Theoretical Computer Science 585 (2015) 91–114

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

A multi-domain incremental analysis engine and its application to incremental resource analysis ☆

Elvira Albert [a,*], Jesús Correas [a], Germán Puebla [b], Guillermo Román-Díez [b]

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

ARTICLE INFO

Article history:
Available online 6 March 2015

Keywords:
Static analysis
Resource usage analysis
Cost analysis
Incremental analysis

ABSTRACT

The aim of *incremental* analysis is, given a program, its analysis results, and a series of changes to the program, to obtain the new analysis results as efficiently as possible and, ideally, without having to (re-)analyze fragments of code which are not affected by the changes. Incremental analysis can significantly reduce both the time and the memory requirements of analysis. The first contribution of this article is a *multi-domain* incremental fixed-point algorithm for a sequential Java-like language. The algorithm is multi-domain in the sense that it interleaves the (re-)analysis for multiple domains by taking into account dependencies among them. Importantly, this allows the incremental analyzer to invalidate only those analysis results previously inferred by certain *dependent* domains. The second contribution is an incremental *resource usage* analysis which, in its first phase, uses the multi-domain incremental fixed-point algorithm to carry out all global pre-analyses required to infer cost in an interleaved way. Such resource analysis is parametric on the cost metrics one wants to measure (e.g., number of executed instructions, number of objects created, etc.). Besides, we present a novel form of *cost summaries* which allows us to incrementally reconstruct only those components of cost functions affected by the changes. Experimental results in the COSTA system show that the proposed incremental analysis provides significant performance gains, ranging from a speedup of 1.48 up to 5.13 times faster than non-incremental analysis.

1. Introduction

Static cost analysis [39] (a.k.a. resource usage analysis) aims at automatically inferring the resource consumption of executing a program as a function of its input data *sizes*, i.e., without actually executing the program. In this work, we rely on a generic notion of resource, which can be instantiated to measure the amount of memory allocated, number of instructions executed, number of calls to methods, etc. Intuitively, the main steps in order to infer the cost of programs written in an object-oriented (OO) language are:

1. *OO pre-analyses*. Almost for every property being analyzed, it is required to perform a *class* (or *application extraction*) analysis [34] which determines the set of reachable classes which must be considered by subsequent global analyses.

☆ This work is an extended and revised version of PEPM'12 [5].

* Corresponding author.
E-mail addresses: elvira@sip.ucm.es (E. Albert), jcorreas@fdi.ucm.es (J. Correas), german@fi.upm.es (G. Puebla), groman@fi.upm.es (G. Román-Díez).

http://dx.doi.org/10.1016/j.tcs.2015.03.002
0304-3975/

Besides, analyzers of OO languages often perform non-nullness analysis [22,33] which allows removing unsatisfiable nullness checks.

2. *Cost relations*. Given the program and the pre-analyses information, this step consists in setting up *cost recurrence equations*, or cost relations for short (CRs), which define the cost of executing the program in terms of the input data sizes. The global analysis underlying this step is the inference of size relations which determine how the sizes of data change along program's execution [3]. In the presence of heap-allocated data structures, size analysis based on path-length [35] relies on a series of pre-analyses, namely, *sharing*, *acyclicity* and *constancy*.
3. *Cost functions*. In the last step, cost relation solvers [2] try to obtain *cost functions* which are not in recursive form and hence are directly evaluable. Since exact solutions seldom exist, analyzers infer upper/lower bounds for the CRs. This is again a global process which starts by solving the CRs which do not depend on any other one and continues by replacing the computed cost functions on the equations which call such relations until all CRs are solved.

Hence, cost analysis is performed by a sequence of *global* analyses, i.e., which require to analyze the whole program in order to obtain sound and precise results. Despite the great progress made in static analysis, most global analyzers still read and analyze the entire program at once in a non-incremental way. In particular, current state of the art resource analyses are non-incremental [3,18,21]. Incremental analysis has applications in the following two scenarios: (1) *Software development.* During software development, programs are often modified, e.g., because a new implementation of an existing method is provided (which improves its efficiency or fixes its correctness) or because an existing code is extended with new functionality (typically by extending a class with further methods). In such cases, the existing analysis information for the program may no longer be correct and/or accurate. (2) *SPLE.* One increasing trend in software engineering is to develop multiple, similar software products instead of just a single individual program. Software Product Line Engineering (SPLE) [10] offers a solution which is based on the explicit modeling of what is common and what differs between product variants, and on building a reuse infrastructure (product line asset) that can be instantiated and possibly extended to build the desired similar products. Building a product consists in incrementally assembling the product from the product line assets by applying the selected features.

Resource analysis is a compute-intensive task and, in scenarios as those mentioned above, starting analysis from scratch (instead of reusing previous results) is inefficient in most cases. Consider a given program, its analysis results and a series of changes to the program, e.g., extensions to build a new product in the SPLE scenario or modifications to fix a bug in the software development scenario. Incremental resource usage analysis aims at obtaining the new analysis results more efficiently, without having to (re-)analyze fragments of code which are not affected by the changes.

1.1. Summary of contributions

In this article, we present a generic incremental multi-domain analysis engine for an imperative object-oriented programming language, and study its application in the context of incremental resource usage analysis. The main challenge when devising an incremental analysis framework is to recompute the least possible information and do it in the most efficient way. Our main contributions can be summarized as follows:

- We introduce a *multi-domain* incremental analysis engine which interleaves the computation for multiple analysis domains. Dealing with a large number of pre-analyses, and threading the information about change through all of them is the main challenge we face here. Our algorithm takes into account the dependencies among them in such a way that it is possible to invalidate only part of the pre-computed dependent information.
- We describe how the previous algorithm can be used in order to incrementally compute all global pre-analyses required to infer the resource usage of a program (including class analysis, nullness, sharing, cyclicity, constancy and size analyses mentioned above). All such analysis information is included in the so-called *cost method summary* and used by the multi-domain incremental analysis engine.
- Even a small change within a method (e.g., adding an instruction) can change the overall cost of the program. Our contribution, in order to minimize the amount of information that needs to be recomputed, is on the notion of *upper bound summary* which allows us to distinguish the cost subcomponents associated to each method, so that the final cost functions can be recomputed by replacing only the affected subcomponents.
- The correctness of our approach has been proved within the article. In addition, complete details of some proofs can be found in Appendix A.
- We have implemented the incremental analysis in the COSTA system, a cost and termination analyzer for Java bytecode programs. Experimental results are performed on selected benchmarks from the standardized *JOlden* benchmark suite [36] and from the *Apache Commons* Project [28]. Our results show that the proposed incremental analysis achieves a significant speedup with respect to the non-incremental approach.

To the best of our knowledge, this is the first approach to the incremental inference of resource usage bounds.