Context-Sensitive Meta-Constraint Systems for Modular and Explainable Program Analysis*

Kalmer Apinis and Vesal Vojdani

Deptartment of Computer Science, University of Tartu,
Narva mnt 18, EE-51009 Tartu, Estonia
{kalmera,vesal}@cs.ut.ee

Keywords: Program analysis, Data-flow analysis, Constraint systems, Side-effects, Abstract domains, Explainability

1 Introduction

Sound static analyzers can be used to verify computer programs [3, 7]; however, the analyzer itself is a complicated program that should not be trusted blindly. When used in safety-critical settings, manual checking of the analysis results might be neccessary [5]. The review of analysis results is not a simple process, as the user may not understand how the result was computed. Empirical studies suggest that poor explainability of analysis results is an as serious obstacle as false positives in preventing the wider adoption of static analysis tools [2, 6, 8].

Successful analysis of programs written in a real-world programming language needs a sophisticated analyzer. With each additional programming language feature, the analysis tool grows in complexity, requiring more helper analyses that increase the probability of bugs in the analyzer. Novel solutions are needed to make analysis tools easier to implement, use and work with.

In this paper, we embrace the demand for the analyzer to communicate its behavior to the user. As an example, we show how to provide the user with interval expression in addition to the resulting interval values. When the result is unexpected, such analysis expressions provide a valuable middle ground between the complete behavior of the analyzer and the behavior of the program under analysis. The expressions help the end-user by exposing the relevant part of the computation that leads to unexpected values. For error verification, counterexample witnesses [1] may be generated based on the inspection of expression information to minimize the set of paths required to reach an error state.

Providing expression information also helps the analysis developer since inspecting the entire trace of fixpoint computation is still a tedious activity. Generalizing the approach allows analyses to be decomposed into multiple stages where each analysis behavior may be viewed separately. The proposed method fits into the general framework of meta-abstract interpretation described by Cousot et al. [4]. We get either online or offline meta-abstract interpretation – depending on whether the generation of expressions may reference their evaluation.

* This work was supported by the Estonian Research Council grant PSG61.

2 Interval expressions

Given a functional approach [9] analysis where the domain consists of an arbitrary "helper" analysis domain H and the box domain, we extend the domain to include interval expression information:

$$D' = H \times (\mathrm{Var} \to \mathbb{E}^\sharp)_\bot \times (\mathrm{Var} \to \mathrm{I})_\bot$$

For abstract expressions we use values in the form $\mathrm{join}(X) \in \mathbb{E}^\sharp$ where $X \in 2^{\mathrm{E}}$ is a set of expressions defined using the following grammar:

$$\mathrm{E} ::= [N, D', \mathrm{Var}] \mid \mathrm{F}(\mathrm{E}^*) \mid \top$$

The ordering is defined as $\mathrm{join}(X) \sqsubseteq \mathrm{join}(Y) := \top \in Y \vee X \subseteq Y$. The variable $[n, d, x]$ refers to the value of the program variable x in program point n in context d. Furthermore, an expression can be unknown ($\top$) or an n-ary function from the set F together with its argument expressions. It is assumed that F contains interval constants as nullary functions.

This analysis can be implemented directly using the functional approach.

The naive approach has several downsides. First, a buggy analysis may output inconsistent expressions and interval values. Furthermore, a function would be analyzed for each expression and value at the start point, i.e., context, not only for each distinct value. This is excessive as the analyzed program can only access the numeric value – not the way values were computed – and therefore cannot behave differently based on it. Thus, we only store interval values as the context so that the expressions at the start of the function will have literal values.

As a solution, we use three kinds of constraint system variables to reduce unnecessary re-computation. First, helper domain variables $[u, d]_1$ with values from the domain H. Second, expression map variables $[u, d]_2$ with values from the domain $(\mathrm{Var} \to \mathbb{E}^\sharp)_\bot$, and finally, interval map variables $[u, d]_3$ with values from the domain $(\mathrm{Var} \to \mathrm{I})_\bot$.

Interval values for all $v \in N$ are computed from interval expression by evaluation. Thus guaranteeing that they will agree w.r.t. the solution.

$$[v, d]_3 \;\sqsupseteq\; \lambda x.\, [\![[v, d]_2(x)]\!]^\sharp_V (\lambda\, (u, d', y).\, [u, d']_3(y))$$

The constraints for each CFG edge $e = (u, l, v) \in E$ are as follows:

$$([v, d]_1, [v, d]_2) \;\sqsupseteq\; [\![e]\!]^\sharp (([u, d]_1, [u, d]_3), d)$$

Constraints related to function calls can be constructed analogously. Note that the transfer function does *not* get the expression component as a parameter and does *not* contribute directly to the interval component. Also, the current calling context is passed on to the function so that it is able to reference variables for this context in the expression component. The calling context may not be used for any other purpose. Implementation of such a constraint system in a side-effecting framework can be achieved by extending the "helper" analysis.

3 Multi-stage analyses using expressions

Real-world programming languages and real-world programs often use many features in combination: multi-threading, dynamic memory, function pointers, structs, arrays, linked-lists, integers, etc. The meaning of each such feature might be easy to explain in isolation, but together they are complicated. It is not always easy to see why the analysis gives unexpected results if other features are used in conjunction. Keep in mind that the analyzed program might be buggy, the analyzer itself might be buggy, or the analysis precision might be insufficient.

We can reformulate our scheme as a method for separating the analysis into multiple stages where the intermediate results are inspectable. The idea is to limit the expression function symbols F to CFG edge labels and subsequent analysis domain literals. Additionally, we only use a single variable for each program point $\mathrm{Var} = \{\bullet\}$ and only use $\bullet$ as the context in the constraint system. The result of the analysis can be interpreted as a CFG and may be analyzed further at a later stage. This way, the use of some language features, e.g., pointers, may be eliminated from the code. The general goal is to make each stage simple so that it is easier to explain, implement correctly, and hopefully also prove correct.

Since CFG transformations may be too limiting to cover all use cases, we also propose constraint system transformations As opposed to CFG edges, constraints can depend on several variables, and side-effecting constraints can also affect multiple variables. For example, side-effecting constraint systems can be used to analyze multi-threaded programs by handling global variables flow-insensitivity [10].

References

[1] Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. pp. 721–733. ACM, Bergamo Italy (Aug 2015), https://dl.acm.org/doi/10.1145/2786805.2786867

[2] Christakis, M., Bird, C.: What Developers Want and Need from Program Analysis: An Empirical Study. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. pp. 332–343. ASE 2016, ACM, New York, NY, USA (2016)

[3] Cousot, P.: Proving the absence of run-time errors in safety-critical avionics code. In: Proceedings of the 7th ACM & IEEE international conference on Embedded software. pp. 7–9. EMSOFT '07, Association for Computing Machinery, New York, NY, USA (Sep 2007), https://doi.org/10.1145/1289927.1289932

[4] Cousot, P., Giacobazzi, R., Ranzato, F.: A2I: Abstract2 Interpretation. Proceedings of the ACM on Programming Languages 3(POPL), 42:1–42:31 (Jan 2019), https://doi.org/10.1145/3290355

[5] Delmas, D., Souyris, J.: Astrée: From Research to Industry. In: Nielson, H.R., Filé, G. (eds.) Static Analysis. pp. 437–451. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2007)
[6] Johnson, B., Song, Y., Murphy-Hill, E., Bowdidge, R.: Why Don't Software Developers Use Static Analysis Tools to Find Bugs? In: Proceedings of the 2013 International Conference on Software Engineering. pp. 672–681. ICSE '13, IEEE Press, Piscataway, NJ, USA (2013)
[7] Møller, A., Schwartzbach, M.I.: Static program analysis (October 2018), department of Computer Science, Aarhus University
[8] Nguyen Quang Do, L., Bodden, E.: Explaining static analysis with rule graphs. IEEE Transactions on Software Engineering (Jan 2020)
[9] Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Muchnick, S., Jones, N. (eds.) Program Flow Analysis: Theory and Application. pp. 189–233. Prentice-Hall (1981)
[10] Vojdani, V., Apinis, K., Rõtov, V., Seidl, H., Vene, V., Vogler, R.: Static race detection for device drivers: the goblint approach. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016. pp. 391–402. ACM (2016)