Abstract

Software commonly consumes unexpectedly high amounts of memory, frequently due to programming idioms that are used to make software more reliable, maintainable and understandable. In the case of modern object-oriented systems this problem is partly due to creation of large numbers of co-existing isomorphic objects. Intuitively, two objects are isomorphic if they are of the same type, have identical values in corresponding primitive _elds, and are such that corresponding reference _elds themselves point to isomorphic objects. In other words, the portions of memory rooted at the two objects are isomorphic shape-wise as well as values-wise. A significant reduction in heap usage can therefore be achieved if the code is refactored to de-duplicate or share objects whenever possible instead of always creating distinct but possibly isomorphic objects. Such a refactoring, which employs a cache to keep track of objects created so far and to share them, is termed as object-sharing refactoring. In practice, object-sharing refactoring is commonly used, as it has the potential to reduce memory utilization significantly. However, manual refactoring is tedious and error prone.

To support object-sharing refactoring we have (1) designed and implemented an approach to estimate memory-savings potential due to this refactoring, (2) espoused the idea of full initialization points of objects, and a static analysis to identify these points, and (3) proposed a scalable refinement-based points-to analysis for verifying the safety of object-sharing refactoring, but which has general applicability to several other verification tasks as well.

1) We present a dynamic analysis technique for estimating for all the allocation sites in a program, for a given input, the reduction in heap memory usage (in bytes, or as a percentage) to be obtained by employing object sharing at each site. The quantitative estimates produced by our technique of a user-observable benefit (i.e., actual memory savings) make it easier for developers to select sites to refactor. Experimentation with our estimation tool on real Java programs indicate that nearly all applications have potential for reduction of memory usage by object sharing, with a mean savings of 12.62% per benchmark.

(2) We define a novel concept termed full-initialization points (FIPs) to characterize the points in the program where objects allocated at any chosen allocation site become fully initialized. We present a novel and conservative static analysis to detect FIPs for a given allocation site. By introducing code to cache and share objects at the FIPs suggested by our analysis, object-sharing refactoring was able to obtain a mean memory savings of 11.4% on a set of real Java benchmarks.

(3) A standard points-to analysis approach, namely, the object sensitivity approach, uses an equal level of precision to represent symbolic objects allocated at all allocation sites in a program. This

approach does not scale to large programs unless low levels of precision are used. We propose a novel, program-slicing based approach to improve the precision of object sensitivity analysis to answer user queries more precisely. Our slicing technique differs from most standard techniques for context-sensitive slicing of Java programs, which are based on system dependence graphs and which do not scale readily to larger programs. Our evaluation reveals that for a given time budget (6 hours per benchmark), our approach gives more precise results than the most precise results possible under the baseline object sensitivity analysis on nine of the 10 DaCapo benchmarks. Our approach exhibits 28% greater precision over the baseline object sensitivity approach in identifying allocation sites where object-sharing refactoring can be performed.