Incrementalizing Abstract Interpretation

Position Paper

Noah Van Es, Maarten Vandercammen, Coen De Roover
Software Languages Lab – Vrije Universiteit Brussel – Brussels, Belgium
{noahves, mvdcamme, cderoove}@vub.ac.be

***Abstract*—A powerful approach to design static analysers is abstract interpretation, which reasons over an approximation of the program's behaviour. The *Abstracting Abstract Machines* (AAM) technique, introduced by Van Horn & Might, presents a systematic approach to derive abstract interpreters. However, it is often not useable in the development and evolution of large-scale applications, as with the current state-of-the-art, an AAM analysis can not incrementally update its results upon changes in the program. That is, whenever minor modifications occur in the program's source code, one needs to recompute the entire AAM analysis from scratch, which can easily get time-consuming.**

We therefore propose an incremental approach to abstract interpretation, more precisely the AAM technique. That is, we modify the technique so that the result of an AAM analysis, the *abstract state graph*, can incrementally be updated following a change in the program's source code. Our algorithm tracks dependencies between the nodes in the abstract syntax tree (AST) of the program and the transitions in the abstract state graph to invalidate and recompute new transitions in the state graph upon a change in the AST. Our experiments using a set of Scheme micro-benchmarks reveal that in practice this approach is often limited, as only states that are identical in both state graphs are reusable. We therefore introduce an improvement to the original incremental algorithm, which we refer to as *state adaptation*. State adaptation also enables reusing states that are not identical, but similar. Both the original and improved algorithm are integrated and evaluated in the Scala-AM framework. Our current implementations already show good results in terms of incremental efficiency, although more optimization is required to achieve actual gains in run time performance with our approaches.

***Index Terms*—static analysis, incremental computation, abstract interpretation, AAM**

I. Introduction

Static analysis tools are often an integral part in the evolution of software artefacts to detect bugs early on in the development process. Today's IDEs are equipped with powerful analysis tools that developers have come to rely on to detect bugs in their programs early on. Static analysis is the foundation of these tools, making it possible to check properties of a program without actually executing the program. Static analyses that are used in this context not only require precision and correctness, but also performance. In particular, when developers make incremental changes to the source code, they expect responsive feedback from the program analysis. This results in a need for performant static analysis tools that can efficiently update their results in response to changes in the program.

We explore the incrementalization of the Abstracting Abstract Machines (AAM) technique [1], an approach to sound and decidable program analysis. Currently, when this analysis technique analyses a given program and this program's source code is later edited, the analysis must throw away its original result and recompute the analysis from scratch. Given a series of small modifications to a program, which are typical for an application under development, this often results in time-consuming recomputation of the analysis. Indeed, we expect a minor change to the program to only have a limited impact on the resulting state graph, making recomputation of the entire state graph redundant. Furthermore, analyses developed using the AAM technique are notorious for being slow due to the large state graph that needs to be computed, so efficiency and performance are critical for these kinds of analyses [2].

We therefore hypothesise that an incremental approach for the AAM technique can be developed, where as much as possible of the original analysis result is reused while analysing the updated program.

Concretely, we make the following contributions:

- We design an incremental variant of the AAM approach to abstract interpretation that enables updating the abstract state graph given a change in the AST of the program.
- We present state adaptation as an improvement to this algorithm, so that more reuse can be exploited from the previously computed state graph.
- An implementation of both the original algorithm as well as the improved version with state adaptation is made available in the Scala-AM framework.
- We evaluate our implementation using a series of Scheme micro-benchmarks that enable us to point out strengths and weaknesses of our incrementalization approach.

II. Background

Abstract interpretation [3] is a program analysis technique, used to statically check some property of a given program. The main idea behind abstract interpretation is that the analysis reasons over an approximation of the program's execution behaviour. That is, the program's semantics are abstracted using an over-approximation to obtain an analysis that is sound and decidable. In particular, we focus on the AAM technique.

This technique starts from a *concrete* $CESKt*$ abstract machine, a state machine which acts as a concrete interpreter for a language and hence models the concrete execution of

a program. The AAM technique systematically abstracts this machine to an *abstract* $\widehat{CESKt*}$ abstract machine modelling the abstract execution behaviour of a program. Running the abstracted abstract machine on a given program produces an abstract state graph approximating the execution of the program. More precisely, a given program expression is *injected* into an initial state s_0. A transition relation modelling the abstract semantics of the programming language then defines how we step from one state to another, until the entire state graph is computed. Figure 1 depicts an example of such a resulting abstract state graph.

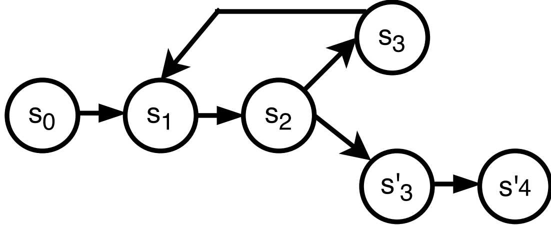

Fig. 1. Illustration of a state graph resulting from abstract interpretation.

III. Incrementalizing AAM

We now present the foundation of our approach to incrementalizing abstract interpretation, more precisely the AAM technique. The goal of this incrementalization is to avoid full recomputation of the analysis given a small change in the source code of the analysed program. Therefore, the resulting algorithm is able to incrementally update the abstract state graph produced by the analysis, given a change to the program's abstract syntax tree (AST). That is, given the previously computed state graph and a list of changes in the AST as input, it produces a new, updated state graph.

Our approach operates on the level of the abstract state graph. It does not rely on any language-specific features or characteristics. Instead, it reuses the state graph computed from the original program (henceforth referred to as the previous state graph) as a cache that is kept up-to-date with modifications in the AST through the invalidation of outdated transitions. When recomputing the new state graph, transitions that are still valid are reused directly, while invalidated transitions have to be recomputed.

To support our incrementalization approach, we need to explicitly track dependencies between the nodes in the AST and transitions in the abstract state graph. These dependencies are registered and maintained in a data structure during the analysis when building up the state graph.

By explicitly tracking these dependencies, we can efficiently update the state graph given a change in the program's source code. Once such a change occurs, we first require an AST differencer that matches nodes between the new and the old AST and points out which of these nodes have been modified. Afterwards, our algorithm works in two phases to update the previously computed state graph. An example of such an initial state graph is given in Figure 2.

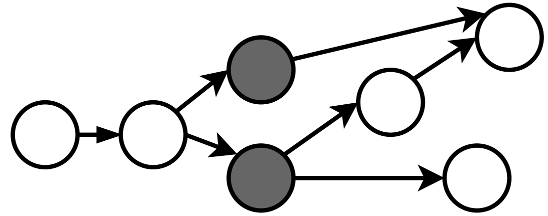

Fig. 2. Illustration of an initially computed state graph.

- In the first phase, transitions that may be affected by the change are *invalidated*, resulting in an invalidated state graph. Finding out which transitions need to be invalidated is done by examining the tracked dependencies. An example of the resulting state graph is given in Figure 3, where states whose transitions were invalidated are highlighted in grey.

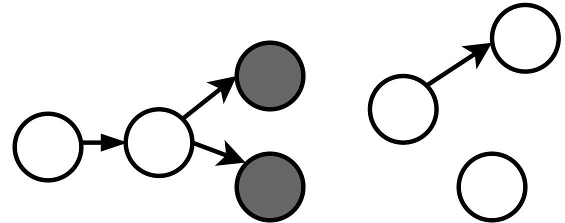

Fig. 3. Illustration of the state graph after invalidation.

- In the second phase, new transitions are *recomputed* from those invalidated states, resulting in the updated state graph. An example is given in Figure 4, where newly computed states and transitions are highlighted in grey.

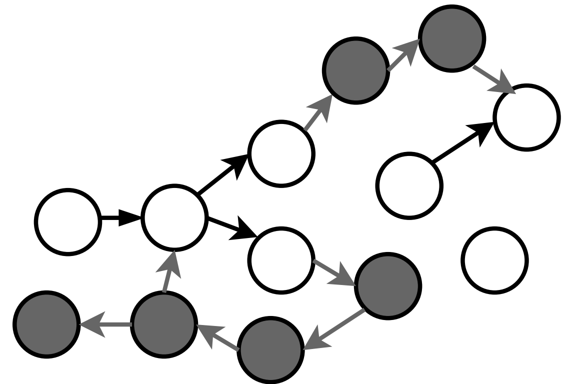

Fig. 4. Illustration of the state graph after recomputation.

Given these two steps, one ends up with a state graph identical to the one obtained from recomputing it from scratch.

IV. State Adaptation

We now present an improvement to the previous incrementalization technique. More precisely, we introduce the concept of state adaptation to overcome a shortcoming of the original approach. A key observation is that when recomputing new

transitions, we often end up with states that are similar, but not entirely identical to corresponding states in the previously computed state graph. The original algorithm is not able to exploit this, as these program states no longer appear identical between the new and the old state graph. Indeed, transitions that are not completely identical will be invalidated and recomputed, and the initial approach's effectiveness becomes limited by the amount of states that are shared identically between both graphs. Using state adaptation, we can now also reuse the transitions of states that are not identical, but *similar* to a state of the new graph.

The main idea is that if the difference, or *delta*, between both states is unimportant to the computation of their successors, the transitions of one can be reused for the other. Doing so avoids recomputation of transitions from scratch because of minor differences in the program state after recomputation. The high-level idea of state adaptation is shown in Figure 5.

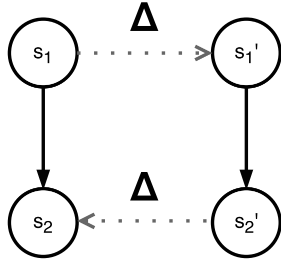

Fig. 5. Illustration of state adaptation.

We refer to state adaptation as an *indirect* form of reuse, as it does not enable direct reuse of states from the previous state graph. Rather, it is more of a hybrid approach, as some work is still required before a transition can be reused. That is, to reuse the transition (s_1', s_2') when determining the transition (s_1, s_2) we require the following steps:

- Given the current state s_1 for which we have to determine the successor state, a similar state s_1' is looked up the previous state graph. If such a state exists, we compute the delta between s_1 and s_1'. In Figure 5, this is indicated by an arrow from s_1 to s_1'.
- Next, we examine the effects and dependencies of the transition (s_1', s_2'). If we can determine that the computed delta does not violate any of these, the transition is ready for indirect reuse using state adaptation.
- This is done by adapting the successor of s_1' (i.e. s_2'), so that it can be reused as the successor of s_1 (i.e. s_2). As indicated by the arrow from s_2' to s_2, this involves applying the computed delta to s_2' to obtain s_2.

The main motivation for state adaptation is to avoid redundant computation of transitions. Instead, an efficient incrementalization strategy should maximise reuse from the previous state graph. State adaptation offers more flexibility in reusing transitions, since it no longer requires states to be completely identical. Of course, the current formulation of state adaptation does not necessarily translate to better run time performance in updating the state graph. While state adaptation can avoid recomputation of transitions, it itself requires searching for similar states, computing a state delta, checking for violations of the delta with the effects of a transition, and then reapplying the delta to some state. The approach presented here provides a solid and safe technique to further improve the incremental AAM algorithm.

V. Implementation

Both the original incrementalization approach, as well as the improved algorithm augmented with state adaptation have been integrated into the Scala-AM framework [4]. All source code has been made available publicly in an online repository[1].

Integrating our novel incremental approach into this existing framework as an additional component offers several benefits. It enables reusing most components of the abstract machine that already exist in the framework. In addition, we avoid producing an isolated artefact by implementing our approach as a component of the framework, so that we can present it as a flexible addition to an established environment.

At the time of writing, our implementation does not yet feature an advanced AST differencer. Instead, it uses a simple differencer that expects the same structure for the new and old AST of the program. While the AST differencing is just a front-end to our algorithm, we envision that future work could integrate a more advanced AST matcher [5] in our implementation to get more accurate differencing on larger, real-world applications.

VI. Evaluation

To evaluate the effectiveness of our incrementalization approach, we study to what extent the previous state graph can be reused. More precisely, when recomputing the updated state graph after a change in the AST, we measure how many of those states are recomputed from scratch compared to how many are reused or adapted from the state graph which was computed in the previous run of the analysis. We expect that lowering the amount of states that need to be recomputed results in an improvement in the total run-time performance for successive runs of the analysis given incremental modifications to a program's source code. Figure 6 shows how many states need to be recomputed for a set of micro benchmarks.

Clearly, the original – or 'naive' algorithm – often falls short, as for most benchmarks the majority of states have to be recomputed. The reason for this is that after recomputation, we often notice minor changes in all subsequent program states, so that only states occurring before the recomputation can be properly reused. A resulting insight here from this observation is that direct reuse – i.e. reuse based on identical states between both graphs – is limited in the context of incremental updates to the state graph of an AAM analysis. This points out the need for an alternative approach to reusability. Indeed, we observe greater incremental efficiency for the 'improved' algorithm that employs state adaptation. Using state adaptation, small

[1] http://github.com/noahvanes/scala-am

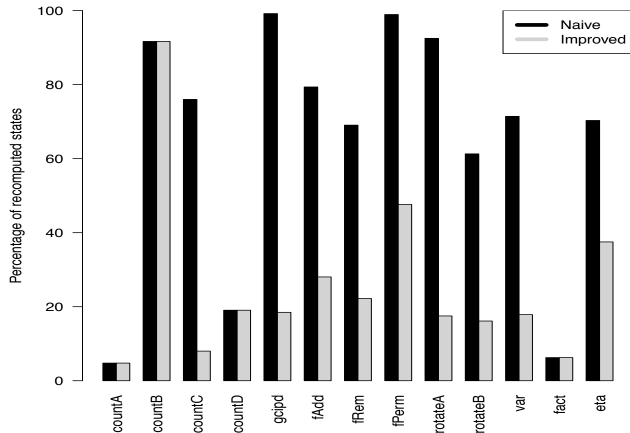

Fig. 6. Recomputation ratios for both algorithms (lower is better).

differences that occur after recomputation can be accommodated for. As a result, using state adaptation fewer states need to be recomputed from scratch.

Currently, the focus of our evaluation is not yet to measure absolute run time performance of the analysis. With our current implementation, our gains in incremental efficiency do not yet result in consistent gains in performance. The main reason for this is that our implementation still copies reused transitions from the previous state graph, and hence still requires traversal of the entire state graph. We aim to solve this issue by restarting recomputation only from invalidated states; however, in order to do so, an improved implementation that takes into account dynamic connectivity issues [6] is required. This is necessary to avoid recomputation in graph components that are disconnected from the initial state after invalidation and therefore not reachable during the execution of the program. Another major issue is that the current formulation of state adaptation is not that efficient. As previously mentioned, performing state adaptation can require several costly steps, which can be more computationally expensive than simply recomputing a transition from scratch. However, we believe that the core idea of state adaptation offers a foundation that enables exploiting more reuse in the context of incremental AAM, and that such a foundation is more amendable to future optimization than from scratch recomputation. It is clear that evaluating the incremental effectiveness enables to assess the potential performance benefits that could be achieved with a more optimized implementation.

Finally, we asserted the correctness of our implementation by comparing the results of our incremental analysis with those of the original Scala-AM framework. For all the experiments we conducted, the output of the incremental analysis was identical to that of the original one.

VII. Related Work

At the time of writing, no other incremental version of AAM has yet been proposed or developed, so the work we presented on our incremental AAM algorithm is entirely novel in that regard. Nevertheless, a large body of research already exists for incremental computation (IC) [7]. Looking at existing work in the incrementalization of programs, in particular that of static analyses, we discern two incrementalization approaches.

On one hand, *manual* approaches, where for a particular kind of analysis an incremental version is designed manually. Examples of such manual incrementalization efforts include incremental static taint analysis in Andromeda [8] and the static analysis of web applications in Gulfstream [9].

On the other hand, *automatic* approaches where an analysis is automatically made incremental by specifying the analysis in some framework. For instance, analyses specified in the DSL of IncA [10] are automatically made incremental, and the Reviser framework [11] enables automatic incrementalization of analyses specified in the IFDS framework. Language support can also offer automatic incrementalization, using some form of self-adjusting computation [12]. Examples include adaptive functional programming [13] and incremental computation using Adapton [14]. In particular, incremental evaluation of tabled Prolog [15] has already been employed to develop incremental versions of existing static analysers [16].

Our own incremental AAM algorithm can be seen as an ad hoc incrementalization approach. Manually incrementalizing AAM allows for explicit, fine-grained control over the incrementalization mechanism which is usually lost in automatic incrementalization approaches.

VIII. Conclusion

We presented a brief overview of our work on designing an incremental approach to abstract interpretation, more precisely the AAM technique. The main idea is that we aim to avoid full recomputation of the abstract state graph, given a change in the source code of our program. Instead, our algorithm is able to efficiently update the previously computed state graph by invalidating affected transitions and recomputing new transitions as required. We observed that this approach can fall short, as states are often no longer identical after recomputation. In general, we concluded that to efficiently update the state graph, we can not only rely on reusing identical transitions from the previous state graph. Therefore, state adaptation was presented as a novel technique to exploit more indirect reuse from the previous state graph. Our experiments reveal that state adaptation can overcome the main weakness of the original algorithm, since it can accommodate for insignificant different in program states after recomputation. As a result, the improved algorithm achieves great incremental efficiency, i.e. it greatly reduces the amount of transitions that have to be recomputed from scratch. However, future work should aim to further optimise our current implementation, so that these gains in incremental efficiency can also be translated into actual performance gains. Nevertheless the results demonstrate that our approach can provide a solid foundation for an efficient and incremental approach to AAM.

REFERENCES

[1] Van Horn, D., & Might, M. (2010, September). Abstracting abstract machines. In ACM Sigplan Notices (Vol. 45, No. 9, pp. 51-62). ACM.

[2] Johnson, J. I., Labich, N., Might, M., & Van Horn, D. (2013). Optimizing abstract abstract machines. ACM SIGPLAN Notices, 48(9), 443-454.

[3] Cousot, P., & Cousot, R. (1977, January). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (pp. 238-252). ACM.

[4] Stiévenart, Q., Vandercammen, M., De Meuter, W., & De Roover, C. (2016, October). Scala-am: A modular static analysis framework. In Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on (pp. 85-90). IEEE.

[5] Falleri, J. R., Morandat, F., Blanc, X., Martinez, M., & Monperrus, M. (2014, September). Fine-grained and accurate source code differencing. In Proceedings of the 29th ACM/IEEE international conference on Automated software engineering (pp. 313-324). ACM.

[6] Thorup, M. (2000, May). Near-optimal fully-dynamic graph connectivity. In Proceedings of the thirty-second annual ACM symposium on Theory of computing (pp. 343-350). ACM.

[7] Ramalingam, G.,& Reps, T. (1993, March). A categorized bibliography on incremental computation. In Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (pp. 502-510). ACM.

[8] Tripp, O., Pistoia, M., Cousot, P., Cousot, R., & Guarnieri, S. (2013, March). Andromeda: Accurate and Scalable Security Analysis of Web Applications. In FASE (Vol. 7793, pp. 210-225).

[9] Guarnieri, S., & Livshits, B. (2010). GULFSTREAM: Staged Static Analysis for Streaming JavaScript Applications. WebApps, 10, 6-6.

[10] Szab, T., Erdweg, S., & Voelter, M. (2016, August). IncA: A DSL for the definition of incremental program analyses. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (pp. 320-331). ACM.

[11] Arzt, S., & Bodden, E. (2014, May). 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 (pp. 288-298). ACM.

[12] Acar, U. A. (2009, January). Self-adjusting computation:(an overview). In Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation (pp. 1-6). ACM.

[13] Acar, U. A., Blelloch, G. E., & Harper, R. (2006). Adaptive functional programming. ACM Transactions on Programming Languages and Systems (TOPLAS), 28(6), 990-1034.

[14] Hammer, M. A., Phang, K. Y., Hicks, M., & Foster, J. S. (2014, June). Adapton: Composable, demand-driven incremental computation. In ACM SIGPLAN Notices (Vol. 49, No. 6, pp. 156-166). ACM.

[15] Saha, D., & Ramakrishnan, C. R. (2006, January). Incremental evaluation of tabled prolog: Beyond pure logic programs. In International Symposium on Practical Aspects of Declarative Languages (pp. 215-229). Springer, Berlin, Heidelberg.

[16] Saha, D., & Ramakrishnan, C. R. (2005, July). 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 (pp. 117-128). ACM.