Modular Heap Shape Analysis for Java Programs*

Florian Frohn and Jürgen Giesl

LuFG Informatik 2, RWTH Aachen University, Germany
{florian.frohn,giesl}@informatik.rwth-aachen.de

Abstract

We report on our ongoing work towards an automatic heap shape analysis for Java programs in order to approximate the possible forms of sharing and cycles on the heap. Our analysis is completely modular, i.e., Java methods are analyzed independently and each method is analyzed only once. Moreover, in contrast to similar analyses, our technique does not only analyze cyclicity and reachability, but also sharing, i.e., it checks whether two objects may have a common successor. Finally, our analysis is path-sensitive, i.e., if two objects o and o' share, then our analysis approximates which paths of object fields lead from o and o' to their common successor. Use-cases for such an analysis include (but are not limited to) the modularization of termination analysis, complexity analysis, and data-flow analysis.

1 Introduction

Our tool AProVE [4] has been the most powerful termination analyzer for Java programs for many years, as witnessed by the annual termination competition.[1] Its outstanding feature is its precise handling of the heap, which is analyzed via symbolic execution using a rather complex abstract domain [5, 1]. Recently, AProVE's termination analysis has been adapted for complexity analysis of Java programs [3]. This adaption was driven by our project *CAGE* [2] with Draper Inc. and the University of Innsbruck where AProVE analyzes programs with tens of thousands lines of code. Unsurprisingly, this project revealed bottlenecks w.r.t. the scalability of AProVE's current heap shape analysis (HSA). While we can bypass these restrictions by providing *method summaries* as additional input interactively [3], a more scalable automatic HSA is desirable.

In Sect. 2, we sketch such an analysis after discussing AProVE's current heap shape analysis (which we call $\mathcal{HS}_{\mathsf{AProVE}}$ from now on) and its limitations w.r.t. scalability. Like $\mathcal{HS}_{\mathsf{AProVE}}$, our new analysis $\mathcal{HS}_{reg}$ can be used to analyze sharing as well as cyclicity, but in this paper we focus on sharing for reasons of space. Sect. 3 briefly discusses related work and concludes.

2 An Alternative to AProVE's Heap Shape Analysis

Like many other HSAs, AProVE represents sets of program states using a symbolic heap and a mapping from local variables to symbolic references (i.e., pointers to memory locations). To express information about the heap, it uses predicates over symbolic references [5]:[2] The *points-to* predicate $o \xrightarrow{f} o'$ expresses that the field f of the object referenced by o stores the reference o', the *may-alias* predicate $o =^? o'$ means that o and o' may be equal, and the *may-share* predicate $o \searrow\!\!\swarrow o'$ states that o and o' may have a common successor. The latter also covers reachability, i.e., there could be a path from o to o' or vice versa. $\mathcal{HS}_{\mathsf{AProVE}}$ assumes that all objects are in disjoint parts of the heap unless predicates allow sharing or aliasing. Note that the points-to

*Supported by the DFG grant GI 274/6-1 and the Air Force Research Laboratory (AFRL).

[1] http://termination-portal.org/wiki/Termination_Competition

[2] Here, we omit some predicates for reasons of space.

predicate means that some connection *must* exist, whereas the may-alias and the may-share predicate express that references *may* share or alias, i.e., $\mathcal{HS}_{\mathsf{AProVE}}$ simultaneously performs may- and must-analyses. Consequently, the semantics of $\mathcal{HS}_{\mathsf{AProVE}}$ is rather complicated.

Example 1. *Consider a class for lists with the fields* `value` *and* `next`*. To represent a list* o_1 *with at least two elements whose last value is* o_3*, the predicates* $\{o_1 \xrightarrow{\texttt{next}} o_2, o_2 \searrow\!\!\swarrow o_3\}$ *can be used. Note that the connection from* o_2 *to* o_3 *of the form* `next`*`.value` *cannot be expressed precisely since the may-share predicate is not path-sensitive. In particular,* $o_2 \searrow\!\!\swarrow o_3$ *also allows paths from* o_3 *to* o_2*, i.e., it over-approximates the connection between* o_2 *and* o_3 *rather coarsely.*

The main drawback of $\mathcal{HS}_{\mathsf{AProVE}}$ for scalability is its dependence on the context in which methods are called (context-sensitivity). So the same method often has to be analyzed several times to take differences in the calling states into account. This need for context-sensitivity is closely related to $\mathcal{HS}_{\mathsf{AProVE}}$'s restricted expressivity. As shown in Ex. 1, $\mathcal{HS}_{\mathsf{AProVE}}$ cannot describe the effect of many common heap manipulating methods like appending to the end of a list precisely. Context-sensitivity often allows us to focus on specific cases instead. For example, the result of appending to the end of a list *of known length* can be expressed using the points-to predicate.

Hence, the goal of our new analysis $\mathcal{HS}_{reg}$ is to improve expressivity such that context-sensitivity can be sacrificed without major regressions w.r.t. precision. To this end, $\mathcal{HS}_{reg}$ uses a single path-sensitive predicate $o \xrightarrow{\pi}\!\xleftarrow{\tau} o'$. Here, π and τ are regular languages over the set $\mathcal{F}$ of all fields in the program (i.e., $\pi, \tau \subseteq \mathcal{F}^*$). The semantics of $o \xrightarrow{\pi}\!\xleftarrow{\tau} o'$ is that o and o' may only share if o reaches their common successor via a path from π and o' reaches this successor via a path from τ. (More precisely, if $o.v = o'.w$ and $o.v' \neq o'.w'$ holds for every prefix v' of v and every prefix w' of w where $v' \neq v$ or $w' \neq w$, then we have $v \in \pi$ and $w \in \tau$). Hence, $o \xrightarrow{\mathcal{F}^*}\!\xleftarrow{\mathcal{F}^*} o'$ and $o \xrightarrow{\varepsilon}\!\xleftarrow{\varepsilon} o'$ correspond to $\mathcal{HS}_{\mathsf{AProVE}}$'s may-share and may-alias predicates.

Since $\mathcal{HS}_{reg}$ does not provide a points-to predicate, it is a pure may-analysis and hence avoids the complexity that arises from combining may- and must-analyses.

Example 2. *With* $\mathcal{HS}_{reg}$*, the situation in Ex. 1 can be expressed by the predicates* $\{o_1 \xrightarrow{\texttt{next}}\!\xleftarrow{\varepsilon} o_2,$ $o_2 \xrightarrow{\texttt{next}^*\texttt{.value}}\!\xleftarrow{\varepsilon} o_3\}$*. The path from* o_2 *to* o_3 *is described more precisely than in Ex. 1, since these predicates do not allow paths from* o_3 *to* o_2 *anymore. However, they only express that* o_1*'s field* `next` may *store* o_2*, whereas the predicates from Ex. 1 express that it* must *store* o_2*.*

Ex. 2 shows that $\mathcal{HS}_{\mathsf{AProVE}}$ and $\mathcal{HS}_{reg}$ are orthogonal in general, but $\mathcal{HS}_{reg}$ can describe possible (as opposed to definite) sharing more precisely than $\mathcal{HS}_{\mathsf{AProVE}}$, i.e., $\mathcal{HS}_{reg}$ fits our needs w.r.t. expressivity. However, applying $\mathcal{HS}_{reg}$ in an interprocedural, context-insensitive setting with reasonable precision is non-trivial. The reason is that interprocedural program analyses usually summarize methods using pre- and postconditions. Hence, to ensure that every method is analyzed at most once, each method has to be analyzed with the most general precondition. For $\mathcal{HS}_{reg}$, this means that we have to add the predicate $o \xrightarrow{\mathcal{F}^*}\!\xleftarrow{\mathcal{F}^*} o'$ for each pair o, o' of arguments of the analyzed method to the initial state. Clearly, this would diminish the precision of $\mathcal{HS}_{reg}$.

Our solution is to analyze a slightly different property than $\mathcal{HS}_{\mathsf{AProVE}}$ and many other similar analyses. $\mathcal{HS}_{\mathsf{AProVE}}$ analyzes the property "which references may share" for every program position of the analyzed method `m`. Clearly, this property is highly context-sensitive, i.e., it can differ significantly depending on the program state in which `m` is invoked. Instead, $\mathcal{HS}_{reg}$ analyzes the property "which references may share *due to side-effects of* `m`", i.e., it just considers sharing that has been introduced by the currently analyzed method itself. While this property is clearly context-insensitive, it coincides with the property analyzed by $\mathcal{HS}_{\mathsf{AProVE}}$ if there is no sharing in the initial state, as it is the case for the `main` method of Java programs.

Example 3. *Consider a method* `add` *that appends a value o' to the end of a list o. Independently from the states in which* `add` *is called, the property "which references may share when* `add` *returns* due to side-effects of `add`*" can be approximated by the predicate* $o \xrightarrow{\mathtt{next}^*.\mathtt{value}}\xleftarrow{\varepsilon} o'$.

As a consequence, whenever $\mathcal{HS}_{reg}$ encounters an invocation of a previously analyzed method `m`, it has to incorporate the connections that might be introduced by `m` into the calling state.

Example 3 (continued)**.** *Assume that* `add` *is called in a state whose heap is described by* $o \xrightarrow{\mathtt{next}^*}\xleftarrow{\varepsilon} o''$. *To construct the state* after *the invocation of* `add`*,* $\mathcal{HS}_{reg}$ *has to incorporate the new connection* $o \xrightarrow{\mathtt{next}^*.\mathtt{value}}\xleftarrow{\varepsilon} o'$ *into the predicate* $o \xrightarrow{\mathtt{next}^*}\xleftarrow{\varepsilon} o''$ *of the calling state. Since o'' may be reachable from o via* `next`*-pointers, o' may be inserted behind o'', i.e., o' and o'' may share. Hence, the resulting state is* $\{o \xrightarrow{\mathtt{next}^*}\xleftarrow{\varepsilon} o'', o \xrightarrow{\mathtt{next}^*.\mathtt{value}}\xleftarrow{\varepsilon} o', o'' \xrightarrow{\mathtt{next}^*.\mathtt{value}}\xleftarrow{\varepsilon} o'\}$.

To infer predicates that approximate the side-effects introduced by methods (with loops or recursion), $\mathcal{HS}_{reg}$ requires techniques for generalization and symbolic reasoning with regular languages. We developed a library which accomplishes this task and yields promising results.

3 Conclusion

In this paper, we discussed some shortcomings of AProVE's current heap shape analysis and sketched an alternative analysis to overcome them. The key idea is to use a path-sensitive abstract domain to improve expressivity, which in turn allows us to sacrifice context-sensitivity without major regressions w.r.t. precision. As a result, we obtain a high degree of modularity. First results with a prototypical implementation are promising.

Apart from AProVE's previous approach to HSA, the most closely related techniques are [7, 6]. However, these approaches are just field-sensitive, i.e., in contrast to our technique they do not take the order of fields on paths into account. Moreover, they just analyze reachability and cyclicity, and they rely on a field-insensitive sharing analysis, whereas our sharing analysis is also field- (and even path-) sensitive.

References

[1] M. Brockschmidt, R. Musiol, C. Otto, and J. Giesl. Automated termination proofs for Java programs with cyclic data. In *Proc. CAV '12*, LNCS 7358, pages 105–122, 2012.

[2] Complexity Analysis-Based Guaranteed Execution. `www.draper.com/news/draper-s-cage-could-spot-code-vulnerable-denial-service-attacks`.

[3] F. Frohn and J. Giesl. Complexity Analysis for Java with AProVE. In *Proc. iFM '17*, LNCS. To appear.

[4] J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, J. Hensel, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Analyzing program termination and complexity automatically with AProVE. *JAR*, 58(1):3–31, 2017.

[5] C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Automated termination analysis of Java Bytecode by term rewriting. In *Proc. RTA '10*, LIPIcs 6, pages 259–276, 2010.

[6] E. Scapin and F. Spoto. Field-sensitive unreachability and non-cyclicity analysis. *Sci. Comput. Program.*, 95:359–375, 2014.

[7] D. Zanardini and S. Genaim. Inference of field-sensitive reachability and cyclicity. *ACM Transactions on Computational Logic*, 15(4):33:1–33:41, 2014.