Automatic Inference of Resource Consumption Bounds

Elvira Albert[1], Puri Arenas[1], Samir Genaim[1], Miguel Gómez-Zamalloa[1], and Germán Puebla[2]

[1] DSIC, Complutense University of Madrid, Spain
[2] DLSIIS, Technical University of Madrid, Spain

Abstract. One of the main features of programs is the amount of resources which are needed in order to run them. Different resources can be taken into consideration, such as the number of execution steps, amount of memory allocated, number of calls to certain methods, etc. Unfortunately, manually determining the resource consumption of programs is difficult and error-prone. We provide an overview of a state of the art framework for automatically obtaining both upper and lower bounds on the resource consumption of programs. The bounds obtained are functions on the size of the input arguments to the program and are obtained statically, i.e., without running the program. Due to the approximations introduced, the framework can fail to obtain (non-trivial) bounds even if they exist. On the other hand, modulo implementation bugs, the bounds thus obtained are valid for any execution of the program. The framework has been implemented in the COSTA system and can provide useful bounds for realistic object-oriented and actor-based concurrent programs.

1 Introduction

One of the most important characteristics of a program is the amount of resources that its execution will require, i.e., its *resource consumption*. Resource analysis (a.k.a. cost analysis [37]) aims at *statically* bounding the cost of executing programs for any possible input data value. Typical examples of resources include execution time, memory watermark, amount of data transmitted over the net, etc. Resource usage information has many applications, both during program development and deployment. *Upper bounds* are useful because they provide *resource guarantees*, i.e., it is ensured that the execution of the program will never exceed the amount of resources inferred by the analysis. *Lower bounds* on the resource usage have applications in program parallelization, they can be used to decide if it is worth executing locally a task or requesting remote execution. Therefore, automated ways of estimating resource usage are quite useful and the general area of resource analysis has received [37,21,33] and is nowadays receiving [10,23,25] considerable attention. In this paper, we describe the main components underlying resource analysis of a today's imperative programming language, e.g., such techniques have been applied to analyze the resource consumption of sequential Java, Java bytecode [28], Featherweight X10 [27] and concurrent ABS programs [26].

The rest of the paper is organized as follows. Section 2 describes the process of, from the program, generating *cost relations* which define, by means of recurrence equations, the resource consumption of executing the program in terms of the input data sizes. Section 3 overviews a general approach to, from the cost relations, obtain upper and lower bounds which are not in recursive form. The compositionallity and incrementality of the analysis are described in Section 4. Standard cost analysis can be applied to infer any *cumulative* type of resource which always increases along the execution. We discuss in Section 5 the required extensions to estimate non-cumulative resources like the memory consumption in the presence of garbage collection. As distribution and concurrency are now mainstream, one of the most interesting extensions is to handle concurrency in cost analysis. This will be described in Section 6. The results of the analysis are correct only if the implementation does not contain bugs. In Section 7, we describe how the analysis results can be certified by existing program verification tools. Finally, in Section 8 we conclude and point out directions for future research.

2 Generation of Cost Relations

In the first phase, cost analysis takes as input a program, a selection of a cost model (among those available in the system), and yields a set of *recursive equations* which capture the cost of executing the program. For a general purpose programming language, the following steps are performed in order to generate the equations:

1 A *control flow graph* is constructed for each method in the original program by using standard techniques from *compiler theory* [1,2].

2 The control flow graph can then be represented by using some intermediate formalism, with the purpose of making the subsequent static analysis simpler. In [10], we propose that the control flow graph is represented as a set of procedures (defined by one or more rules) by using a *rule-based*, recursive *representation.*

3 Static analysis can be then performed on the rule-base representation in order to infer, for each rule, *size relations*, which define the size relationships among the input variables to the rule and the variables in the calls performed within the rule.

4 A parametric notion of *cost model* is used, which allows specifying the resource of interest (e.g., steps, memory). In particular, the cost model defines the cost assigned to each execution step and, by extension, to an entire execution trace.

5 From the rule-based representation, the size relations, and the selected cost model, a *cost relation system* is automatically generated. Cost relations are

```
static void m(List x, int i, int n){
  while (i<n){
    if (x.data) {g(i,n); i++;}
    else {g(0,i); n=n-1;}
    x=x.next;
}}
```

$$
\begin{array}{l}
(1)\ \langle C_m(i,n) = 3 \\
\qquad\qquad ,\varphi_1 = \{i \geq n\}\rangle \\
(2)\ \langle C_m(i,n) = 15 + C_g(i,n) + C_m(i',n) \\
\qquad\qquad ,\varphi_2 = \{i < n, i' = i+1\}\rangle \\
(3)\ \langle C_m(i,n) = 17 + C_g(0,i) + C_m(i,n') \\
\qquad\qquad ,\varphi_3 = \{i < n, n' = n-1\}\rangle
\end{array}
$$

Fig. 1. Java method and Cost Relation

defined by means of recursive expressions which define the cost of executing a block in the control flow graph (or rule in the rule-based representation) in terms of the cost of executing the block itself plus the cost of its successor blocks.

All details about how to automatically obtain a cost relation from a program can be found in [8,10,22,32,37]. We illustrate the notion of cost relation systems by means of a simple example.

Example 1. Consider the Java method m shown in Figure 1 (left), which invokes the auxiliary method g, where x is a linked list of Boolean values implemented in the standard way. We have selected a cost model which counts the number of executed instructions. The cost relation associated to method m is shown in Figure 1 (right). The relations C_m and C_g capture, respectively, the costs of executing methods m and g. Intuitively, in cost relations, variables represent the sizes of the corresponding data structures in the program and in the case of integer variables they represent their integer value. Equation 1 is a base case and captures the case where the loop body is not executed. It can be observed that we have two recursive equations (Equations 2 and 3) which capture the respective costs of the then and else branches within the while loop. The constraints in the equations (namely φ_1, φ_2 and φ_3) contain the applicability conditions for the equations and also the size constraints computed in the previous step of the analysis (see Section 2). As the list x has been abstracted to its length, the values of x.data are not visible in the cost relation and the two equations have the same (incomplete) guard, which results in a *non-deterministic* cost relation. Also, variables which do not affect the cost (e.g., x) do not appear in the cost relation [9].

3 Inference of Closed-Form Bounds

The second main phase of cost analysis consists in generating *closed-form* bounds from the cost relations, i.e., cost expressions which are not in recursive form. Let us first see how we can infer an upper bound for m from the equations in Figure 1. We shall start by computing upper bounds for the cost relations which do not depend on any other cost relations, referred to as standalone cost relations, and continue by replacing the computed upper bounds on the equations which call such relations. For instance, assuming that the upper bound for g is $C_g^+(a,b)$=4+5$*\mathsf{nat}(b-a)$, where $\mathsf{nat}(v) = max(\{v,0\})$, the cost relation in Figure 1 becomes standalone:

$$\begin{array}{llll}
(1) & \langle C_m(i,n) = 3 & & , \varphi_1 = \{i \geq n\}\rangle \\
(2) & \langle C_m(i,n) = 15 + \mathsf{nat}(n-i) & +C_m(i',n) & , \varphi_2 = \{i < n, i' = i+1\}\rangle \\
(3) & \langle C_m(i,n) = 17 + \mathsf{nat}(i) & +C_m(i,n') & , \varphi_3 = \{i < n, n' = n-1\}\rangle
\end{array}$$

The use of nat in cost expressions is required in order to avoid incorrectly evaluating upper bounds to negative values (see [7]), as for instance, when $b < a$ in $C_g^+(a,b)$. The problem is thus now reduced to automatically inferring bounds from standalone relations. In general, given a standalone cost relation made up of nb base cases of the form $\langle C(\bar{x}){=}base_j, \varphi_j\rangle$, $1 \leq j \leq nb$ and nr recursive equations of the form, $\langle C(\bar{x}){=}rec_j{+}\sum_{i=1}^{k_j} C(\bar{y}_i), \varphi_j\rangle$, $1 \leq j \leq nr$, [7] proposes to compute an upper bound for $C(\bar{x})$, denoted $C(\bar{x})^+$, as follows:

$$(*) \qquad C(\bar{x})^+ = \mathsf{l_b} * worst(\{base_1, \ldots, base_{nb}\}) + \mathsf{l_r} * worst(\{rec_1, \ldots, rec_{nr}\})$$

where $\mathsf{l_b}$ and $\mathsf{l_r}$ are, respectively, upper bounds of the number of visits to the base cases and recursive equations and $worst(\{Set\})$ denotes the worst-case (the maximum) value that the expressions in Set can take. The first challenge is thus to have an automatic method to compute $\mathsf{l_b}$ and $\mathsf{l_r}$. In [7], we have proposed the use of *ranking functions* to automatically find $\mathsf{l_b}$ and $\mathsf{l_r}$. Intuitively, a ranking function [30] is a function on the variables of a relation which (1) is positive and (2) decreases in each iteration. For our example, a ranking function is $f_{C_m}(i,n) = n-i$ since the constraints of the corresponding equations imply the previous two conditions, i.e., $\varphi_2 \models f(i,n) > f(i',n) \wedge f(i,n) > 0$ and $\varphi_3 \models f(i,n) > f(i',n) \wedge f(i,n) > 0$.

The next challenge is to have an automatic method to compute the maximum value $worst(\{Set\})$. Following [6], in order to obtain $worst(\{Set\})$, first we need to infer *invariants* between the equation's variables and their initial values. For example, the cost relation $C_m(i,n)$ admits as invariant for the recursive equations the formula $\mathcal{I}$ defined as $\mathcal{I}((i_0,n_0),(i,n)) \equiv i \geq i_0 \wedge n \leq n_0 \wedge i < n$, which captures that the values of i (resp. n) are greater (resp. smaller) or equal than the initial value and that i is smaller than n at all iterations. Once we have the invariant, we can *maximize* the expressions w.r.t. these values and take the maximal: $worst(\{rec_1, \ldots, rec_{nr}\}) = \max(maximize(\mathcal{I}, \{rec_1, \ldots, rec_{nr}\}))$ (see [6] for the technical details of the maximization procedure). For instance, for our cost relation, we compute:

$$worst(\{rec_1, rec_2\}) = \max(maximize(\mathcal{I}, \{\mathsf{nat}(n{-}i), \mathsf{nat}(i)\}))$$

which results in $worst(\{rec_1, rec_2\}) = \max(\{\mathsf{nat}(n_0 - i_0), \mathsf{nat}(n_0{-}1)\})$. The same procedure can be applied to the expressions in the base cases. However, it is unnecessary in our example, because the base case is a constant. By applying the same reasoning to the standalone cost relation above, we obtain the following upper bound (on the number of instructions):

$$C_m^+=6+\mathsf{nat}(n-i)*\max(\{21+5*\mathsf{nat}(n-1),19+5*\mathsf{nat}(n-i)\})$$

The framework could be adapted to infer closed-form lower bounds by using minimization instead of maximization (i.e., using min instead of max when defining *best*, the counterpart of *worst*). However, taking always the best case cost of all iterations would lead to a too pessimistic lower bound, indeed, the obtained lower bound would be in most cases zero. In [16], we propose a refined method in which, instead of assuming the best-case cost for all iterations, we infer tighter bounds on each of them in an automatic way and then approximate the summation of the sequence.

4 Modularity and Incrementality

Typically, cost analysis performs a global analysis of the program which requires that all reachable code is available. Such traditional analysis scheme in which all code is analyzed from scratch, and no previous analysis information is available, is unsatisfactory in many situations. For a practical uptake of cost analysis, it is thus required to reach some degree of compositionallity which allows decomposing the analysis of large programs into the analysis of smaller parts.

In [31], a *modular* approach for the particular case of termination analysis is presented, which allows reasoning on a method at a time. This approach is generalized in [13], where an *incremental* resource analysis scheme is presented. The aim of incremental global 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.

5 Memory Consumption Analysis for Garbage-Collected Languages

Predicting the memory required to run a program is crucial in many contexts like in embedded applications with stringent space requirements or in real-time systems which must respond to events or signals within a predefined amount of time. When the considered resource is *cumulative*, the approach to cost analysis described so far is parametric w.r.t. the notion of *cost model*, which defines the type of resource we are measuring and, which gives a cost unit to each instruction. It can therefore be directly applied for inferring bounds on the *total* memory usage of programs by devising a cost model which assigns the corresponding cost only to those instructions that consume memory [14].

However, in presence of *garbage collection*, memory is not a cumulative resource but rather it can increase or decrease at any point of the execution.

Thus, the above approach, though still correct, can produce too pessimistic estimations. *Peak memory analysis* [36,19,20], also known as *live memory analysis*, aims at approximating the maximum memory usage during a program's execution, which provides a much tighter estimation. Whereas analyzing the total memory consumption needs to observe the consumption at the *final* state only, peak memory analysis has to reason on the memory consumption at *all program states* along the execution, hence making the analysis much more complicated. In [15], we present a peak memory analysis which is parametric w.r.t the *lifetime* of objects and which can therefore be instantiated with different garbage collection strategies. This requires a non-standard type of recurrence equations.

Example 2. Let us consider method m of Figure. 1, and let us assume that method g only creates an array of 10 integers. Our memory consumption analysis obtains an upper bound of $40{*}\mathsf{nat}(n{-}i)$ bytes for the total memory usage of method m (assuming that the size of an integer is 4 bytes). Let us now consider a *scope-based* garbage collection, i.e., one that reclaims the local memory created in a method call when it finishes. In this case, our memory consumption analysis obtains a constant upper bound of 40 bytes for the peak memory usage of method m. This intuitively corresponds to the maximum between the peak usage of method g (40 bytes) and what *escapes* (i.e. what the garbage collection cannot collect) from g plus the consumption after the call to g (which is 0 bytes).

6 Concurrency in Cost Analysis

Distribution and concurrency are now mainstream, as the availability of multiprocessors radically influences software. This brings renewed interest in developing techniques that help in understanding, analyzing, and verifying the behavior of concurrent programs. This includes the resource consumption behavior.

In concurrent and distributed settings, the meaning of a resource is not limited to traditional measures such as memory consumption or number of executed instructions, but it rather extends to other measures such as the tasks spawned [11], the number of requests to remote servers, etc. In addition, the consumption is not anymore associated to a single execution entity, but rather is distributed over the different components of the system (servers, CPUs, etc). This opens up new interesting applications for resource guarantees, for example, upper bounds can be used to predict that one component may be overloaded, while other siblings are idle most of the time; and lower bounds can be used to decide if it is worth executing locally a task or requesting remote execution.

In [4], we extended the underlying techniques used in the analysis for a concurrency model based on the notion of concurrently running (groups of) objects, in the spirit of the actor-based and active-objects approaches [34,35]. These models aim at taking advantage of the inherent 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. The main challenges we had to deal with are: (1) developing a size analysis that is able to infer sound size relations taking into account all possible interleavings between the different tasks. The main problem is that when a suspended task is resumed, it cannot assume that the global (shared) state has not been changed since the task has been suspended, but rather it should take into account that it has been possibly modified by other tasks; and (2) incorporating the idea of distributing the cost over the system components in the cost relations.

For (1), we have developed a size analysis that is based on the idea of tracking the global state when it is possible, and using class invariants at points in which interleaving might occur, in order to describe how the global state has been modified. In many cases we are able to automatically infer these invariants, and in some others the user has to provide them. For (2), we have introduced a new kind of cost relations which extend those standard ones with explicit cost centers that represent the system components on which the cost should be distributed. Example of such centers are classes, objects, and groups of objects.

7 Certified Resource Bounds

Resource guarantees, like any other program property which is automatically inferred by static analysis tools, are generally not considered to be completely trustworthy, unless the correctness of the tools or the results have been formally verified. In the case of static analyzers, verifying the correctness of the tools is a daunting task, among other things, because of the sophisticated algorithms used for the analysis and their evolution over time. Thus, in COSTA we take an alternative approach [18,12] in which we verify the results after each run using KeY, a state-of-the-art theorem prover for Java programs. This approach, which apart from being simpler, has the advantage that the proof generated by the theorem prover can then be translated to independently checkable *certificates* in the proof-carrying code style [29].

The certification component of COSTA is based on verifying that all intermediate results produced by COSTA are correct. Then, the correctness of composing them into bounds is straightforward. The intermediate results need to be verified include: (1) *ranking functions*, which are used to bound the number of iterations of each loop; (2) *loop invariants*, which provide an insight on the values that each variable can take, and relations between them; (3) *size relations*; which describe how the size of the data change when moving from one part of the program to another; and (4) *heap properties*, such as depth of data-structures and acyclicity, which are essential for inferring resource guarantees for heap manipulating programs.

8 Conclusions and Future Work

We have described the main techniques used in cost analysis of a today's programming language. Our approach is based on the traditional cost analysis framework which generates recurrence relations from programs and then solves them to obtain upper and/or lower bounds. There exist other approaches to cost analysis (e.g., [23,24]) which are not based on recurrence relations. It is hence not possible to formally compare the resulting upper bounds in the general case (more details are provided in [16]).

COSTA (available at `http://costa.ls.fi.upm.es`) is a state-of-the-art cost and termination analyzer which implements our approach. The system was originally designed to obtain upper bounds from *bytecode* programs. Analyzing bytecode has a much wider application area than analyzing Java source code since the latter is often not available. The COSTA system can be used online through a web interface and also from its Eclipse plugin. The user can provide assertions which state the expected resource consumption of selected methods by using JML notation. COSTA then tries to verify (or falsify) the assertions [5]. The assertions can also be written in asymptotic form [3]. This facilitates human reasoning, as asymptotic cost expressions are much simpler than the non-asymptotic ones.

Recently, the system has been extended to obtain upper bounds from concurrent ABS programs [26]. The resulting extension is called COSTABS [17]. ABS is an Abstract Behavioral Specification language for distributed object-oriented systems. COSTABS is integrated in the ABS development tools (available at `http://www.hats-project.eu`), which can be used within the Eclipse development environment. Moreover, it can be used through a web-interface[1], and can be downloaded and used through a command-line. In [11], we have applied similar, but simpler, techniques to an async-finish concurrency model.

As regards certifying the upper bounds, we have succeed to make the COSTA and KeY system cooperate as follows: COSTA outputs the intermediate results of the analyzed program by means of extended JML annotations, and then KeY verifies the resulting proof obligations in its program logic and produces proof certificates that can be saved and reloaded. Realizing the above cooperation between COSTA and KeY, and integrating it in the Eclipse development environment, has required a number of non-trivial extensions of both systems.

Our current research is mainly focused on improving the accuracy and efficiency of the analysis. Being more accurate is especially relevant in the context of concurrency as, due to potential tasks interleavings, shared data are currently lost when the processor is released. We are working on automatically generating class invariants to improve the abstraction of share data such that we gain precision when the cost depends on the sizes of such data.

Acknowledgments. This work was funded in part by the Information & Communication Technologies program of the EC, 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 *DOVES* project, the UCM-BSCH-GR35/10-A-910502 *GPD* Research Group and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS-CM* project.

[1] `http://costa.ls.fi.upm.es/costabs`

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley (1974)
2. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers – Principles, Techniques and Tools. Addison-Wesley (1986)
3. Albert, E., Alonso, D., Arenas, P., Genaim, S., Puebla, G.: Asymptotic Resource Usage Bounds. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 294–310. Springer, Heidelberg (2009)
4. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: Cost Analysis of Concurrent OO Programs. In: Yang, H. (ed.) APLAS 2011. LNCS, vol. 7078, pp. 238–254. Springer, Heidelberg (2011)
5. Albert, E., Arenas, P., Genaim, S., Herraiz, I., Puebla, G.: Comparing Cost Functions in Resource Analysis. In: van Eekelen, M., Shkaravska, O. (eds.) FOPARA 2009. LNCS, vol. 6324, pp. 1–17. Springer, Heidelberg (2010)
6. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 221–237. Springer, Heidelberg (2008)
7. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Closed-Form Upper Bounds in Static Cost Analysis. Journal of Automated Reasoning 46(2), 161–203 (2011)
8. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost Analysis of Java Bytecode. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 157–172. Springer, Heidelberg (2007)
9. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Removing Useless Variables in Cost Analysis of Java Bytecode. In: ACM Symposium on Applied Computing (SAC) - Software Verification Track (SV 2008), Fortaleza, Brasil, pp. 368–375. ACM Press, New York (2008)
10. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost Analysis of Object-Oriented Bytecode Programs. Theoretical Computer Science 413(1), 142–159 (2012)
11. Albert, E., Arenas, P., Genaim, S., Zanardini, D.: Task-Level Analysis for a Language with Async-Finish parallelism. In: Vitek, J., De Sutter, B. (eds.) Proceedings of the ACM SIGPLAN/SIGBED 2011 Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2011, Chicago, IL, USA, April 11-14, pp. 21–30. ACM (2011)
12. Albert, E., Bubel, R., Genaim, S., Hähnle, R., Román-Díez, G.: Verified Resource Guarantees for Heap Manipulating Programs. In: Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering, FASE 2012, Tallinn, Estonia. Springer, Heidelberg (to appear, 2012)
13. Albert, E., Correas, J., Puebla, G., Román-Díez, G.: Incremental Resource Usage Analysis. In: Proceedings of the 2012 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2012, Philadelphia, Pennsylvania, USA, January 23-24. ACM Press (to appear, 2012)
14. Albert, E., Genaim, S., Gómez-Zamalloa, M.: Heap Space Analysis of Java Bytecode. In: 6th International Symposium on Memory Management (ISMM 2007), pp. 105–116. ACM Press (2007)
15. Albert, E., Genaim, S., Gómez-Zamalloa, M.: Parametric Inference of Memory Requirements for Garbage Collected Languages. In: 9th International Symposium on Memory Management (ISMM 2010), pp. 121–130. ACM Press, New York (2010)

16. Albert, E., Genaim, S., Masud, A.N.: More Precise Yet Widely Applicable Cost Analysis. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 38–53. Springer, Heidelberg (2011)
17. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: COSTABS: A Cost and Termination Analyzer for ABS. In: Proceedings of the 2012 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2012, Philadelphia, Pennsylvania, USA, January 23-24, ACM Press (to appear, 2012)
18. Albert, E., Bubel, R., Genaim, S., Hähnle, R., Puebla, G., Román-Díez, G.: Verified resource guarantees using costa and key. In: Khoo, S.-C., Siek, J.G. (eds.) PEPM, pp. 73–76. ACM (2011)
19. Braberman, V., Fernández, F., Garbervetsky, D., Yovine, S.: Parametric Prediction of Heap Memory Requirements. In: ISMM. ACM Press (2008)
20. Chin, W.-N., Nguyen, H.H., Popeea, C., Qin, S.: Analysing Memory Resource Bounds for Low-Level Programs. In: ISMM. ACM Press (2008)
21. Debray, S.K., Lin, N.W.: Cost Analysis of Logic Programs. ACM Transactions on Programming Languages and Systems 15(5), 826–875 (1993)
22. Debray, S.K., Lin, N.W.: Cost analysis of logic programs. ACM TOPLAS 15(5), 826–875 (1993)
23. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In: The 36th Symposium on Principles of Programming Languages (POPL 2009), pp. 127–139. ACM (2009)
24. Hoffmann, J., Hofmann, M.: Amortized Resource Analysis with Polynomial Potential. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 287–306. Springer, Heidelberg (2010)
25. Hofmann, M., Hoffmann, J., Aehlig, K.: Multivariate Amortized Resource Analysis. In: The 38th Symposium on Principles of Programming Languages (POPL 2011), pp. 357–370. ACM (2011)
26. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A Core Language for Abstract Behavioral Specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011)
27. Lee, J.K., Palsberg, J.: Featherweight X10: A Core Calculus for Async-Finish Parallelism. In: Principles and Practice of Parallel Programming (PPoPP 2010), pp. 25–36. ACM, New York (2010)
28. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Addison-Wesley (1996)
29. Necula, G.: Proof-Carrying Code. In: ACM Symposium on Principles of programming languages (POPL 1997). ACM Press (1997)
30. Podelski, A., Rybalchenko, A.: A Complete Method for the Synthesis of Linear Ranking Functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
31. Ramírez-Deantes, D., Correas, J., Puebla, G.: Modular Termination Analysis of Java Bytecode and Its Application to phoneME Core Libraries. In: Barbosa, L.S. (ed.) FACS 2010. LNCS, vol. 6921, pp. 218–236. Springer, Heidelberg (2010)
32. Sands, D.: Complexity Analysis for a Lazy Higher-Order Language. In: Jones, N.D. (ed.) ESOP 1990. LNCS, vol. 432, pp. 361–376. Springer, Heidelberg (1990)
33. Sands, D.: A Naïve Time Analysis and its Theory of Cost Equivalence. Journal of Logic and Computation 5(4) (1995)

34. Schäfer, J., Poetzsch-Heffter, A.: JCoBox: Generalizing Active Objects to Concurrent Components. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 275–299. Springer, Heidelberg (2010)
35. Srinivasan, S., Mycroft, A.: Kilim: Isolation-Typed Actors for Java. In: Ryan, M. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 104–128. Springer, Heidelberg (2008)
36. Unnikrishnan, L., Stoller, S.D., Liu, Y.A.: Optimized Live Heap Bound Analysis. In: Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S. (eds.) VMCAI 2003. LNCS, vol. 2575, pp. 70–85. Springer, Heidelberg (2002)
37. Wegbreit, B.: Mechanical Program Analysis. Communications of the ACM 18(9) (1975)