analyses. However, neither McSherry et al. [6] nor Motik et al. [7] gives details on how to efficiently incrementalize recursive aggregation, which is a must have in our research agenda. Next, we extend DDF to support recursive aggregation.

4. RECURSIVE AGGREGATION IN DDF

We extend our example in Fig 2 to a k-update points-to analysis, which is a lattice-based analysis using recursive aggregation. The analysis keeps track of at most `k` points-to targets per variable, and it gives up tracking them beyond `k` by assigning `Top` to variables. We use the IncA$_{\text{D}}$ variant of Datalog to define the new analysis using the custom `ObjSet` lattice, whose definition we omit for space reasons:

```
PT(v, os) :- Store(v, o), os := ObjSet.singleton(o).
PT(v, lub(os)) :- Assign(v, w), PT(w, os).
```

The analysis uses the `lub` aggregation operator to *join* together `ObjSet` values for the same variable `v`, and it also over-approximates to `Top` if the `ObjSet` grows beyond `k` in size. For the sake of simplicity, we use a degenerate case of this analysis where `k`$=\infty$, but we use a "real" `k` value in our performance evaluation. The result of the new analysis associates one tuple to each variable: Instead of storing `PT(v5, o1)` and `PT(v5, o2)` (as in Fig 2), the new result shall only contain `PT(v5, {o1, o2})`. However, if we carefully follow how the fixpoint computation goes, we realize that the analysis first derives `PT(v5, {o2})` (at timestamp 1) and then, after aggregation, `PT(v5, {o1, o2})` (at timestamp 3). This means that without further coordination, the final analysis result would contain also the intermediate aggregate results. However, analysis clients typically do not want to see such intermediate results. Thus when adopting DDF to recursive aggregation, it is a challenge to prune the results as the fixpoint computation progresses.

To support recursive aggregation in DDF, we built different algorithms with different performance trade-offs. For space reasons, we discuss only one of their main characterizing features, and that is how they do *timeline management*:

- *Faithful* algorithms maintain timestamp intervals instead of punctual timestamps, e.g. `PT(v5, o2)` is present at `[1, 3)`, while `PT(v5, {o1, o2})` is present at `[3,` ∞`)`. Faithful algorithms guarantee that only the result that is present at "infinity" will actually show up on the output, all intermediate results have a finite upper bound for the timestamp interval.
- *First-only* algorithms instead maintain punctual timestamps *of appearance* and keep intermediate results around internally. However, first-only algorithms filter out all but the tuple with the highest timestamp per variable before writing the results to the output. This yields correct results because of the monotonicity of IncA$_{\text{D}}$ analyses: The result with the highest timestamp represents the final result as that was derived latest during the fixpoint computation.

We evaluated our different algorithms on real world code bases from the Qualitas Corpus.[1] For space reasons, we show the numbers for a specific code base and algorithm only. We ran a 5-update inter-procedural points-to analysis with IncA$_{\text{D}}$ using faithful timeline management on the jEdit code base (107KLoC). The first run of the analysis takes 55s and IncA$_{\text{D}}$ uses 4GB of memory. The average time required to update the analysis result through a series of code changes is 37ms, which is exactly the kind of number we look for when using analyses in interactive applications in IDEs.

[1]http://qualitascorpus.com/

References

[1] Yannis Smaragdakis and Martin Bravenboer. Using Datalog for Fast and Easy Program Analysis. In *Proceedings of the First International Conference on Datalog Reloaded*, Datalog'10, pages 245–251, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-24205-2. . URL http://dx.doi.org/10.1007/978-3-642-24206-9_14.

[2] Tamás Szabó, Edlira Kuci, Matthijs Bijman, Mira Mezini, and Sebastian Erdweg. Incremental Overload Resolution in Object-oriented Programming Languages. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, ISSTA '18, pages 27–33, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5939-9. . URL http://doi.acm.org/10.1145/3236454.3236485.

[3] Steven Arzt and Eric Bodden. 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, pages 288–298, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. . URL http://doi.acm.org/10.1145/2568225.2568243.

[4] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 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, pages 320–331, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3845-5. . URL http://doi.acm.org/10.1145/2970276.2970298.

[5] Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. Incrementalizing Lattice-based Program Analyses in Datalog. *Proc. ACM Program. Lang.*, 2 (OOPSLA):139:1–139:29, October 2018. ISSN 2475-1421. . URL http://doi.acm.org/10.1145/3276509.

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

[7] Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks. Maintenance of datalog materialisations revisited. *Artificial Intelligence*, 269:76–136, 2019.

[8] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 157–166, New York, NY, USA, 1993. ACM. ISBN 0-89791-592-5. . URL http://doi.acm.org/10.1145/170035.170066.