# Resource Analysis in the COSTA System

E. Albert[1], D. Alonso[1], P. Arenas[1], J. Correas[1],
A. Flores[1], S. Genaim[1], M. Gómez-Zamalloa[1], A. Masud[2],
G. Puebla[2], J.M. Rojas[2], G. Román-Díez[2], and D. Zanardini[2]

[1] DSIC, Complutense University of Madrid (UCM), Spain
[2] DLSIIS, Technical University of Madrid (UPM), Spain

Having information about the execution cost of programs, i.e., the amount of resources that the execution will require, is useful for many different purposes, including program optimization, verification and certification. Reasoning about execution cost is difficult and error-prone. Therefore, it is widely recognized that *cost analysis*, sometimes also referred to as *resource analysis* or *automatic complexity analysis*, is quite important. COSTA[3] is a state-of-the-art cost and termination analyzer which automates this task. The system is able to infer upper and lower bounds on the resource consumption of a large class of programs. Given a program $P$, the analysis results allow bounding the cost of executing $P$ on any input data $\overline{x}$ without having to actually *run* $P(\overline{x})$.

The first successful proposal for *automatically* computing the complexity of programs was the seminal work of Wegbreit [33]. Since then, a number of cost analysis frameworks have been proposed, mostly in the context of *declarative* programming languages (functional programming [23, 27, 32, 28, 17] and logic programming [18, 25]). Cost analysis of imperative programming languages has received significantly less attention. It is worth mentioning the pioneering work of [1]. To the best of our knowledge, COSTA has been the first system which automatically infers bounds on cost for a large class of Java-like programs, getting meaningful results. The system is implemented in Prolog (it runs both on Ciao [20] and SWI Prolog [34]) and uses the Parma Polyhedra Library [16] for manipulating linear constraints. More info on COSTA is available from its website[3], where it can be used from a web interface.

## 1 From Programs to Cost Relations

The classical approach to static cost analysis consists of two phases. First, given a program and a *cost model*, the analysis produces *cost relations*, i.e., a system of recursive equations which capture the cost of the program in terms of the size of its input data. In a second phase described below, the cost relations are solved into an upper and/or a lower bound. This first phase is in turn twofold:

*Intermediate representation.* The program is first transformed into a recursive intermediate representation (IR) [6]. The transformation starts by constructing

[3] Further information of the system is available at http://costa.ls.fi.upm.es.

the Control Flow Graph (CFG) of the program. Each block of the CFG is transformed into one rule in the IR. Iteration is transformed into recursion and conditional constructs are represented as multiple (mutually exclusive) guarded rules. Intermediate programs resemble declarative programs due to their rule-based form. However, they are still imperative, since they use destructive assignment and store data in mutable data structures (stored in a global memory or heap).

*Generation of cost relations.* A *cost model* declares the cost of each basic instruction of the language. COSTA incorporates a range of cost models for counting the number of executed instructions, the memory consumed, the number of invocations to a given method, among others. Given a program (in intermediate form) and a selection of a cost model, we automatically generate *cost relations* (CRs) which define the cost of executing the program w.r.t. the selected cost model. CRs are basically an extended form of recurrence relations defined by equations of this form $\langle c(\bar{x}) = e, \varphi \rangle$, where $e$ is a cost expression and $\varphi$ is a set of linear constraints which define the applicability conditions for the equations and the size relations among the variables in the equation. The inference of $\varphi$ is done by means of an *abstract compilation* of the program and a subsequent *size analysis* (see [4] for more details).

## 2 From Cost Relations to Closed-Form Bounds

Though CRs are simpler than the programs they originate from, since all variables are of integer type, in several respects they are not as static as one would expect from the result of a static analysis.

One reason is that they are recursive and thus we may need to iterate for computing their value for concrete input values. Another reason is that even for deterministic programs, it is well known that the loss of precision introduced by the size abstraction may result in cost relations which are non-deterministic. For both reasons, it is clear that it is interesting to compute *closed-form* upper bounds (UB for short) and lower bounds (LB) for the cost relation, whenever this is possible, i.e., bounds which are not in recursive form. For inferring closed-form bounds, COSTA relies on program analysis techniques instead of on purely mathematical approaches as those incorporated in computer algebra systems. Indeed, if one ignores the cost expressions in the equations, CRs are *constraint logic programs*, and existing techniques developed in the context of CLP can be easily adapted to reason on the equations. The choice of treating CRs as (CLP) programs is probably the main reason for which COSTA has succeeded where previous cost analyzers have failed.

*Upper Bounds.* The first step for inferring closed-form bounds consists in applying *partial evaluation* [22] to make all recursive calls in the CRs direct. This makes the loops, and their caller-callee relation, simpler and explicit. Then, an UB for each loop is inferred in a compositional manner by using one of the following two approaches. The first approach [4] consists in inferring an UB $f(\bar{x}_0)$

on the number of iterations of the loop and an UB on the worst-case cost $e(\bar{x}_0)$ of its iterations, and then $f(\bar{x}_0) * e(\bar{x}_0)$ is a sound UB. Here, $f$ is inferred by synthesizing a ranking function [26] for the corresponding loop and $e$ by inferring a loop invariant and then using parametric integer programming [19]. This approach is widely applicable and reasonably precise. However, when the cost accumulated varies significantly from one iteration to another, taking the worst case cost for all iterations is too imprecise. The second approach [12] allows obtaining more precise UBs by simulating the variation from one iteration to the next one using recurrence relations. While this approach computes very precise UBs, it is less applicable that the first one.

*Lower Bounds.* In the latter approach, i.e., [12], the inference of LBs is a dual problem to that of inferring UBs. The main difference is that one has to use (new) techniques for inferring LBs on the number of iterations and for obtaining the best-case cost of each iteration.

*Amortized cost analysis.* There are programs for which *amortized cost analysis* [31] is required in order to produce precise UBs for them. In such programs, costly operations cannot occur with high frequency and thus taking their worst-case cost, as in CRs, would lead to a very imprecise bound. To overcome these limitations, we have recently developed [15] techniques based on a novel definition of CRs that involve input an output arguments, and a corresponding solving procedure that is based on real quantifier elimination. However, this subsystem is still prototypical and not fully integrated in COSTA.

## 3 Advanced Topics in Resource Analysis

We briefly overview some advanced topics in resource analysis which we have investigated within the context of the COSTA system.

### 3.1 Non-Cumulative Resources

The approach described so far can be used to estimate the *accumulated* cost along the program's execution. Unfortunately, it is not suitable to infer non-cumulative resources (e.g., the peak memory consumption or the task-level described below). This is because non-cumulative resources require reasoning on all possible states to obtain their maximum.

*Memory Consumption.* In the context of languages with garbage collection, such as Java, the amount of data stored in the heap increases and decreases along the execution and, therefore, the interest is in estimating the maximum memory consumption (a.k.a. peak memory consumption). COSTA develops a novel approach [11, 10, 9] which is *parametric* on the notion of *object lifetime*, i.e., on when objects become collectible. If objects lifetimes are inferred by a *reachability* analysis, then our analysis infers accurate upper bounds on the memory consumption for a reachability-based garbage collector. Interestingly, if objects lifetimes are inferred by a *heap liveness* analysis, then we approximate the program *minimal memory requirement*, i.e., the peak memory usage when using an optimal garbage collector which frees objects as soon as they become dead. The key idea is to integrate information on objects lifetimes into the process of generating the recurrence relations which capture the memory usage at the different program states.

*Task-Level.* In parallel programs, the number of *tasks* which can be active at the same time is a valuable piece of information, which is also a non-cumulative type of resource. Knowledge on the task-level is useful to maximize the performance when the computation must be carefully distributed among the processors of a multicore architecture. COSTA features a static analysis [13] which can find upper bounds to the level of parallelism of programs written in languages (the analysis works on a subset of the X10 language [24]) whose concurrency model is based on the *async* and *finish* primitives. As before, the key idea is to integrate information on the lifetime of tasks into the recurrence relations.

### 3.2 Heap-Sensitive Analysis

Shared mutable data structures, such as those stored in the heap, are the bane of formal reasoning and static analysis. In object-oriented programs most data reside in objects and arrays stored in the heap. Analyses which keep track of heap-allocated data are referred to as *heap-sensitive.* The approach developed in COSTA [14, 5, 3] is based on the observation that, by analyzing scopes, rather than the application as a whole, it is often possible to keep track of heap accesses in a similar way to local variables. Under some *locality* conditions, heap accesses can be replaced by equivalent non-heap allocated *ghost* variables. The resulting program can then be the input to any heap-insensitive static analyzer, in particular, to the one that we already have in COSTA.

### 3.3 Incremental Resource Analysis

A key challenge for static analysis techniques is achieving a satisfactory combination of precision and scalability. Making precise (and hence expensive) static analysis incremental is a step forward in this direction. The difficulty when devising an incremental analysis framework is to recompute the least possible information and do it in the most efficient way. In other approaches to incremental analysis, such as in the logic programming context [21], the analysis is focused in a single abstract domain. In contrast, COSTA includes a *multi-domain* incremental analysis engine [7] which can be used by all global pre-analyses required to infer the resource usage of a program (including class analysis, sharing, cyclicity, constancy and size analysis). The algorithm is multi-domain in the sense that it interleaves the computation for the different domains and takes into account dependencies among them, in such a way that it is possible to invalidate only partial pre-computed information. A fundamental idea to minimize the amount of information which needs to be recomputed is to be able to distinguish within a *cost summary* the cost subcomponent associated to each method, so that the final cost functions can be recomputed by replacing only the affected subcomponents.

## 3.4 Concurrency and Distribution

In order to develop a resource analysis for distributed and concurrent programs, we have considered a concurrency model based on the notion of concurrently running (groups of) objects, similar to the actor-based and active-objects approaches [29, 30]. These models take advantage of the concurrency implicit in the notion of object in order to provide programmers with high-level concurrency constructs that help in producing concurrent applications more modularly and in a less error-prone way. We have incorporated in COSTA a static cost analysis [2, 8] for concurrent OO programs, which is parametric w.r.t. the notion of resource that can be instantiated to measure both traditional and concurrency-related resources. The main novelty of the analysis is that it provides the resource consumption per *cost center*, where each cost center represents a distributed component. Having anticipated knowledge on the resource consumption of the different components which constitute a system is useful for distributing the load of work. Upper bounds can be used to predict that one component may receive a large amount of remote requests, while other siblings are idle most of the time. Also, our framework allows instantiating the different components with the particular features of the infrastructure on which they are deployed.

*Acknowledgments.* This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, by the Spanish Ministry of Science and Innovation (MICINN) under the TIN-2008-05624 and PRI-AIBDE-2011-0900 projects, by UCM-BSCH-GR35/10-A-910502 grant and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS-CM* project.

## References

1. A. Adachi, T. Kasai, and E. Moriya. A Theoretical Study of the Time Analysis of Programs. In *Proc. of MFCS'79*, volume 74 of *LNCS*, pages 201–207. Springer, 1979.
2. E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. Cost Analysis of Concurrent OO programs. In *Proc. of APLAS'11*, volume 7078 of *LNCS*, pages 238–254. Springer, 2011.
3. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Field-Sensitive Value Analysis by Field-Insensitive Analysis. In *Proc of FM'09*, volume 5850 of *LNCS*, pages 370–386. Springer, 2009.
4. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011.

5. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Ramírez. From Object Fields to Local Variables: A Practical Approach to Field-Sensitive Analysis. In *Proc. of SAS'10*, volume 6337 of *LNCS*, pages 100–116, Springer, 2010.
6. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. *Theoretical Computer Science (Special Issue on Quantitative Aspects of Programming Languages)*, 413(1):142–159, 2012.
7. E. Albert, J. Correas, G. Puebla, and G. Román-Díez. Incremental Resource Usage Analysis. In *Proc. of PEPM'12*, pages 25–34. ACM Press, 2012.
8. E. Albert, A. Flores-Montoya, and S. Genaim. Analysis of May-Happen-in-Parallel in Concurrent Objects. Technical report, UCM, 2012. Available at: https://costa.ls.fi.upm.es/papers/discotec12/paper.pdf.
9. E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap Space Analysis for Java Bytecode. In *Proceedings of the 6th International Symposium on Memory Management (ISMM'07)*, pages 105–116, New York, NY, USA, October 2007. ACM Press.
10. E. Albert, S. Genaim, and M. Gómez-Zamalloa. Live Heap Space Analysis for Languages with Garbage Collection. In *Proc. of ISMM'09*, pages 129–138. ACM Press, 2009.
11. E. Albert, S. Genaim, and M. Gómez-Zamalloa. Parametric Inference of Memory Requirements for Garbage Collected Languages. In *Proc. of ISMM'10*, pages 121–130. ACM Press, 2010.
12. E. Albert, S. Genaim, and A. N. Masud. More Precise yet Widely Applicable Cost Analysis. In *Proc. of VMCAI'11*, volume 6538 of *LNCS*, pages 38–53. Springer, 2011.
13. E. Albert, P. Arenas, S. Genaim, and D. Zanardini. Task-Level Analysis for a Language with Async-Finish parallelism. In *Proc. of LCTES'11*, pages 21–30. ACM, 2011.
14. E. Albert, S. Genaim, and G. Román-Díez. Conditional Termination of Loops over Arrays. In *Proc. of Bytecode'12*, 2012. To appear.
15. D. Alonso and S. Genaim. On the Limitations of the Classical Approach to Cost Analysis. In *Proc. of SAS'12*, volume 7460 of *LNCS*, pages 405–421, Springer, 2012.
16. R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems. *Science of Computer Programming*, 72(1–2), 2008.
17. R. Benzinger. Automated Higher-Order Complexity Analysis. *Theoretical Computer Science*, 318(1-2), 2004.
18. S. K. Debray and N. W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, 1993.
19. P. Feautrier. Parametric Integer Programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.
20. M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J.F. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming*, 12(1–2):219–252, January 2012.
21. M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, March 2000.
22. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

23. D. Le Metayer. ACE: An Automatic Complexity Evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2), 1988.
24. J. K. Lee and J. Palsberg. Featherweight X10: A Core Calculus for Async-Finish Parallelism. In *Proc. of PPoPP'10*, pages 25–36, ACM, 2010.
25. J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *Proc. of ICLP'07*, volume 4670 of *LNCS*. pages, 348-363, Springer, 2007.
26. A. Podelski and A. Rybalchenko. A Complete Method for the Synthesis of Linear Ranking Functions. In *Proc. of VMCAI'04*, volume 2937 of *LNCS*, pages 239–251, Springer, 2004.
27. M. Rosendahl. Automatic Complexity Analysis. In *Proc. of FPCA'89*, pages 144-156, ACM Press, 1989.
28. D. Sands. A Naïve Time Analysis and its Theory of Cost Equivalence. *Journal of Logic and Computation*, 5(4), 1995.
29. J. Schäfer and A. Poetzsch-Heffter. Jcobox: Generalizing Active Objects to Concurrent Components. In *Proc. of ECOOP'10*, volume 6183 of *LNCS*, pages 275–299. Springer, 2010.
30. S. Srinivasan and A. Mycroft. Kilim: Isolation-Typed Actors for Java. In *Proc. of ECOOP'08*, volume 5142 of *LNCS*, pages 104–128. Springer, 2008.
31. Robert Endre Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.
32. P. Wadler. Strictness Analysis Aids Time Analysis. In *Proc. of POPL'88*, pages 119-132, ACM Press, 1988.
33. B. Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9), 1975.
34. Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *TPLP*, 12(1-2):67–96, 2012.