Science of Computer Programming 92 (2014) 2–24

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Conditional termination of loops over heap-allocated data ☆

Elvira Albert [a], Puri Arenas [a], Samir Genaim [a], Germán Puebla [b], Guillermo Román-Díez [b,*]

[a] DSIC, Complutense University of Madrid (UCM), Spain
[b] DLSIIS, Technical University of Madrid (UPM), Spain

HIGHLIGHTS

- Termination of programs that depends on heap-allocated data.
- A reference constancy analysis to infer constant access paths heap data.
- Notion of locality partition to guarantee the locality of heap-allocated data.
- Inferring the aliasing preconditions to guarantee termination.
- Implementation and experimental evaluation of our approach in the COSTA system.

ARTICLE INFO

Article history:
Received 1 August 2012
Received in revised form 21 April 2013
Accepted 22 April 2013
Available online 4 May 2013

Keywords:
Static analysis
Heap-sensitive analysis
Termination
Java bytecode
Program transformation

ABSTRACT

Static analysis which takes into account the values of data stored in the heap is considered complex and computationally intractable in practice. Thus, most static analyzers do not keep track of object fields nor of array contents, i.e., they are *heap-insensitive*. In this article, we propose *locality conditions* for soundly tracking heap-allocated data in Java (bytecode) programs, by means of *ghost* non-heap allocated variables. This way, heap-insensitive analysis over the transformed program can infer information on the original heap-allocated data without sacrificing efficiency. If the locality conditions cannot be proven unconditionally, we seek to generate *aliasing preconditions* which, when they hold in the initial state, guarantee the termination of the program. Experimental results show that we greatly improve the accuracy w.r.t. a heap-insensitive analysis while the overhead introduced is reasonable.

1. Introduction

It is well known that shared mutable data structures, such as those stored in the heap, are the bane of formal reasoning and static analysis (see e.g. [23,11]). This problem is exacerbated in object-oriented programs, since most data reside in objects and arrays stored in the heap. Analyses which keep track (resp. do not keep track) of heap-allocated data are referred to as *heap-sensitive* (resp. *heap-insensitive*). In most cases, neither of the two extremes of using a fully heap-insensitive analysis or a fully heap-sensitive analysis is acceptable. The former produces too imprecise results and the latter is often computationally intractable. There has been significant interest in developing techniques that result in a good balance between the accuracy of analysis and its associated computational cost. A number of heuristics exist which differ in how the

☆ Preliminary versions of some parts of this work were presented at FM'09 (Albert et al., 2009) [4], SAS'10 (Albert et al., 2010) [5] and Bytecode'12 (Albert et al., 2012) [10].

* Corresponding author. Tel.: +34 659615728.
E-mail addresses: elvira@sip.ucm.es (E. Albert), puri@sip.ucm.es (P. Arenas), samir@fdi.ucm.es (S. Genaim), german@fi.upm.es (G. Puebla), guillermo.roman.fi@gmail.com, groman@fi.upm.es (G. Román-Díez).

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2013.04.006

values of heap-allocated data are modeled. A well-known heuristic is *field-based* analysis, in which only one variable is used to model all instances of a field, regardless of the number of objects for the same class which may exist in the heap. This approach is efficient, but loses precision quickly.

The approach we propose in this article is based on the observation that, by analyzing *program fragments* (or *scopes*), rather than the application as a whole, it is often possible to keep track of the values of heap-allocated data in a similar way as for non-heap allocated variables. Such fragments can be built starting from methods, loops, or even blocks of contiguous sentences. Our final goal is to be able to instrument programs such that accesses to heap-allocated data are *replaced* with (or replicated by) equivalent accesses to, non-heap allocated, *ghost* variables whose values represent the values of the corresponding heap-allocated data. The instrumented program can then be input to any heap-insensitive static analysis, which can now obtain heap-sensitive information, since the ghost variables expose the heap-allocated values.

The kind of properties that can benefit from our approach are those which require a *local* or *compositional* reasoning, i.e., they require the inference of the property for certain fragments, rather than a global inference for the whole program execution. Termination, the target application of our article, is a property that requires such kind of local reasoning, where the scopes of interest are the loops. Basically, in order to prove termination, the analysis has to keep track of how the size of the data involved in loop guards changes when the loop goes through its iterations. This information is used for determining (the existence of) a *ranking function* [25] for the loop, which is a function which strictly decreases on a well-founded domain at each iteration of the loop and which ensures termination of the corresponding loop.

Obviously, not all heap-allocated data are *transformable*, i.e., their behavior reproducible using ghost variables. In the most general characterization, the replacement is possible when two sufficient conditions hold within the scope: (a) the memory location where the heap-allocated data is stored does not change, i.e., the reference to such data remains *constant*, and (b) all accesses (if any) to such memory location are done through the same reference (and not through aliases). This characterization captures the situations in which heap-allocated data behave *locally* (i.e., like non-heap allocated variables) in the given scopes.

1.1. Summary of contributions

The overall contribution is a practical method to perform heap-sensitive termination analysis by (a) first performing a pre-analysis which allows us to determine when heap-allocated data behave locally (i.e., their behavior is reproducible using *ghost* variables) and, otherwise, infer the conditions under which locality holds and, (b) then instrumenting the program with ghost variables whose contents represent the contents of the corresponding heap-allocated data. Heap-insensitive termination analysis on the transformed program allows us to reason on data allocated in the heap through the ghost variables. Technically, our main contributions can be summarized as follows:

1. We first develop a semantic-based *reference constancy* analysis (instead of just performing syntactic checks) which infers the (constant) access paths to both fields and array elements in a uniform way.
2. We then present a general notion of *locality* for heap-allocated data which can be checked using the results obtained by the reference constancy analysis, and which determines if heap-allocated data behave as local variables.
3. Sometimes, heap-allocated data behave locally only under certain *aliasing* conditions. We introduce the notion of *locality partition* which, by assuming such aliasing conditions, guarantees the locality of the considered heap-allocated data.
4. Based on the notion of locality partition, we introduce a novel transformation which replaces the accesses to local heap-allocated data by non-heap allocated *ghost* variables. An interesting aspect of our conditional heap-sensitive analysis that we will show in the article is that we can improve the accuracy of the unconditional heap-insensitive analysis even in cases for which programs terminate unconditionally.
5. We then propose an approach for automatically inferring the aliasing preconditions that, when they hold in the initial state, guarantee the termination of the program under consideration.
6. We implement our approach in the COSTA system [8], a cost and Termination Analyzer for Java bytecode, and evaluate it on the Apache Commons Libraries [26].

1.2. Organization of the article

This article is organized as follows. The next section briefly describes the language we consider, which is an intermediate (rule-based) representation of Java bytecode [22], and its semantics. Section 3 is devoted to presenting the reference constancy analysis for programs written in this language.

Section 4 introduces the heap-sensitive analysis in three main steps. We first define a simple locality condition which relies on the information inferred by the reference constancy analysis. Then, we discuss that such a condition might only hold under some *aliasing* (or *not aliasing*) conditions among the heap accesses. This leads to the notion of *locality partition* explained above. We can finally present a transformation which actually replaces the heap accesses which meet the locality condition by ghost variables for the given locality partition.

In Section 4, we do not specify how the termination (aliasing) preconditions can be generated. This is considered in Section 5 where we propose an approach to infer the conditions based on two notions of termination: *local termination*, which guarantees that the loops defined in a given scope S are terminating, ignoring the termination behavior of loops