Automatic Inference of Bounds on Resource Consumption

Elvira Albert[1], Diego Esteban Alonso-Blas[1], Puri Arenas[1], Jesús Correas[1],
Antonio Flores-Montoya[2], Samir Genaim[1], Miguel Gómez-Zamalloa[1],
Abu Naser Masud[3], German Puebla[3], José Miguel Rojas[3],
Guillermo Román-Díez[3], and Damiano Zanardini[3]

[1] Complutense University of Madrid (UCM), Spain
[2] Technische Universität Darmstadt (TUD), Germany
[3] Technical University of Madrid (UPM), Spain

Abstract. In this tutorial paper, we overview the techniques that underlie the automatic inference of resource consumption bounds. We first explain the basic techniques on a Java-like sequential language. Then, we describe the extensions that are required to apply our method on concurrent ABS programs. Finally, we discuss some advanced issues in resource analysis, including the inference of non-cumulative resources and the treatment of shared mutable data.

1 Introduction

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 [46,45][1] 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 [42]. Since then, a number of cost analysis frameworks have been proposed, mostly in the context of *declarative* programming languages (functional programming [31,36,41,37,18] and logic programming [21,33]). 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

[1] Further information of the system is available at
`http://costa.ls.fi.upm.es/~costa/costa/costa.php`

E. Giachino et al. (Eds.): FMCO 2012, LNCS 7866, pp. 119–144, 2013.

Ciao [26] and SWI Prolog [43]) and uses the Parma Polyhedra Library [17] for manipulating linear constraints.

1.1 Organization of the Tutorial

We use the classical approach to static cost analysis which 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. Section 2 overviews this first phase which requires, among other things, the translation of the imperative program into an intermediate representation and the inference of size relations. In a second phase the cost relations are solved into a closed-form, i.e., an expression which is not in recursive form and that can be directly evaluated. Section 3.1 describes our approach to infer closed-form upper bounds on the worst-case cost and Section 3.2 the techniques to infer closed-form lower bounds on the best-case cost. Known limitations of this classical approach are described in Section 3.3, where we also compare our approach with amortized cost analysis.

Section 4 overviews the extensions needed to infer the resource consumption of ABS programs [6] which adopt the concurrent objects concurrency model. The main challenge is in handling concurrent interleavings in a sound and precise way. This requires redefining the size analysis component for the concurrent objects model. Also, the fact that concurrent objects represent distributed components brings in a new notion of cost which is not monolithic (like in traditional sequential applications) but rather it captures the cost attributed to each distributed component separately. These two issues are explained in Section 4.1. The precision of the resource analysis of concurrent languages can be improved if we infer may-happen-in-parallel (MHP) relations that over-approximate the set of program points that may execute in parallel. Section 4.2 describes the MHP analysis integrated in COSTABS [4]. COSTABS is the extension of the COSTA system to analyze ABS programs.

Section 5 discusses advanced issues in resource analysis. We start by describing the analysis of memory consumption in Section 5.1. Memory consumption is different from other type of (cumulative) resources if the language has a garbage collector. We will see that the information on which objects are garbage collected can be integrated in the analysis. As a follow-up, we will discuss in Section 5.2 the inference of the task-level of a concurrent program which tries to over-approximate the number of tasks that can be simultaneously executing in a concurrent system. The similarity with the heap consumption analysis is that both types of resources are non-cumulative. Another advanced issue that we describe in Section 5.3 is the treatment of the shared mutable data in resource analysis. This is currently one of the main challenges in static analysis of object-oriented programs. Finally, Section 5.4 overviews the design of an incremental resource analysis which, given some previous analysis results and a change in a program, is able to recompute the analysis information by reanalyzing only the components affected by the changes.

Section 6 concludes and points out directions for future work.

2 From Programs to Cost Relations

This section describes how a program is analyzed in order to produce a *cost relation system* which defines its resource consumption. The analysis consists of a number of steps: (1) the program is transformed into a *rule-based representation* which facilitates the subsequent steps of the analysis without losing information about the resource consumption; (2) size analysis and abstract compilation are used to generate size relations which describe how the size of data changes during program execution; (3) the chosen cost model is applied to each instruction in order to obtain an expression which represents its cost; (4) finally, a cost relation system is obtained by joining the information gathered in the previous steps. Let us illustrate these steps by means of an example.

Rule-Based Intermediate Representation. The input language of the programs that COSTA analyze is Java bytecode [32]. The bytecode program is first transformed into a recursive intermediate representation (RBR) [8]. The transformation starts by constructing the Control Flow Graph (CFG) of the program. Each block of the CFG is transformed into one rule in the RBR. Iteration (i.e. for/while/do loops) is transformed into recursion and conditional constructs are represented as multiple (mutually exclusive) guarded rules. Bytecode instructions for method calls are transformed into the call of the corresponding rule in RBR and recursive method calls are thus transformed into recursions. These transformations determines the recursive structure of the resulting cost relation system (CR). Each rule in the RBR program will result in an equation in the CR. 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).

Example 1. Fig. 1 shows at the top the Java source code of our running example. The Java code is shown only for clarity, since the analysis works directly on the bytecode. The example implements a sorting algorithm over an input array of integers. At the bottom of Fig. 1, the CFG and the RBR corresponding to the inner loop in the example are shown. The parameters in the rules of the RBR are tupled into input parameters corresponding to the variables on which they operate on, and the single output parameter corresponding to the return value of the rules. The CFG contains the bytecode instructions of the original input program. The entry rule to the loop is while_1. Its input arguments are the array `a` and local variables `j` and `v` and its output argument is the possibly modified array. Procedures while_2 and while_3 correspond to the two conditions of the loop and both are defined by two mutually exclusive guarded rules. The iterative structure of the loop is preserved by the recursive call to while_1 in the second while_3 rule.

Cost Models. A *cost model* $\mathcal{M}$ determines the cost (a natural number) of each basic instruction b of the language. COSTA incorporates, among others, the following cost models:

```
static void sort(int a[]) {
   for (int i = a.length-2; i>=0; i--)
     {
ⓠ     int j = i+1;
      int v = a[i];
      while ( j<a.length && a[j]<v ) {
ⓟ         a[j-1] = a[j];
          j++;
      }
      a[j-1]=v;
   }
}
```

$$
\begin{array}{lll}
\mathit{while}_1(\langle a,j,v\rangle,\langle a'\rangle) & \leftarrow & \mathit{while}_2(\langle a,j,v\rangle,\langle a'\rangle).\\
\mathit{while}_2(\langle a,j,v\rangle,\langle a\rangle) & \leftarrow & j\geq a.\mathit{length}.\\
\mathit{while}_2(\langle a,j,v\rangle,\langle a'\rangle) & \leftarrow & j<a.\mathit{length},\\
 & & \mathit{while}_3(\langle a,j,v\rangle,\langle a'\rangle.\\
\mathit{while}_3(\langle a,j,v\rangle,\langle a\rangle) & \leftarrow & a[j]\geq v.\\
\mathit{while}_3(\langle a,j,v\rangle,\langle a'\rangle) & \leftarrow & a[j]<v,\\
 & & a[j-1]{=}a[j], j{=}j+1,\\
 & & \mathit{while}_1(\langle a,j,v\rangle,\langle a'\rangle).
\end{array}
$$

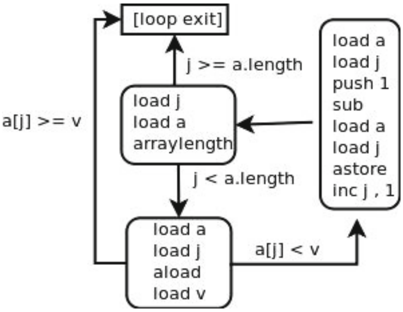

Fig. 1. Running Java example, Control Flow Graph and RBR

- *Number of instructions*: we have that $\mathcal{M}_i(b) = 1$, i.e., each bytecode instruction b in the rule-based program counts 1;
- *Number of calls to a method*: calls to methods in bytecode are of the form `invoke method_name`, thus $\mathcal{M}_c(b) = 1$ if $b \equiv$ `invoke m`; otherwise $\mathcal{M}_c(b) = 0$.
- *Heap consumption*: $\mathcal{M}_h(b) = \mathit{size}(\mathtt{C})$ if $b \equiv$ `new C`, otherwise $\mathcal{M}_c(b) = 0$, where $\mathit{size}(\mathtt{C})$ returns the amount of memory allocated in the heap when executing `new C`.

Generation of Cost Relations. Given a program P (in RBR form) and a cost model $\mathcal{M}$, we automatically generate a *cost relations system* (CR) which defines the cost of executing the program on some input $\overline{x}$ w.r.t. the selected cost model $\mathcal{M}$. CR are basically an extended form of recurrence relations. A CR is defined by a finite set of equations of the form $\langle c(\bar{x}) = e, \varphi\rangle$, where e is a cost expression and φ is a set of linear constraints which define the applicability conditions for the equations and the size relations among the variables. Variables in the equations represent the "sizes" of the corresponding data in the program according to the selected *size measure* [19]. Each program variable is abstracted using a size measure such that every non-integer value is represented as a natural number. Classical size measures used for non-integer types are: *array length* for arrays, the length of the longest reference path for linked data structures, etc. In our running example, arrays are abstracted to their length. Thus, variable a in the

CR represents the length of the array a. Note that due to this choice of size abstraction, we cannot observe the values stored in the elements of the array. All information about them is not present in the CR.

The inference of φ is defined as a *fixpoint computation* which comprises two steps: First, *abstract compilation* of the program replaces bytecode instructions in the RBR by size constraints. Then, *size analysis* infers *size-relations* between the states at different program points, i.e., it approximates how the sizes of variables change from one call in the cost relation to another. Analysis is often done by obtaining an abstract version of the program by relying on abstract interpretation [20]. In Fig. 2, such size relations are shown to the right of the equations.

Example 2. Let us assume that $\textcircled{q}$ (resp. $\textcircled{p}$) represents the cost of executing the body of the `for` excluding the cost of the inner loop (resp. the cost of executing the body of the `while` loop). Here, $\textcircled{q}$ and $\textcircled{p}$ are symbolic cost expressions (or constants) and we assume this for simplicity at this stage. Fig. 2 shows the resulting CR for the running example of Fig 1. Here, variables are constraint variables corresponding to those of the original rule, e.g. i and i' both correspond to values of variable `i`, but at different program points. Instructions are replaced by linear constraints. Array `a` is abstracted to its length a. Thus, the first rules for while_1 become non-deterministic, as it is not possible to observe the array elements. Output variables are removed by inferring input-output size relations.

$$
\begin{array}{rll}
\mathit{sort}(a) & = \mathit{for}(a,i) & \{i{=}a{-}2,\ a{\geq}0\} \\[4pt]
\mathit{for}(a,i) & = 0 & \{i{<}0\} \\
\mathit{for}(a,i) & = \textcircled{q}{+}\mathit{while}_1(a,j){+}\mathit{for}(a',i') & \{i{\geq}0, j{=}i{+}1, i'{=}i{-}1, a'{=}a\} \\[8pt]
\mathit{while}_1(a,j) & = 0 & \{j{\geq}a\} \\
\mathit{while}_1(a,j) & = 0 & \{j{<}a\} \\
\mathit{while}_1(a,j) & = \textcircled{p}{+}\mathit{while}_1(a',j') & \{j{<}a, j'{=}j{+}1, a'{=}a\}
\end{array}
$$

Fig. 2. CR from example in Fig 1

In practice, the process of generating cost relations additionally involves several other static analysis techniques. In particular, *class analysis* is performed to compute the reachable code; *nullity* and *array bound* analysis for dead code elimination, *slicing* to eliminate variables which are irrelevant to cost and *cyclicity* analysis to identify cyclic data structures.

3 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, namely: (1) in order to obtain a concrete bound, for a given input, we still need to evaluate the CRs, and moreover, due to the nondeterministic nature of CRs, such evaluation must consider all possibilities and select the worst/best among them; and (2) CRs do not provide a clear insight on the complexity class to which the resource consumption belongs, e.g., polynomial, exponential, etc. For these reasons, *closed-form* bounds, which are functions composed from simple arithmetic expressions, are preferable. This form makes it possible to address the above two issues efficiently. COSTA includes a solver called PUBS, that is encharged of solving CRs into closed-form upper and lower bounds.

Solving CRs into closed-form (lower or upper) bounds in PUBS is done in two phases. In the first phase, the CRs are simplified such that all recursions are direct (i.e. cycles in the call graph are of length one), which is achieved by applying *partial evaluation* [29] in order to unfold intermediate equations. After this step, each iterative or recursive construct in the original program is represented by a single directly recursive CR. The CRs of Fig. 2 are given in this form. In the second phase, the CRs are solved into closed-form bounds in a compositional way, by handling one CR at a time, in a reversed order to the calling relation. E.g., for the CRs of Fig. 2 the solving proceeds as follows: it solves CR while_1; substitutes the resulting bound in the CR *for*; solves CR *for*; substitutes the resulting bound in CR *sort*; and finally solves CR *sort*. These two phases are common for inferring both upper and lower bounds, the difference is in how each CR (that does not call any other CRs, i.e., standalone), in the second phase above, is solved. This is discussed in sections 3.1 and 3.2. Sec. 3.3 discusses programs whose cost cannot be modeled precisely with CRs, and explain a corresponding solution.

Note that a common feature to all solving methods, that we describe in this section, is that they heavily rely on the use of program analysis techniques. This, we believe, is the most important factor that made COSTA succeed where other previous cost analyses had failed.

3.1 Upper Bounds

PUBS includes two approaches for solving CRs into closed-form upper-bounds. They have different applicability and precision properties. In what follows we explain the essentials of both approaches.

We start by intuitively explaining the first approach using the CRs of Fig. 2, starting with the standalone CR while_1. Let a_0 and j_0 be unknown initial values, for a and j respectively, with which while_1 is called. Solving this CR is done by inferring an upper-bound $\hat{f}(a_0, j_0)$ on the number of times that the recursive equation can be applied, when evaluating $\mathit{while}_1(a_0, j_0)$, then, $\hat{f}(a_0, j_0) * \textcircled{p}$ is guaranteed to be an upper-bound for $\mathit{while}_1(a_0, j_0)$ since all applications of the recursive equation contribute the constant symbol $\textcircled{p}$. Inferring $\hat{f}(a_0, j_0)$ automatically can be done by relying on techniques from the field of termination analysis, such as *synthesis of ranking functions* [34]. For the case of while_1, we automatically infer $\hat{f}(a_0, j_0) = \mathtt{nat}(a_0 - j_0)$, where $\mathtt{nat}(v) = \max(v, 0)$, and thus,

$\texttt{nat}(a_0 - j_0) * \textcircled{p}$ is an upper-bound for $\mathit{while}_1(a_0, j_0)$. The $\texttt{nat}$ function is used to lift negative values to zero because negative evaluations of $a_0 - j_0$ means that the number of times the recursive equation of while_1 applied in an evaluation is zero.

Next we proceed to solve the CR *for*. First we substitute the upper-bound of while_1 in the CR of *for*, which converts it into standalone

$$\begin{array}{ll} \mathit{for}(a,i) = 0 & \{i{<}0\} \\ \mathit{for}(a,i) = \textcircled{q} + \texttt{nat}(a-j) * \textcircled{p} + \mathit{for}(a',i') & \{i{\geq}0, j{=}i{+}1, i'{=}i{-}1, a'{=}a\} \end{array}$$

In this case, $\hat{f}(a_0, i_0) = \texttt{nat}(i_0 - 1)$ is an upper-bound on the number of times that the recursive equation can be applied, when evaluating $\mathit{for}(a_0, i_0)$. Assuming that we have a function $\hat{e}(a_0, i_0)$ such that it is guaranteed to be bigger than any instance of $\textcircled{q} + \texttt{nat}(a-j) * \textcircled{p}$, then $\hat{f}(a_0, i_0) * \hat{e}(a_0, i_0)$ is an upper-bound for $\mathit{for}(a_0, i_0)$. Let us explain how to automatically compute $\hat{e}(a_0, i_0)$. Since $\textcircled{q} + \texttt{nat}(a-j) * \textcircled{p}$ takes its maximum values when $a - j$ is maximal, it is enough to compute an upper-bound on $a-j$ (in terms of a_0 and j_0). This can be done using invariant generation and linear programming as follows: (1) we infer an invariant Ψ that relates the initial values a_0 and i_0 to the values of a and i at any call $\mathit{for}(a, i)$, which is $\Psi = \{a{=}a_0, i_0{\geq}i\}$ in this case; and (2) the maximum value to which $a-j$ can be evaluated is obtained applying the recursive equation in the context of Ψ, and asking what is the maximum of $a-j$ for this application. This is equivalent to solving the following parametric integer programming [22] problem:

$$\begin{array}{l} \texttt{maximize}\ a{-}j\ \texttt{w.r.t} \\ \qquad \{a{=}a_0, i_0{\geq}i\} \wedge \{i{\geq}0, j{=}i{+}1, i'{=}i{-}1, a'{=}a\} \text{ and the } \texttt{parameters}\ a_0, i_0 \end{array}$$

which results in $a_0 - 1$. Then, $\hat{e}(a_0, i_0) = \textcircled{q} + \texttt{nat}(a_0 - 1) * \textcircled{p}$, and thus $\texttt{nat}(i_0 + 1) * (\textcircled{q} + \texttt{nat}(a_0 - 1) * \textcircled{p})$ is an upper-bound for CR *for*. An upper-bound $\texttt{nat}(a_0 - 1) * (\textcircled{q} + \texttt{nat}(a_0 - 1) * \textcircled{p})$ for CR *sort* is then obtained by substituting the one of *for* in the corresponding equation. Note that this approach is general enough to handle the CRs which is constructed with possible multiple equations having possible multiple recursive calls. E.g, if CR while_1 had two recursive calls, then it would obtain the upper-bound $2^{\texttt{nat}(a_0 - j_0)} * \textcircled{p}$ – For more details see [6]. Note also that PUBS provides a mechanism for converting the above non-asymptotic bounds to asymptotic ones [2], e.g., for the case of *sort* it computes $O(\texttt{nat}(a_0)^2 * \textcircled{p} + \texttt{nat}(a_0) * \textcircled{q})$.

In the above approach, the contributions of all applications of the recursive equation are approximated by the same amount. E.g., in the case of CR *for*, all instances of $\textcircled{q} + \texttt{nat}(a-j) * \textcircled{p}$ are approximated by $\textcircled{q} + \texttt{nat}(a_0 - 1) * \textcircled{p}$, while in practice this happens only in the last application, when $j = 1$ (i.e., when $i = 0$ since $j = i + 1$). This leads to some imprecision that might be crucial for some applications. To overcome this imprecision, PUBS provides an alternative approach that is based on simulating the contributions of the different applications using sequences (arithmetic, geometric, etc.). E.g., in the case of CR

$$\begin{array}{ll}
\mathit{while}_1(a,j,\lambda) = 0 & \{j \geq a\} \\
\mathit{while}_1(a,j,\lambda) = 0 & \{j < a\} \\
\mathit{while}_1(a,j,\lambda) = \textcircled{p} + \mathit{while}_1(a',j',\lambda') & \{j<a, j'=j+1, a'=a, \lambda' = \lambda+1\}
\end{array}$$

Fig. 3. CR after instrumenting counter variables in the CRs in Fig 2

for, it first infers that the difference between the contributions of two consecutive application is at least $\hat{d} = 1$, and then it considers the arithmetic sequence $\hat{u}_1 = \hat{e}(a_0, i_0)$, $\hat{u}_i = \hat{u}_{i-1} - \hat{d}$, and sum the first $\hat{f}(a_0, i_0)$ elements (which are positive) of this sequence. This sum is guaranteed to be an upper-bound for $\mathit{for}(a_0, i_0)$ since the equation is applied at most $\hat{f}(a_0, i_0)$ times, and moreover the sequence starts from the maximum value $\hat{e}(a_0, i_0)$. For the case of CR *for* we obtain $\frac{1}{2} * \textcircled{p} * (\mathtt{nat}(a_0 - 1) * (\mathtt{nat}(a_0 - 1) - 1)) + \textcircled{q} * \mathtt{nat}(a_0 - 1)$ which is more precise than what we have obtained above. Note, however, that they are asymptotically equivalent. In practice, the summation is computed by solving a corresponding recurrence relation using a computer algebra system – For more details see [14].

3.2 Lower Bounds

In the latter approach [14], 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 obtaining the best-case cost of each iteration. LB on the number of iterations can be inferred by instrumenting the given CR and inferring an invariant on that CR. First, the arguments of each head of the given CR are augmented with a new counter variable λ that is incremented by 1 in each recursive call of that CR. Next, an invariant Ψ is inferred for this new CR such that Ψ holds between an initial call with 0 as the initial counter value and any other call to the new CR with counter variable λ. Then, the LB on the number of iterations is obtained by minimizing λ w.r.t Ψ and φ_0, where φ_0 is the set of constraints in the base-case of the new CR. Minimization of λ can be done using parametric integer programming or by looking syntactically $\lambda \geq l$ in $\Psi \wedge \varphi_0$ where l is over the initial arguments.

Example 3. Let us consider the CR *while₁* in Fig. 2. We now instrument it with the counter variable λ as shown in Fig. 3. The invariant Ψ between an initial call $\mathit{while}_1(a_0, j_0, 0)$ and another call $\mathit{while}_1(a, j, \lambda)$ is $\Psi \equiv \{j \geq j_0, a = a_0, \lambda = j - j_0\}$. λ is minimized to 0 w.r.t Ψ and the base-case constraints $(j \geq a \vee j < a)$. The LB on the iterations of *for* obtained is $\mathit{nat}(i_0 + 1)$ where i_0 is the initial value of i.

The best-case cost in each iteration is obtained by transforming the given CR into a best-case recurrence relation (RR for sort). Suppose $\langle C(\bar{x}) = e + C(\bar{x}'), \varphi\rangle$ be any CR and $e_1, \cdots, e_n$ be the costs contributed by e along the n iterations of $C(\bar{x})$. In order to obtain the LB cost for C, first, a LB $\check{n}$ on n (i.e. $\check{n} \leq n$)

is obtained (as above). Then, a series of costs $u_1, \cdots, u_{\check{n}}$ such that $u_i \leq e_i$ for all $1 \leq i \leq \check{n}$ are obtained, and then $u_1 + \cdots + u_{\check{n}}$ is the LB of C. When e is a simple linear expression, the novel idea is to view $u_1, \cdots, u_{\check{n}}$ as an arithmetic sequence that starts from $u_1 \equiv \check{e}$ and each time increases by $\check{d}$, i.e., $u_i = u_{i-1} + \check{d}$ where $\check{e}$ is the minimization of e, and $\check{d}$ is an under-approximation of all $d_i = e_{i+1} - e_i$. Minimization of e, i.e. $\check{e}$, can be obtained by using parametric integer programming w.r.t an appropriate invariant. When e is a complex non-linear expression, e.g., $l * l'$, it can be approximated by approximating each sub-expressions (which are linear) separately. Technically, the summation $u_1 + \cdots + u_{\check{n}}$ can be approximated by transforming the CR into a RR whose closed-form solution is the LB cost after instantiating the recurrence counter by $\check{n}$.

Example 4. Let us consider again the CR in Fig. 2. The LB cost of *while₁* is 0 since the LB on the iterations of *while₁* is 0 (see example 3). After substituting the cost of *while₁*, the recursive equation for CR *for* is $\langle \mathit{for}(a, i) = \textcircled{q} + \mathit{for}(a', i'), \{i \geq 0, j = i + 1, i' = i - 1, a' = a\}\rangle$. Here, $\check{d} = 0$ as cost $\textcircled{q}$ is constant and $\check{n} = \mathit{nat}(i_0 + 1)$. Next, the RR of *for* is $\langle P_{\mathit{for}}(0) = 0,\ P_{\mathit{for}}(N) = \textcircled{q} + P_{\mathit{for}}(N - 1)\rangle$ whose closed-form solution is $E = \textcircled{q} * N$, and LB cost is $\textcircled{q} * \mathit{nat}(i_0 + 1)$ (after replacing N by $\mathit{nat}(i_0 + 1)$ in E). Finally, the LB cost of $\mathit{sort}(a_0)$ is $\textcircled{q} * \mathit{nat}(a_0 - 1)$. This is the LB that we have expected, since when the array is sorted, the inner loop does not perform any iteration and the best-case cost is linear on the length of the array.

3.3 Amortised Cost Analysis

The classical approach of COSTA is based on assuming that the cost of a procedure is solely determined by the size of its *input* data. In some procedures, there is also a codependency between the outputs and the cost, which may be crucial to infer precise cost bounds. Yet, since CRS do not model this codependency, for such programs the COSTA approach necessarily infers imprecise bounds.

Example 5. Consider the program of Fig. 4 (left), adapted from [40], where $*$ in the guard of the *while* loop corresponds to a nondeterministic evaluation of *true* or *false*. This nondeterministic choice is reflected in the constraints of equation $\mathit{rpop}(s) = 0$ in Fig. 4 (right) as the while loop can terminate for any value of $s \geq 0$. The procedure `main` admits the UB $\textcircled{r} * s$, but COSTA gets the asymptotically imprecise UB $\textcircled{r} * s * m$ instead. The reason is that the nondeterministic procedure *rpop* sets up a codependency between its cost and s', its return value: a possible execution of `rpop(s)` consumes $\textcircled{r} * s$ and returns $s' = 0$; another one returns $s' = s$ and consumes zero; but no one both consumes $\textcircled{r} * s$ and also returns $s' = s$. COSTA abstracts the program into the CRS at Fig. 4 (right up), solves $\mathit{rpop}(s)$ into the precise bound $s * \textcircled{r}$ and unfolds this bound in the *main* CRS (right down). Although both this UB and the postcondition $s \geq s' \geq 0$ are precise abstractions w.r.t. `rpop`, they miss the described output-cost codependency. Thus, the CRS semantics now includes the spurious case of an execution of `rpop` consuming $\textcircled{r} * s$ and returning $s' = s$. For this reason, the

```
int rpop(int s){
    while(s > 0 && * )
        s-- ;
    return s ;
}
void main(int s,int m){
    for(;m>0;m--)
        s = rpop(s);
}
```

$$\begin{array}{rll}
rpop(s) = & 0 & \{s \geq 0\} \\
rpop(s) = & \textcircled{r} + rpop(s-1) & \{s \geq 1\} \\
main(s,m) = & 0 & \{m = 0, s \geq 0\} \\
main(s,m) = & \begin{array}{l} rpop(s)+ \\ main(s', m-1) \end{array} & \left\{\begin{array}{l} m \geq 1 \\ s \geq s' \geq 0 \end{array}\right\} \\
\hline
main(s,m) = & 0 & \{m = 0, s \geq 0\} \\
main(s,m) = & \begin{array}{l} \textcircled{r} * s+ \\ main(s', m-1) \end{array} & \left\{\begin{array}{l} m \geq 1 \\ s \geq s' \geq 0 \end{array}\right\}
\end{array}$$

Fig. 4. Example of Amortised cost, with the Java program (left), the inferred CRS (right up) and the CRS unfolding the UB for *rpop* (right down)

CRS does not admit the bound $\textcircled{r}{*}s$ that we look for, and the techniques of Section 3.1 give $\textcircled{r}{*}s{*}m$ as the most precise UB for the *main* CRS.

Examples like this usually appear in the context of *amortised cost analysis* [40]. There, the output-cost codependency is described like the variable s storing *credit* or *potential* to pay the decrement operations. To overcome these limitations, we have recently developed [16] a novel definition of UBs that involve input and output arguments: a net-cost UB $\widetilde{rpop}(s|s')$ bounds the cost of any terminating evaluation of *rpop* from an input s to an output s'. By making s' an input of the UB, net-cost UBs capture the output-cost codependency. In [16] we also describe a solving procedure based on real quantifier elimination, and draw a relation between net-cost functions and the potential functions used in the *automated amortised approach* [27,30,35].

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 [38,39]. 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. 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.

```
def B look⟨A,B⟩(Map⟨A,B⟩ ms, A k) =
  case ms {Ass(Pair(k,y),_) ⇒ y;
           Ass(_,tm) ⇒ look(tm,k);
}
class AddrBook {
  Map⟨String,User⟩ users = EmptyM;
  User getUser(String email){
    return look(users,email);
  }
}
class User {
  List⟨String⟩ msgs = Nil;
  Unit receive(String m) {
    msgs = Cons(m,msgs);
  }
}
```

```
class MailServer(AddrBook ab) {
  List⟨String⟩ emails =Nil;
  Unit addUser(String email) {
    emails = Cons(email, emails);
  }
  Unit addUsers(List⟨String⟩ l) {
    while ( l != Nil ) {
      this ! addUser(head(l));
      l=tail(l);
    }
  }
  Unit notify(String m) {
    while (emails != Nil) {
      Fut⟨User⟩ u;
      u = ab ! getUser(head(emails));
      await u ? ;
      User us = u.get;
      us ! receive(m);
      emails = tail(emails);
    }
  }
}
```

Fig. 5. ABS Implementation of a Mail Server

4.1 The Basic Cost Analysis Framework for Concurrency

ABS [28] is an *a*bstract *b*ehavioral *s*pecification language for distributed object-oriented systems. COSTA has been recently extended to be able to infer meaningful bounds for ABS programs [3]. The main novelties are related to the concurrency and distribution aspects of the language. *Concurrency* poses new challenges to the process of obtaining sound and precise size relations. This is mainly because the interleaving behaviour inherent to concurrent computations can influence how the sizes of data are modified. *Distribution* does not match well with the traditional monolithic notion of cost which aggregates the cost of all distributed components together. We use *cost centers* to keep the resource consumption of the different distributed components separate. An implementation of this cost analysis framework is described in [4]. The system is open-source and can be downloaded (together with examples, documentation, etc.) from: `http://costa.ls.fi.upm.es/costabs`.

Example 6. The example in Fig. 5 shows a simple mail server application programmed in ABS. Due to lack of space, we omit data and type definitions. At the top, we see a fragment of the functional subprogram which includes the function *look*. The imperative concurrent part contains the implementation of all classes. Calls to functions and functional data structures appear in italics. A mail server is composed of an address book (the class parameter ab) and a list of email addresses (the field emails). Email addresses can be added to the server by invoking method addUser and addUsers. Method notify sends a message (m) to all users in the list emails. To this end, it first asynchronously invokes getUser in order to retrieve the next user (variable u) in the list. The await instruction allows releasing the processor if the information is not ready. The next instruction get blocks the execution of the current task until the requested information has arrived. When it arrives, the asynchronous call to receive is encharged of sending the message to the corresponding user without any kind of synchronization.

In what follows, we explain briefly the differences w.r.t. cost analysis in a sequential setting on the running example:

Cost Models for Concurrency. We consider the cost models *steps*, *memory*, *objects* and *task-level*. The first two ones are inherited from the sequential setting (see Section 2), while the last two ones are specific for concurrency. The "objects" cost model counts the total number of objects created along the execution. This provides an indication of the amount of parallelism that might be achieved, since each object could be running in a different processor. The "task-level" cost model estimates the number of tasks that are spawned along an execution. This can be counted by tracing how many asynchronous calls are performed. The task-level is useful for finding optimal deployment configurations, and detect situations like when one component is receiving too many requests while its siblings are idle.

Size Analysis. In order to handle the concurrency primitives, the classical sequential size analysis described in Section 2 is modified as follows: (a) when executing an instruction which does not cause the suspension of the current task, then fields (i.e., the global state) are tracked as if they were local variables, since in the concurrent objects setting it is guaranteed that in such circumstances no other tasks can modify those fields simultaneously; and (b) when executing an instruction that might cause suspension (e.g., `await`) of the current task, then the analysis loses all information about the corresponding fields, this is because they might be modified by other tasks in the meantime. This simple modification guarantees soundness of size analysis for a concurrent setting. However, it often loses precision. For example, in the while loop of method notify, losing the information on the field emails when executing `await` prevents us from proving that its size decreases in each iteration. Thus, the technique fails to bound the number of iterations of that loop. To overcome this problem, we provide a way to incorporate class invariants. For example, if we add the following invariant (using JML syntax) `//@invariant \old(emails) == emails` before the `await` instruction in the while loop of method notify, then we state that it is guaranteed that when the process resumes, the value of emails will be the same as when the process has been suspended. At present, we can infer class invariants automatically in a limited manner (See [3] for details). For example, if a variable (or shared location) is initialized and is never updated afterwards, we can infer that the values of this variable before and after a release point are always equal.

Cost Centers. The last step in this framework uses the inferred size relations and the selected cost model in order to generate cost equations and solve them into closed-form bounds. See Section 2 for more details. Now, let us explain the upper bounds that we obtain for the running example.

By applying the analysis starting from method notify and using the *steps* cost model, we obtain the following upper bound (after simplifying the constants for the sake of readability): $5 + (22 + 4 * users^+) * emails^+$. Variables $emails^+$ and $users^+$ refer to the maximum sizes of the fields emails and users respectively. The subexpression $(22 + 4 * users^+)$ refers to the cost of each iteration of the while loop. Note that the subexpression $4 * users^+$ refers to the cost consumed by function *look*. The constant 4 is for executing the code of *look* once, and $users^+$ is the number of recursive calls. The cost of each iteration is then multiplied by $emails^+$, which is a bound on the number of iterations of the while loop. Finally, we add 5 to account for the cost of the instructions outside the loop (in this case it refers to the last comparison of the while loop's guard).

Instead of computing a monolithic cost expression, there exists the option of splitting the cost into *Cost Centers* that represent the different distributed components of the system. By assuming that objects of the same type belong to the same cost center, we obtain the following upper bounds (after simplification of constants for the sake of readability):

Cost Center	Upper Bound
MailServer	$5 + 16 * emails^+$
User	$3 * emails^+$
AddrBook	$(3 + 4 * users^+) * emails^+$

Observe that the sum of these bounds is identical to the single bound we have obtained before.

4.2 MHP

In the previous section, an invariant was used to ensure that the list of emails was not modified during the await. However, in order for the analysis to be safe, this invariant must be proven correct. An invariant in an `await` instruction expresses properties of the object fields that are maintained during the execution of the `await`. Therefore, the validity of these invariants depends on the actual changes that can take place while the current task is suspended.

A first step towards verifying these invariants consists on approximating the instructions that can be executed at that point. In our example, the invariant `//@invariant \old(emails) == emails` will hold if the instructions that can be executed during the `await` do not modify the field *emails*. For that purpose, a may-happen-in-parallel analysis has been developed [11,12].

To illustrate the behavior of the MHP analysis, we complete our example code with the main block in Figure 6 that defines the entry point of the program. The main block implements the following usage scenario: (a) it creates several `User` objects, each with a unique email address; (b) it creates an `AddrBook` object, and passes to it a list of pairs (name,user), `[p1,...]`; (c) it creates a `Notifier` object which receives the address book `ab` as class parameter; (d) it adds some email addresses to be notified by asynchronously calling `addUsers`, and waits until it has terminated; and (e) finally it calls method `notify` in order to notify all registered users with a given message.

```
User u1=new UserImp();
Pair<String,User> p1 =Pair("John",u1);
...
AddrBook ab=new AddrBook(map[p1, p2, p3]);
MailServer ms =new MailServer(ab);
Fut<Unit> x =ms!addUsers(list["Alice","Bob"]);
await x?;
ms!notify("Hello␣Alice␣and␣Bob");
```

Fig. 6. Usage scenario: Main method

First, the MHP analysis generates a MHP graph that captures all MHP relations between the different program points of the program. Then, using this graph, it infers the set of MHP pairs of the form (i,j) which indicates that the instruction at program point i might execute in parallel with the one at program point j, and vice versa.

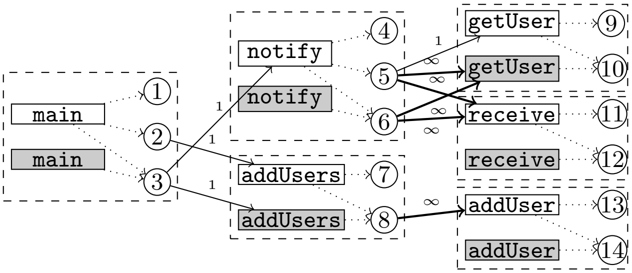

Fig. 7. MHP graph

MHP Graphs. The MHP graph corresponding to our current example is depicted in Fig. 7. Each program point i that corresponds to a context switch, i.e., a program point in which the execution might switch from one method to another, is represented by a node ⓘ. These nodes always include the method's *entry* and *exit* program points. In principle, other program points can be included, however, these are the only ones required for soundness. Each method `m` contributes two nodes: $\boxed{\texttt{m}}$ represents an instance of `m` that is *active*, i.e., running and can be at any program point, and $\fcolorbox{black}{lightgray}{\texttt{m}}$ represents an instance of `m` that is *finished*, i.e., it is at the exit program point.

The MHP graph is composed of 6 subgraphs, one for each method and that are represented as dashed rectangles. In each subgraph: (a) the *active* method node (the white rectangle) is connected to all program point nodes of that method, meaning that when the method is active it can be executing at any of those program points; and (b) the *finished* method node (the gray rectangle) is connected to the exit program point node, meaning that when the method is finished it must be at the exit program point. For example, in the subgraph of method `main`, there are edges from $\boxed{\texttt{main}}$ to nodes ①, ②, and ③; and from $\fcolorbox{black}{lightgray}{\texttt{main}}$ to ③.

The subgraphs are interconnected by weighted edges. Each such edge starts at a program point node in one subgraph, and ends in an active or finished method node in another subgraph (it can be the same if the method is recursive). These edges are inferred by applying a *method-level* MHP analysis which analyzes each method separately. This analysis infers, for each program point, which methods might be running in parallel with that program point, how many instances of each, and in which mode (active or finished). This information is inferred by considering only the code of the corresponding method. For example, the method-level analysis infers: (a) for method `main`, at 2 (that corresponds to the *await* instruction), there might be one active instance of method `addUsers`. This will add an edge from ② to $\boxed{\texttt{addUsers}}$. The edge is labeled with 1 to indicate that it is only one instance of `addUsers`; and (b) for method `notify`, at 5(the *await* instruction), there might be an active instance of `getUser`, many finished instances of `getUser`, and many active instances of `receive`. This will add an edge from ⑤ to $\boxed{\texttt{getUser}}$ with label 1, to $\fcolorbox{black}{lightgray}{\texttt{getUser}}$ with label ∞, and to $\boxed{\texttt{receive}}$ with label ∞. Edges with ∞ should be interpreted as infinitely many edges with weight 1.

MHP Property. The MHP graph guarantees that if there is an execution in which the instructions at program points i and j might execute in parallel, at least one of the following holds:

- *direct relation*: there is a path from ⓘ to ⓙ (or vice versa); or
- *indirect relation*: there is a node ⓚ that has two *different* paths to both ⓘ to ⓙ.

These properties are the base of the MHP inference. We can see that there is a path from ② to ⑬ which induces the direct MHP pair (2 ,13). Also, there are different paths from ③ to both ⑬ and ⑤ which induces the indirect MHP pair (5,13). Given this pair we cannot verify our invariant. In fact, we just detected a synchronization error. At 5(where our invariant is placed) the *emails* list can be modified by the method `addUser`.

The MHP analysis can also be used to spot synchronization errors, find the causes of those errors (debugging), and acquire a better understanding of the program concurrent behavior (program understanding) –See [11,12].

5 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.

5.1 Memory Consumption Analysis

Predicting the dynamic memory (heap) required to run a program is crucial in many contexts such as 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. On the other hand, garbage collection (GC) is a very powerful and useful mechanism which is increasingly used in high-level languages such as Java. Unfortunately, GC makes it difficult to predict the amount of memory required to run a program. A first approximation to this problem is to simply ignore the GC and infer bounds on the *total heap consumption*, i.e., the *accumulated* amount of memory dynamically allocated by a program. This can be done directly applying the COSTA framework using the $\mathcal{M}_h$ cost model defined in Section 2. If such amount of memory is available it is ensured that the program can be executed without exhausting the memory, even if no GC is performed during its execution. However, this is an overly pessimistic estimation of the actual heap consumption since, in the presence of GC, the memory usage increases and decreases along the execution.

COSTA incorporates a novel *peak heap space analysis* [13], also known as *live heap space analysis*, which aims at approximating the maximum size of the data on the heap during a program's execution, which provides a much tighter estimation. Whereas analyzing the total heap consumption requires to observe the consumption at the *final* state only, peak heap consumption analysis has to reason on the heap consumption at *all program states* along the execution. As a consequence, the basic COSTA framework cannot be directly applied.

When considering GC, several techniques exist which differ on:

(1) *what* can be collected, i.e., the *lifetime* of objects;
(2) *when* GC is performed.

As regards (1), a *GC strategy* classifies objects in the heap into two categories: those which are collectible and those which are not. Most types of garbage collectors determine *unreachable* objects as collectible, i.e., they eliminate those objects to which there is no variable in the program environment pointing directly or indirectly. The more precise alternative is to rely on the notion of *liveness*. An object is said to be *not live* (or *dead*) at some state if it is not used from that point on during the execution.

As regards (2), we consider several possibilities. One is *scope-based* GC in which deallocation of unreachable objects takes place on methods' return, and, only those objects created during the method's execution can be freed. Another possibility is the so-called *ideal* GC in which objects are collected as soon as they become collectible. The third one assumes a given limit on the heap, and applies GC only when we are about to exceed this limit.

```
void m₁() {
  A a=new A();①
  a.f=new B();②
  a=m₂(a);④
  D d=new D();
}
A m₂(A a) {
  C c=new C();
  int i=a.f.data+c.data
  a.f = null;③
  return new E(i);
}
```

```
m₁(⟨⟩,⟨⟩)  ←
    a:=new A,①
    a.f:=new B,②
    m₂(⟨a⟩, ⟨a⟩),④
    d:=new D.

m₂(⟨a⟩,⟨r⟩)  ←
    c:=new C,
    i:=a.f.data+c.data,
    a.f:=null,③
    r:=new E.
    init_E(⟨r, i⟩, ⟨⟩).
```

$$\begin{aligned}
\mathsf{T} &= \mathrm{s(A)} + \mathrm{s(B)} + \mathrm{s(C)} + \mathrm{s(D)} + \mathrm{s(E)}\\
\mathsf{S} &= \mathrm{s(A)} + \mathrm{s(B)} + \mathrm{s(E)} + \max(\mathrm{s(C)}, \mathrm{s(D)})\\
\mathsf{R} &= \max(\mathrm{s(A)} + \mathrm{s(B)} + \mathrm{s(C)}, \mathrm{s(A)} + \mathrm{s(C)} + \mathrm{s(E)}, \mathrm{s(E)} + \mathrm{s(D)})\\
\mathsf{L} &= \max(\mathrm{s(A)} + \mathrm{s(B)} + \mathrm{s(C)}, \mathrm{s(E)}, \mathrm{s(D)})
\end{aligned}$$

Fig. 8. A Java Program and its memory requirements: T=total-allocation; S=scope-based; R=reachability-based; L=liveness-based.

COSTA offers a general framework to infer accurate bounds on the peak heap consumption of bytecode programs which improves the state-of-the-art in that:

- it is not restricted to any complexity class and deals with all bytecode language features including recursion,
- it is parametric w.r.t the *lifetime* of objects and,
- it can be instantiated with different GC strategies, e.g., the scope-based and ideal GC discussed above.

Example 7. Let us consider the Java program in Figure 8 (to the left). To the right, we show the RBR. Because the program has simple (constant) memory consumption, it is useful to describe intuitively the differences among the different approximations to memory consumption. In Figure 8 (to the bottom) we provide four possible approximations inferred by our analysis for the memory consumption of executing method m_1, where the notation s(X) means the memory required to hold an instance of class X.

First, we consider a scope-based GC in which objects lifetimes are inferred by an *escape* analysis. In this case, we can take advantage of the knowledge that at ④ (i.e., upon exit from m_2) the object to which "c" refers can be freed, i.e., it does not *escape* from the method. Hence, the UB S is obtained. The important point is that s(A) and s(B) are always accumulated, plus the largest of the consumption of m_2 (i.e., s(C) + s(E)) and the memory *escaped* from m_2 (i.e., s(E)) plus the continuation (i.e., s(D)).

As another instance, we consider a reachability-based GC but without the assumption of being scope-based, rather we assume an ideal GC. Then, our method is able to obtain the UB R in Fig 8. This is due to the fact that the object to which "a.f" points becomes unreachable at program point ③, the object to which "c" points becomes unreachable upon exit from m_2, and the object created immediately before ① becomes unreachable at ④. We can observe that this information is reflected in R by taking the maximum between: the consumption up to the first allocation instruction in m_2; the consumption up to the end of m_2 taking into account that the object to which "a.f" points becomes unreachable, plus the consumption until the end of m_1 taking into account that both the object pointed by "a.f" and the object created immediately before ① become unreachable.

As the third instance, we consider the combination of an ideal garbage collector based on liveness, i.e., objects are reclaimed as soon as they become dead (i.e, will not be used in the future). Then, we obtain the UB L by taking advantage of the fact that the object created immediately before ① and those to which "a.f" and "c" point are dead at program point ③, and that the object created at the end of m_2 is dead at program point ④. This information is reflected in the elements of the max similarly to what we have seen for R. Note that, in theory, the peak heap consumption L is indeed the minimal memory requirement for executing the method.

5.2 Inference of Task-Level in Concurrent Languages

Another type of non-cumulative resource is the task-level. In parallel languages, we refer to a *task* as the unit of parallelism in a program execution, i.e., a sequential computation which can be executed in parallel and communicate with a number of other computations going on at the same time. The *task level* of a program is the maximum number of tasks that can be *available* (i.e., not finished nor suspended) simultaneously during its execution, regardless of the input data (e.g., considering all possible inputs). Knowing statically the task level of a program is of utmost importance for program understanding, debugging, and task scheduling.

COSTA includes a component which estimates the task level of parallel programs written in a subset of the X10 programming language. X10 features *async-finish parallelism*, where `async` and `finish` are basic constructs for, respectively, spawning a new task and waiting until some tasks terminate. In particular `finish` waits until all tasks spawned within the block that it delimits finish. Given a parallel program, our analysis [9] returns a *task-level upper bound*, i.e., a function on the input arguments that guarantees that the task level of the program will never exceed its value along any execution.

Example 8. The following recursive program implements the *merge-sort* algorithm working on a global array, and shows how `async-finish` parallelism works:

```
void msort(int from, int to) {
  if (from<to) {
    mid=(from+to)/2;
```

```
   finish {
    async msort(from,mid);
    async msort(mid+1,to);
   }
   merge(from,to,mid);
  } }
```

`msort` recursively calls itself twice inside `async`: this means that the tasks `msort (from,mid)` and `msort(mid+1,to)` are spawned asynchronously and can execute independently. However, calling them inside `finish` means that the main procedure cannot continue to `merge(from,to,mid)` until both tasks have finished.

In the above example, the *total number of tasks* (i.e., all the tasks which may be spawned along the execution) spawned by a call to `msort(from,to)` is bounded by 2(`to`−`from`+1)−2. Moreover, it can also be inferred that the *peak of live tasks* (i.e., the maximum number of tasks which can be alive at the same time) is bounded by the same expression. In general, the peak of live tasks is smaller than the total number since it is often the case that tasks are created only after other tasks have finished (this can be implemented by using `finish`).

Both notions of task level discussed above disregard whether a created task is *active*, i.e., actually executing, or *suspended*. In the example, the main procedure is suspended while the execution of the `finish` block is going on. Our analysis can give an upper bound `to`−`from`+1 to the *peak of available* tasks (i.e., those which can be active at the same time), which is almost half the one obtained for total and live tasks. Such bound is useful when allocating tasks on processors, since active tasks are the only ones actually needing a processor to execute.

The analysis uses most of the machinery already developed in COSTA, and defines all the notions of task level as something similar to cost models. Further effort has been devoted to adapting the analyzer to this specific language, and to optimizations: for example, knowing which tasks, among the ones spawned inside a procedure p, are still live after p ends allows, in general, improving the obtained upper bounds.

5.3 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*. Most existing value analyses are only applicable to numeric variables which satisfy two *locality* conditions: (1) all occurrences of a variable refer to the same memory location, and (2) memory locations can only be modified using the corresponding variable. The conditions above are not satisfied when numeric variables are stored in shared mutable data structures such as the heap. Condition (1) does not hold because memory locations (numeric variables) are accessed using reference variables, whose value can change during the execution. Condition (2) does not hold because a memory location can be modified using different references which are aliases and point to such memory location.

Example 9. Consider the following loops where f is a field of integer type:

```
①while(x.f > 0) {        ②while(x.f > 0) {        ③while(x.y.f > 0) {
   x.f = x.f-1;             x.f = x.f-1;             x.f = x.f-1;
}                           x = x.next;              y.f = y.f+1;
                         }                        }
```

Loop ① terminates in sequential execution because x.f decreases at each iteration and, for any initial value of x.f, there are only a finite number of values which x.f can take before reaching zero. Unfortunately, applying standard value analyses on numeric fields can produce wrong results, and further conditions are required. E.g., if we add the instruction x=x.next; as in loop ②, the memory location pointed to by x.f changes, invalidating Condition 1. Also, Condition 2 is false if we add y.f=y.f+1 as in loop ③, as x and y may be aliases.

The approach developed in COSTA [15,7,5] 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. Our approach consists of the following steps: (1) partition the program to be analyzed into scopes, in our case we use the *strongly connected components* of the program, (2) identify with a *reference constancy analysis* the *access paths* used to access to the heap, (3) check the above *locality* conditions for the access paths, that is, if an access path meets the above conditions it can be safely handled by a heap-insensitive analysis, (4) transform the program by introducing local *ghost* variables whose values represent the values of the corresponding heap accesses, and (5) analyze the transformed program scope by scope using any heap-insensitive static analyzer, in particular, the one that we already have in COSTA. Let us describe the main components of the heap-sensitive analysis implemented in COSTA:

Reference Constancy Analysis. We first develop a *reference constancy analysis* which infers those memory locations of fields which are constant in the considered scope. The idea behind this analysis is similar in spirit to that of the classical numeric constant propagation analysis [20]. However, in addition to numerical constants, the values computed by our analysis can include symbolic expressions that refer to locations in (the initial) heap. Such expressions encode as well the way in that the corresponding memory locations are reached (e.g., the dereferenced fields). Intuitively, our analysis will assign to each variable (at each program point) an *access path* which describes its possible memory locations whenever the execution reaches that point. Access paths can be separated in *read* accesses and *write* accesses. In the analysis, if a reference variable is updated inside a scope its access path is not constant in this scope, so we cannot keep track of this variable by using a local variable.

Example 10. Let us consider the loops in Ex. 9. We obtain the following read $R(f)$ and write sets $W(f)$ for field f by applying the reference constancy analysis.

① $R(f) = \{x.f\}$ $W(f) = \{x.f\}$	② $R(f) = \emptyset$ $W(f) = \emptyset$	③ $R(f) = \{x.f, y.f\}$ $W(f) = \{x.f, y.f\}$

Observe that in loop ① field f is accessed only using reference variable x. In loop ② the sets are empty because the location of variable x is updated by the instruction x = x.next inside the loop, so it is not constant in the scope of the loop. Loop ③ has two different access paths through x and y to field f.

Locality. Intuitively, in order to ensure a sound transformation, a field f can be considered local in a scope S if all read and write accesses to it in all reachable scopes are performed through the same access path l, that is, if $R(f) \cup W(f) = \{l\}$. This makes it safe to replace such heap access by a corresponding local ghost variable.

Example 11. From the results obtained in Ex. 10, and according to the locality condition above, field accesses in loop ① meet the locality condition since $R(f) \cup W(f) = \{\text{x.f}\}$. Field f is not local in the scope of loops ② neither ③, in ② because the union $R(f) \cup W(f)$ is empty, and in ③, the union contains two different references accessing field f, $\{x.f, y.f\}$.

Transformation. Our approach is based on instrumenting the program with extra local (ghost) variables that expose the values of those locations to a heap-insensitive analysis as follows: (1) for each heap access that is local in scope S, we introduce a ghost variable g; (2) when the content of the memory location is modified, we modify g accordingly; and (3) when the memory location is read, we read the value from g. This approach has one clear advantage: there is no need to change existing static analysis tools to make them heap-sensitive, we simply apply them on the transformed program, and then the properties inferred for the ghost variables hold also for the corresponding memory locations.

Example 12. Field f behaves locally in loop ①, so the heap accesses to f can be transformed into ghost variables resulting in the following program, whose termination and cost can be inferred by a heap-insensitive analyzer:

```
g = x.f;
while(g > 0) {
   g = g-1;
}
x.f = g;
```

Recently, COSTA has included an abstract interpretation based heap-sensitive analysis [44] that infers reachability and acyclicity of heap allocated data structures. This analysis infers weather some reference variables point to an acyclic data structure which is useful for the analysis of termination and resource usage.

5.4 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. Incremental resource usage analysis comprises two phases: (1) devising an incremental analysis framework for all pre-analysis required to infer cost relations and (2) making the process of computing a closed-form upper bound incremental.

As regards (1), in other approaches to incremental analysis, such as in the logic programming context [25], the analysis is focused in a single abstract domain. In contrast, COSTA includes a *multi-domain* incremental analysis engine [10] 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 engine 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.

The incremental analysis engine starts from a program, its analysis results, and a modified method m. In addition to reanalyzing method m for all domains, other methods may require reanalysis as well: namely methods invoked by m with a different calling description (those methods are the *descendants* of m in the program call-graph); and methods which invoke m and, because of new results, require their reanalysis (referred to as *ancestors* of m). The process of incremental reanalysis transitively reanalyzes descendants and ancestors of previously reanalyzed methods, until a fixed point is reached. Once the new pre-analyses results have been computed, cost relations that correspond to reanalyzed methods are recomputed.

As regards (2), an upper bound is a global expression which includes the upper bounds of the relations it calls. If method m changes, the upper bound expressions that (directly or transitively) use m are no longer valid, since it is not possible to distinguish within an upper bound which part of the cost is associated to m. A fundamental idea to support incremental inference of upper bounds is to annotate each cost subexpression with the name of the relation it comes from. For this purpose, we define the notion of *upper bound summary* to keep the annotated cost expression, the invariant and the size relations for a method m. Given an upper bound summary for a method, it is possible to replace the cost subexpressions associated to those methods invoked from it whose upper bounds have changed by their new upper bounds, and without having to recompute the whole upper bound for the method.

6 Conclusions and Future Work

We have overviewed the main techniques used to infer resource consumption bounds in the COSTA system. In the future, we plan to extend our work along the following directions.

Improvements in Computing Symbolic Bounds. We plan to study new techniques to infer more precise lower/upper bounds on the number of iterations that loops perform. As this is an independent component, COSTA will directly be benefited from any improvement in this regard. In addition, so far we have used linear invariants for inferring linear ranking functions, minimum number of iterations of a loop and maximization or minimization of cost expressions. Another extension of our work would be inferring nonlinear loop invariants using symbolic summation and algebraic techniques. Another possible direction is inferring nonlinear input-output (size) relations for methods by viewing the output as the cost that is consumed by the corresponding method. This way, we can view the problem of inferring such input-output relations as solving corresponding CR. Also, we plan to develop new techniques to solve cost relations, to handle some programs for which amortised analysis is not needed.

Acyclicity Analysis. COSTA contains a component which performs an acyclicity analysis [23] based on tracking the reachability between locations in the heap. Future work includes improving this analysis by considering, for a path between two heap locations, the name of the fields involved. For example, this could allow to detect that, in a double-linked list, cycles must forcefully traverse both the `next` and the `prev` field, so that a loop where the data structure is traversed by `x=x.next` is guaranteed to terminate in spite of the cyclicity of the data structure, since only `next` is traversed.

Heap-sensitive Analysis. The heap-sensitive approach contained in COSTA shows that analyzing program fragments rather than the application as a whole, and under certain locality conditions, it is feasible to keep track of heap-allocated data by means of local variables. However, there are cases when the locality conditions cannot be proven unconditionally. In such cases, we seek to provide *aliasing preconditions* between the input arguments which, when satisfied in the initial state, can guarantee the termination of the program.

Proving Termination of Concurrent Programs. In the current COSTA system, termination is independently proved for each concurrent component. That is, no assumptions are made about the interactions between different concurrent components. Unfortunately, this approach is insufficient for many real world applications where several concurrent tasks depend on each other. We plan to investigate these cases in the future. Another possible research line is to consider conditional termination in open systems. In this case, we should extract under which conditions a given component terminates.

Deadlock Analysis of ABS Programs. Deadlocks are one of the most common errors in concurrent programs. There has been some theoretical research about deadlocks in ABS programs [24]. However, there is no practical analysis that can be used for real programs. Deadlocks can occur when several concurrent components are waiting for each other (Circular dependency). We intend to develop a deadlock analysis that combines the MHP analysis and a points-to analysis. The MHP information can detect sets of synchronization instructions that may happen in parallel and the points-to analysis can identify dependencies (which components are waiting for which others) within these sets.

Cost Analysis of Concurrent Java programs. At present, the extension of COSTA handles concurrent primitives of ABS programs where the number of context switches among concurrently running objects are determined by the release points defined by the high level language constructs. That means the concurrent code runs sequentially between two release points which simplifies the resource usage analysis for ABS programs. However, in thread-based concurrent java programs, the context switch can happen at any program point. Hence, existing analysis looses too much precision in inferring size relations or bounding loop iterations and thus does not provide any useful information on resource usage of concurrent java programs. In order to extend our analysis to handle thread-based concurrent primitives, it is essential to infer relational invariants both on shared variables and thread local variables at every program points. It is also essential to infer relations between shared and thread local variables. Thus, these invariants can be used to infer the size relations which capture the cost of the programs.

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, TIN2012-38137-C02 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. Adachi, A., Kasai, T., Moriya, E.: A Theoretical Study of the Time Analysis of Programs. In: Becvar, J. (ed.) MFCS 1979. LNCS, vol. 74, pp. 201–207. Springer, Heidelberg (1979)
2. Albert, E., Arenas, P., Alonso, D., Genaim, S., Puebla, G.: Asymptotic Resource Usage Bounds. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 294–310. Springer, Heidelberg (2009)
3. 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)
4. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: COSTABS: A Cost and Termination Analyzer for ABS. In: Procs. of PEPM 2012, pp. 151–154. ACM Press (2012)
5. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Field-Sensitive Value Analysis by Field-Insensitive Analysis. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 370–386. Springer, Heidelberg (2009)
6. 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)
7. Albert, E., Arenas, P., Genaim, S., Puebla, G., Ramírez Deantes, D.V.: From Object Fields to Local Variables: a Practical Approach to Field-Sensitive Analysis. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 100–116. Springer, Heidelberg (2010)

8. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost Analysis of Object-Oriented Bytecode Programs. Theoretical Computer Science (Special Issue on Quantitative Aspects of Programming Languages) 413(1), 142–159 (2012)
9. Albert, E., Arenas, P., Genaim, S., Zanardini, D.: Task-Level Analysis for a Language with Async-Finish parallelism. In: Proc. of LCTES 2011, pp. 21–30. ACM Press (2011)
10. Albert, E., Correas, J., Puebla, G., Román-Díez, G.: Incremental Resource Usage Analysis. In: Procs. of PEPM 2012, pp. 25–34. ACM Press (2012)
11. Albert, E., Flores-Montoya, A.E., Genaim, S.: Analysis of May-Happen-in-Parallel in Concurrent Objects. In: Giese, H., Rosu, G. (eds.) FORTE 2012 and FMOODS 2012. LNCS, vol. 7273, pp. 35–51. Springer, Heidelberg (2012)
12. Albert, E., Flores-Montoya, A., Genaim, S.: Maypar: A May-Happen-in-Parallel Analyzer for Concurrent Objects. In: Proc. of FSE-20, SIGSOFT/FSE 2012, pp. 1–4. ACM (2012)
13. Albert, E., Genaim, S., Gómez-Zamalloa, M.: Heap Space Analysis for Garbage Collected Languages. Science of Computer Programming (2012) (to appear)
14. 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)
15. Albert, E., Genaim, S., Román-Díez, G.: Conditional Termination of Loops over Arrays. In: Proc. of Bytecode 2012 (2012)
16. Alonso-Blas, D.E., Genaim, S.: On the Limits of the Classical Approach to Cost Analysis. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 405–421. Springer, Heidelberg (2012)
17. Bagnara, R., Hill, P.M., Zaffanella, E.: 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), 3–21 (2008)
18. Benzinger, R.: Automated Higher-Order Complexity Analysis. Theoretical Computer Science 318(1-2), 79–103 (2004)
19. Bruynooghe, M., Codish, M., Gallagher, J.P., Genaim, S., Vanhoof, W.: Termination analysis of logic programs through combination of type-based norms. ACM Transactions on Programming Languages and Systems 29(2) (2007)
20. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proc. of POPL 1977, pp. 238–252. ACM Press (1977)
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. Feautrier, P.: Parametric Integer Programming. RAIRO Recherche Opérationnelle 22(3), 243–268 (1988)
23. Genaim, S., Zanardini, D.: The Acyclicity Inference of COSTA. In: 11th International Workshop on Termination (2010)
24. Giachino, E., Laneve, C.: Analysis of Deadlocks in Object Groups. In: Bruni, R., Dingel, J. (eds.) FORTE 2011 and FMOODS 2011. LNCS, vol. 6722, pp. 168–182. Springer, Heidelberg (2011)
25. Hermenegildo, M., Puebla, G., Marriott, K., Stuckey, P.: Incremental Analysis of Constraint Logic Programs. ACM Transactions on Programming Languages and Systems 22(2), 187–223 (2000)
26. Hermenegildo, M.V., Bueno, F., Carro, M., López, P., Mera, E., Morales, J.F., Puebla, G.: An Overview of Ciao and its Design Philosophy. Theory and Practice of Logic Programming 12(1-2), 219–252 (2012), http://arxiv.org/abs/1102.5497

27. Hoffmann, J., Aehlig, K., Hofmannn, M.: Multivariate Amortized Resource Analysis. ACM Transactions on Programming Languages and Systems 34(3), 14:1–14:62 (2012)
28. 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)
29. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice-Hall (1993)
30. Jost, S.: Automated Amortised Analysis. PhD thesis, Ludwig-Maximilians-Universität (August. 2010)
31. Le Metayer, D.: ACE: An Automatic Complexity Evaluator. ACM Transactions on Programming Languages and Systems 10(2), 248–266 (1988)
32. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Addison-Wesley (1996)
33. Navas, J., Mera, E., López-García, P., Hermenegildo, M.V.: User-Definable Resource Bounds Analysis for Logic Programs. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 348–363. Springer, Heidelberg (2007)
34. 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)
35. Rodriguez, D.: Amortised Resource Analysis for Object-Oriented Programs. Phd thesis, LMU Munich (October 2012)
36. Rosendahl, M.: Automatic Complexity Analysis. In: Proc. of FPCA 1989, pp. 144–156. ACM Press (1989)
37. Sands, D.: A Naïve Time Analysis and its Theory of Cost Equivalence. Journal of Logic and Computation 5(4), 495–541 (1995)
38. 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)
39. Srinivasan, S., Mycroft, A.: Kilim: Isolation-Typed Actors for Java. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 104–128. Springer, Heidelberg (2008)
40. Tarjan, R.E.: Amortized Computational Complexity. SIAM Journal on Algebraic and Discrete Methods 6(2), 306–318 (1985)
41. Wadler, P.: Strictness Analysis Aids Time Analysis. In: ACM Symposium on Principles of Programming Languages (POPL 1988). ACM Press (1988)
42. Wegbreit, B.: Mechanical Program Analysis. Communications of the ACM 18(9), 528–539 (1975)
43. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theory and Practice of Logic Programming 12(1-2), 67–96 (2012)
44. Genaim, S., Zanardini, D.: Reachability-based Acyclicity Analysis by Abstract Interpretation. Theoretical Computer Science (2013)
45. 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)
46. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: COSTA: Design and implementation of a cost and termination analyzer for java bytecode. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 113–132. Springer, Heidelberg (2008)