More Precise yet Widely Applicable Cost Analysis

Elvira Albert[1], Samir Genaim[1], and Abu Naser Masud[2]

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

Abstract. Cost analysis aims at determining the amount of resources required to run a program in terms of its input data sizes. Automatically inferring *precise* bounds, while at the same time being able to handle a *wide* class of programs, is a main challenge in cost analysis. (1) Existing methods which rely on computer algebra systems (*CAS*) to solve the obtained *cost recurrence equations* (*CR*) are very precise when applicable, but handle a very restricted class of *CR*. (2) Specific solvers developed for *CR* tend to sacrifice accuracy for wider applicability. In this paper, we present a novel approach to inferring precise *upper* and *lower* bounds on *CR* which, when compared to (1), is strictly more widely applicable while precision is kept and when compared to (2), is in practice more precise (obtaining even tighter complexity orders), keeps wide applicability and, besides, can be applied to obtain useful lower bounds as well. The main novelty is that we are able to accurately bound the worst-case/best-case cost of each iteration of the program loops and, then, by summing the resulting sequences, we achieve very precise upper/lower bounds.

1 Introduction

Static cost analysis [13] aims at automatically inferring the resource consumption (or cost) of executing a program as a function of its input data sizes. The classical approach to cost analysis consists of two phases. First, given a program and a *cost model*, the analysis produces *cost relations* (*CRs*), i.e., a system of recursive equations which capture the cost of the program in terms of the size of its input data. Let us motivate our work on the contrived example depicted in Fig. 1a. The example is sufficiently simple to explain the main technical parts of the paper, but still interesting to understand the challenges and precision gains. For this program and the *memory consumption* cost model, the cost analysis of [3] generates the *CR* which appears in Fig. 1b. This cost model estimates the number of objects allocated in the memory. Cost analyzers are usually parametric on the cost model, e.g., cost models widely used are the number of executed bytecode instructions, number of calls to methods, etc. Observe that the structure of the Java program and its corresponding *CR* match. The equations for C correspond to the `for` loop, those of B to the inner `while` loop and those of A to the outer `while` loop. The recursive equation for C states that the memory consumption of executing the inner loop with $\langle k, j, n\rangle$ such that $k{<}n{+}j$ is 1 (one object) plus

```
void f(int n) {
 List l = null;
 int i=0;
 while (i<n) {
   int j=0;
   while ( j<i ) {
     for(int k=0;k<n+j;k++)
       l=new List(i*k*j,l);
     j=j+random()?1:3;}
   i=i+random()?2:4;
 }  }
```

(a) Running Example

$$
\begin{array}{lll}
F(n) & = A(0,n) & \{\} \\
A(i,n) & = 0 & \{i \geq n\} \\
A(i,n) & = B(0,i,n) + A(i',n) & \\
 & \{\ i < n, i+2 \leq i' \leq i+4\} & \\
B(j,i,n) & = 0 & \{j \geq i\} \\
B(j,i,n) & = C(0,j,n) + B(j',i,n) & \\
 & \{j < i, j+1 \leq j' \leq j+3\} & \\
C(k,j,n) & = 0 & \{k \geq n+j\} \\
C(k,j,n) & = 1 + C(k',j,n) & \\
 & \{k' = k+1, k < n+j\} &
\end{array}
$$

(b) *CRs* for Memory Consumption

Fig. 1: Running Example and its Cost Relation System

that of executing the loop with $\langle k', j, n\rangle$ where k'=k+1. The recursive equation for B states that executing the loop with $\langle j, i, n\rangle$ costs as executing $C(0, j, n)$ plus executing the same loop with $\langle j', i, n\rangle$ where j+1≤j'≤j+3. While, in the Java program, j' can be either j+1 or j+3, due to the static analysis, the case for j+2 is added in order to have a convex shape [7]. The process of generating *CRs* heavily depends on the programming language and, thus, multiple analyses have been developed for different paradigms. However, the resulting *CRs* are a common target of cost analyzers.

Our work focuses on the second phase of cost analysis: once *CRs* are generated, analyzers try to compute *closed-forms* for them, i.e., cost expressions which are not in recursive form. Two main approaches exist: (1) Since cost relations are syntactically quite close to *recurrence relations*, most cost analysis frameworks rely on existing *Computer Algebra Systems* (*CAS*) for finding closed-forms. Unfortunately, only a restricted class of *CRs* can be solved using *CAS*, namely only some of those which have an exact solution. In practice, this seldom happens. For instance, in the cost relation B, variable j' can increase by one, by two or by three at each iteration, so an exact cost function which captures the cost of any possible execution does not exist. (2) Instead, specific upper-bound solvers developed for *CRs* try to reason on the worst-case cost and obtain sound *upper-bounds* (UBs) of the resource consumption. As regards *lower-bounds* (LBs), due in part to the difficulty of inferring under-approximations, general solvers for *CRs* able to obtain useful approximations of the best-case cost have not been developed yet. As regards the number of iterations, for B, the worst-case (resp. best-case) cost must assume that j' increases by one (resp. three) at each iteration. Besides, there is the problem of bounding the cost of each of the iterations. For UBs, the approach of [2] assumes the worst-case cost for all loop iterations. E.g., an UB on the cost of any iteration of B is n_0+i_0−1, where n_0 and i_0 are respectively the initial values for n and i. This corresponds to the memory allocation of the last iteration of the corresponding `while` loop. This approximation, though often imprecise, makes it possible to obtain UBs for most *CRs* (and thus

programs). Observe that it is not useful to obtain LBs since by assuming the best-case cost for all iterations, the obtained LB would be in most cases zero.

Needless to say, precision is fundamental for most applications of cost analysis. For instance, UBs are widely used to estimate the space and time requirements of programs execution and provide resource guarantees [8]. Lack of precision can make the system fail to prove the resource usage requirements imposed by the software client. LBs are used to scheduling the distribution of tasks in parallel execution. Likewise, precision will be essential to achieve a satisfactory scheduling. A main achievement in this paper is the seamless integration of both approaches so that we get the best of both worlds: precision as (1), whenever possible, while applicability as close to (2) as possible. Intuitively, the precision gain stems from the fact that, instead of assuming the worst-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. For UBs, we do so by taking advantage of existing automatic techniques, which are able to infer UBs on the number of loop iterations and the worst-case cost of all of them, in order to generate a novel form of *(worst-case) recurrence relations* which can be solved by *CAS*. The exact solution of such recurrence relation (*RR*) is guaranteed to be a precise UB of the original *CR*. As another contribution, we present a new technique for inferring LBs on the number of iterations. Then, the problem of inferring LBs on the cost becomes dual to the UBs.

To the best of our knowledge, this is the first general approach to inferring LBs from *CRs* and, as regards UBs, the one that achieves a better precision vs. applicability balance. Importantly, when *CRs* originate from nested loops in which the cost of the inner loop depends on the outer loop, our approach obtains more precise bounds than [9, 2]. Moreover, as our experiments show, we are able to produce upper bounds with a tighter complexity order than those inferred by [9, 2], e.g., improving from $O(n * log(n))$ to $O(n)$. On the other hand, when compared to [10], our approach is of wider applicability in the sense that it can infer general polynomial, exponential and logarithmic bounds, not only univariate polynomial bounds as [10]. Since *CRs* obtained from different programming languages have the same features, our work is applicable to cost analysis of any language. Preliminary experiments on Java (bytecode) programs confirm the good balance between the accuracy and applicability of our analysis.

2 Preliminaries

The sets of natural, integer, real, non-zero natural and non-negative real values are denoted respectively by $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, $\mathbb{N}^+$ and $\mathbb{R}^+$. We write x, y, z to denote variables which range over $\mathbb{Z}$. A *linear expression* has the form $v_0{+}v_1x_1{+}\ldots{+}v_nx_n$, where $v_i{\in}\mathbb{Z}$. A *linear constraint* (over $\mathbb{Z}$) has the form $l_1{\leq}l_2$, where l_1 and l_2 are linear expressions. We write $l_1{=}l_2$ instead of $l_1{\leq}l_2 \wedge l_2{\leq}l_1$, and $l_1{<}l_2$ instead of $l_1{+}1{\leq}l_2$. We use $\bar{t}$ to denote a sequence of entities $t_1, \ldots, t_n$. We use φ or Ψ to denote a set (conjunction) of linear constraints and $\varphi_1 \models \varphi_2$ to indicate that φ_1 implies φ_2. A mapping from a set of variables to integers is denoted by σ.

2.1 Cost Relations: The Common Target of Cost Analyzers

Let us now recall the general notion of *cost relation* (*CR*) as defined in [2] which generalizes the *CRs* yield by most analyzers. The basic building blocks of *CRs* are the so-called *cost expressions* e which are generated using this grammar:

$$e ::= r \mid \mathsf{nat}(\ell) \mid e + e \mid e * e \mid e^r \mid \log(\mathsf{nat}(\ell)) \mid n^{\mathsf{nat}(l)} \mid \max(S)$$

where $r \in \mathbb{R}^+$, $n \in \mathbb{N}^+$, l is a linear expression, S is a non empty set of cost expressions and $\mathsf{nat} : \mathbb{Z} \to \mathbb{N}$ is defined as $\mathsf{nat}(v) = \max(\{v, 0\})$. Importantly, linear expressions are always wrapped by nat in order to avoid negative evaluations. For instance, as we will see later, an UB for $C(k, j, n)$ is $\mathsf{nat}(n + j - k)$. Without the use of nat, the evaluation of $C(5, 5, 11)$ results in the negative cost -1 which must be evaluated to zero, since they correspond to executions in which the `for` loop is not entered (i.e., $k \geq n + j$).

Definition 1 (Cost Relation). *A* cost relation C *is defined by a set of equations of the form* $\mathcal{E} \equiv \langle C(\bar{x}) = e + \sum_{i=1}^{k} D_i(\bar{y}_i) + \sum_{j=1}^{n} C(\bar{z}_j), \varphi \rangle$ *with* $k, n \geq 0$*, where* C *and* D_i *are cost relation symbols with* $D_i \neq C$*; all variables* $\bar{x}$*,* $\bar{y}_i$ *and* $\bar{z}_j$ *are distinct;* e *is a cost expression; and* φ *is a set of linear constraints over* $vars(\mathcal{E})$*.*

The evaluation of a *CR* C for a given valuation $\bar{v}$, denoted $C(\bar{v})$, is like a constraint logic program [11] and consists of the next steps: (1) first a matching equation of the form $\langle C(\bar{x}) = e + \sum_{i=1}^{k} D_i(\bar{y}_i) + \sum_{j=1}^{n} C(\bar{z}_j), \varphi \rangle$ is chosen; (2) then, we need to choose an assignment σ s.t. $\sigma \models \bar{v} = \bar{x} \wedge \varphi$; (3) then, evaluate e w.r.t. σ and accumulate it to the result; and (4) evaluate each call $D_i(\bar{v}_i)$ where $\bar{v}_i = \sigma(\bar{y}_i)$ and $C(\bar{v}_j)$ where $\bar{v}_j = \sigma(\bar{z}_j)$. The result (i.e., the cost of the execution) of the evaluation is the sum of all cost expressions accumulated in step (3). Even if the original program is deterministic, due to the abstractions performed during the generation of the *CR*, it might happen that several results can be obtained for a given $C(\bar{v})$. Correctness of the underlying analysis used to obtain the *CR* must ensure that the actual cost is one of such solutions (see [2]). This makes it possible to use *CR* to infer both UBs and LBs from them.

Example 1. Let us evaluate $B(0, 3, 3)$. The only matching equation is the second one for B. In step (2), we choose an assignment σ. Here we have a nondeterministic choice for selecting the value of j' which can be 1, 2 or 3. In step (4), we evaluate the cost of $C(0, 0, 3)$. Finally, one of the recursive calls of $B(1, 3, 3)$, $B(2, 3, 3)$ or $B(3, 3, 3)$ will be made, depending on the chosen value for j'. If we continue executing all possible derivations until reaching the base cases, the final result for $B(0, 3, 3)$ is any of $\{9, 10, 13, 14, 15, 18\}$. The actual cost is guaranteed to be one of such values.

W.l.o.g., we formalize our method by making two simplifications: (1) *Direct recursion:* we assume that all recursions are *direct* (i.e., cycles in the call graph are of length one). Direct recursion can be automatically achieved by applying partial evaluation as described in [2]. (2) *Standalone cost relations:* we assume that *CRs* do not depend on any other *CR*, i.e., the equations do not contain external calls and thus have this simplified form $\langle C(\bar{x}) = e + \sum_{j=1}^{n} C(\bar{z}_j), \varphi \rangle$. This

can be assumed because our approach is compositional. We start by computing bounds for the *CRs* which do not depend on any other *CRs*, e.g., C in Fig. 1b is solved by providing the UB $\mathsf{nat}(n + j - k)$. Then, we continue by replacing the computed bounds on the equations which call such relation, which in turn become standalone. For instance, replacing the above solution in the relation B results in the equation $B(j, i, n) = \mathsf{nat}(n + j) + B(j', i, n), \{j < i, j + 1 \leq j' \leq j + 3\}$. This operation is repeated until no more *CR* need to be solved. In what follows, *CR* refers to standalone *CRs* in direct recursive form.

2.2 Single-Argument Recurrence Relations

It is fundamental for this paper to understand the differences between *CRs* and *RRs*. The following features have been identified in [2] as main differences, which in turn justify the need to develop specific solvers to bound *CRs*:

1. *CRs* often have *multiple arguments* that increase or decrease over the relation (e.g., in A variable i' increases). The number of evaluation steps (i.e., recursive calls performed) is often a function of such several arguments.
2. *CRs* often contain *inexact size relations*, e.g., variables range over an interval $[a, b]$ (e.g., variable j' in B). Thus, given a valuation, we might have several solutions which perform a different number of evaluation steps.
3. Even if the original programs are deterministic, due to the loss of precision in the first stage of the static analysis, *CRs* often involve several *non-deterministic equations*. This will be further explained in Sec. 4.3.

As a consequence of 2 and 3, an exact solution often does not exist and hence *CAS* just cannot be used in such cases. But, even if a solution exists, due to such three additional features, *CAS* do not accept *CRs* as a valid input. Below, we define a class of recurrence equations that *CAS* can handle.

Definition 2 (single-argument *RR*). *A* single-argument *recurrence relation C is defined by at most one recursive equation $\langle C(x) = E + \sum_{i=1}^{n} C(x - 1)\rangle$ where E is a function on x (and might have constant symbols), and a base case $\langle C(0) = \kappa\rangle$ where κ is a symbol representing the value of the base case.*

Depending on the number of recursive calls in the recursive equation and the expression E, such solution can be of different complexity classes (exponential, polynomial, etc.). A closed-form solution for $C(x)$, if exists, is an arithmetic expression that depends only on the variable x, the base-case symbol κ, and might include constant symbols that appear in E. W.l.o.g., in what follows, we assume that $\kappa = 0$. In the implementation, we replace κ in the closed-form UB (resp. LB) by the maximum (resp. minimum) value that it can take, as done in [2].

3 An Informal Account of Our Approach

This section informally explains the approximation we want to achieve and compares it to the actual cost and the approximation of [2]. Consider a *CR* in its

simplest form with a base case $\langle C(\bar{x})=0, \varphi_0\rangle$ and a recursive case with a single recursive call $\langle C(\bar{x})=e+C(\bar{x}'), \varphi_1\rangle$. The challenge is to accurately estimate the cost of $C(\bar{x})$ for any input. *CAS* aim at obtaining the exact cost function. As we have discussed in Sec. 2.2, this is often not possible since even a single evaluation has multiple solutions. Thus, the goal of static cost analysis is to infer closed-form UBs/LBs for C. Our starting point is the general approximation for UBs proposed by [2] which has two dimensions. (1) *Number of applications of the recursive case:* The first dimension is to infer an UB on the number of times the recursive equations can be applied (which, for loops, corresponds to the number of iterations). This is done by inferring an UB $\hat{n}$ on the length of chains of recursive calls; (2) *Cost of applications:* The second dimension is to infer an UB $\hat{e}$ for all e_i. Then, for a relation with a single recursive call, $\hat{n} * \hat{e}$ is guaranteed to be an UB for C. If the relation C had two recursive calls, the solution would be an exponential function of the form $2^{\hat{n}} * \hat{e}$. Programming-languages techniques of wide applicability have been proposed by [2] in order to solve the two dimensions, as detailed below.

Ranking functions. A ranking function is a function f such that for any recursive equation $\langle C(\bar{x})=e+C(\bar{x}_1)+\cdots+C(\bar{x}_k), \varphi\rangle$ in the *CR*, it holds that $\forall 1\leq i\leq k.\varphi \models f(\bar{x})>f(\bar{x}_i)\wedge f(\bar{x})>0$. This guarantees that when evaluating $C(\bar{v})$, the length of any chain of recursive calls to C cannot exceed $f(\bar{v})$. Thus, f is used to bound the length of these chains [2, 5, 6]. We rely on [2] for automatically inferring a ranking function $\hat{f}_C(\bar{x}_0)$ for C (variables $\bar{x}_0$ denote the initial values).

Maximization. In [2] the second dimension is solved by first inferring an *invariant* $\langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \Psi\rangle$, where Ψ is a set of linear constraints, which describes the relation between the values that $\bar{x}$ can take in any recursive call and the initial values $\bar{x}_0$. Then in order to generate $\hat{e}$ each $\mathsf{nat}(l) \in e$ is replaced by $\mathsf{nat}(\hat{l})$ where $\hat{l}$ is a linear expression (over $\bar{x}_0$) which is an UB for any valuation of l. We rely on the techniques of [2] in order to automatically obtain $\mathsf{nat}(\hat{l})$ for $\mathsf{nat}(l)$.

Our challenge is to improve precision of [2] while keeping a similar applicability for UBs and, besides, be able to apply our approach to infer useful LBs. The fundamental idea is to generate a sequence of (different) elements $u_1, \ldots, u_{\hat{n}}$ such that for any concrete evaluation $e_1, \ldots, e_n$ it holds $\forall 0 \leq i \leq n-1$. $u_{\hat{n}-i}\geq e_{n-i}$. Note that it is ensured that the last n elements of the u sequence are larger than (or equal to) the n elements of the e sequence, but it is not guaranteed that $u_i\geq e_i$. This guarantees that $u_1+\cdots+u_{\hat{n}}$ is an UB for $e_1+\cdots+e_n$. Our UB is potentially more precise than $\hat{n}*\hat{e}$, since each e_i is approximated more tightly by a corresponding u_j. Technically, we do this by transforming the *CR* into a (worst-case) *RR* (as in Def. 2) whose closed-form solution is $u_1 + \cdots + u_{\hat{n}}$. The novel idea is to view $u_1, \cdots, u_{\hat{n}}$ as an arithmetic sequence that starts from $u_{\hat{n}}\equiv\hat{e}$ and each time decreases by $\check{d}$ where $\check{d}$ is an under approximation of all $d_i=e_{i+1}-e_i$, i.e., $u_i=u_{i-1}+\check{d}$. In our approach the problem of inferring LBs is dual, namely we can infer a LB $\check{n}$ on the length of chains of recursive calls, the minimum value $\check{e}$ to which e_i can be evaluated, and then sum the sequence $\ell_1, \ldots, \ell_{\check{n}}$ where $\ell_i=\ell_{i-1}+\check{d}$ and $\ell_1=\check{e}$.

4 Inference of Precise Upper Bounds

In this section, we present our approach to accurately infer UBs on the resource consumption in three steps: first in Sec. 4.1, we handle a subclass of *CRs* which accumulate a constant cost, then we handle *CRs* which accumulate non-constant costs in Sec. 4.2 and *CRs* with multiple overlapping equations in Sec. 4.3.

4.1 Constant Cost Relations

We consider *CRs* defined by a single recursive equation with constant cost:

$$\langle C(\bar{x}) = 0, \varphi_1 \rangle \mid \langle C(\bar{x}) = e + C(\bar{x}_1) + \cdots + C(\bar{x}_k), \varphi_2 \rangle \quad (1)$$

where e contributes a *constant cost*, i.e., it is a constant number or an expression that always evaluates to the same value. As explained in Sec. 3, any chain of recursive calls in C is at most of length $\hat{f}_C(\bar{x}_0)$ (when starting from $C(\bar{x}_0)$). We aim at obtaining an UB for C by solving a *RR* P_C in which all chains of recursive calls are of length $\hat{f}_C(\bar{x}_0)$. Intuitively, $P_C(x)$ can be seen as a special case of a *RR* with the same number of recursive calls as in C, where all chains of recursive calls are of length x, and each application accumulates the constant cost e. Its solution can be then instantiated for the case of C by replacing x by $\hat{f}_C(\bar{x}_0)$.

Definition 3. *The worst-case RR of C is* $\langle P_C(x) = e + P_C(x-1) + \cdots + P_C(x-1) \rangle$.

The main achievement of the above transformation is that, for constant *CRs*, we get rid of their problematic features described in Sec. 2.2 which prevented us from relying on *CAS* to obtain a precise solution. The following theorem explains how the closed-form solution of the *RR* P_C can be transformed into an UB for the *CR* C.

Theorem 1. *If E is a solution for* $P_C(x)$ *then* $E[x/\hat{f}_C(\bar{x}_0)]$ *is an UB for* $C(\bar{x}_0)$.

Example 2. The worst-case *RR* of the *CR* C of Fig. 1b is $\langle P_C(x) = 1 + P_C(x-1) \rangle$, which is solved using *CAS* to $P_C(x) = x$ for any $x \geq 0$. The UB for C is obtained by replacing x by $\hat{f}_C(k_0, j_0, n_0) = \mathsf{nat}(j_0 + n_0 - k_0)$.

4.2 Non-constant Cost Relations

During cost analysis, in many cases we obtain *CRs* like the one of Eq. 1, but with a non-constant expression e which is evaluated to different values e_i in different applications of the recursive equation. The transformation in Def. 3 would not be correct since in these cases e must be appropriately related to x. In particular, the main difficulty is to simulate the accumulation of the non-constant expressions e_i at the level of the *RR*. As we have illustrated in Sec. 3, the novel idea is to simulate this behavior with an arithmetic sequence that starts from the maximum value that e can take, and in each step decreases by the minimum distance $\check{d}$ between two consecutive expressions e_i and e_{i+1}. Since the expression e might have a complex form (e.g., exponential, polynomial, etc),

inferring a precise LB on the distance $\check{d}$ is usually impractical. A key observation in our approach is that, since variables are wrapped by nat, it is enough to reason on the behavior of its nat sub-expressions, i.e., we only need to understand how each $\mathsf{nat}(l)$ of e (denoted $\mathsf{nat}(l) \in e$) changes along a sequence of recursive calls.

Definition 4 (nat with linear behaviour). *Consider the CR C of Eq. 1 with e a (possibly) non-constant expression. We say that a given $\mathsf{nat}(l) \in e$ is linearly increasing (resp. decreasing) if there exists a non-negative integer $\check{d}$, such that for a given renamed apart instance of the recursive equation $\langle C(\bar{y}) = e' + C(\bar{y}_1) + \cdots + C(\bar{y}_k),\ \varphi'_2\rangle$, it holds that $\varphi_2 \wedge \varphi'_2 \wedge \bar{x}_i = \bar{y} \models l' - l \geq \check{d}$ (resp. $\varphi_2 \wedge \varphi'_2 \wedge \bar{x}_i = \bar{y} \models l - l' \geq \check{d}$) for any $\bar{x}_i$, where $\mathsf{nat}(l') \in e'$ is the renaming of $\mathsf{nat}(l)$.*

In practice, computing $\check{d}$ for a given $\mathsf{nat}(l) \in e$ can be done using integer programming tools. In what follows, when the conditions of Def. 4 hold for a given $\mathsf{nat}(l) \in e$, we say that it has a linear behavior. Moreover, when all $\mathsf{nat}(l) \in e$ have the same linear behavior (i.e., all increasing or all decreasing), we say that e has a linear behavior.

Example 3. For B, replacing $C(0, j, n)$ by the UB $\mathsf{nat}(n{+}j)$ computed in Ex. 2 results in $\langle B(j, i, n){=}\mathsf{nat}(n{+}j){+}B(j', i, n), \varphi_1\rangle$, where $\varphi_1 = \{j{<}i, j{+}1{\leq}j'{\leq}j{+}3\}$. Its renamed apart instance is $\langle B(j_r, i_r, n_r) = \mathsf{nat}(n_r{+}j_r) + B(j'_r, i_r, n_r), \varphi_2\rangle$ where $\varphi_2{=}\{j_r{<}i_r, j_r{+}1{\leq}j'_r{\leq}j_r{+}3\}$. Then, the formula $\varphi_1{\wedge}\varphi_2{\wedge}\{j'{=}j_r, i{=}i_r, n = n_r\} \models (n_r{+}j_r){-}(n{+}j) \geq \check{d}$ holds for $\check{d}{=}1$. Therefore, $\mathsf{nat}(n{+}j)$ increases linearly.

Let us intuitively explain how our method works by focusing on a single $\mathsf{nat}(l) \in e$ within the relation C. Assume that during the evaluation of an initial query $C(\bar{x}_0)$, $\mathsf{nat}(l)$ is evaluated to $\mathsf{nat}(l_1), \ldots, \mathsf{nat}(l_n)$ in n consecutive recursive calls, and suppose that it is linearly increasing at least by $\check{d}$, i.e., $l_{i+1} - l_i \geq \check{d}$ for all $1 \leq i \leq n-1$. As explained in Sec. 3, we can infer an expression $\mathsf{nat}(\hat{l})$ which is an UB for all $\mathsf{nat}(l_i)$, and a ranking function $\hat{f}_C$ such that $n \leq \hat{f}_C(\bar{x}_0)$. A tight approximation is the arithmetic sequence which starts from $\mathsf{nat}(\hat{l})$ and each time decreases by $\check{d}$. Clearly, the first element of this sequence is greater than $\mathsf{nat}(l_n)$, the second is greater than $\mathsf{nat}(l_{n-1})$, and so on.

However, a main problem is that, since $\hat{f}_C$ provides an over-approximation of the actual number of iterations, the sequence might go to negative values. This is because an imprecise (too large) $\hat{f}_C$ would lead to a too large decrease $\check{d} * \hat{f}_C(\bar{x}_0)$ and the smallest element $\mathsf{nat}(\hat{l}){-}\check{d} * \hat{f}_C(\bar{x}_0)$ (and possibly other subsequent ones) could be negative. Hence, the approximation would be unsound since the actual evaluations of such negative values are zero. We avoid this problem by viewing this sequence in a dual way: we start from the smallest value and in each step increase it by $\check{d}$. Since still the smallest values could be negative, we start from $\mathsf{nat}(\hat{l}{-}\check{d} * \hat{f}_C(\bar{x}_0))$ which is guaranteed to be positive and greater than or equal to $\mathsf{nat}(\hat{l}){-}\check{d} * \hat{f}_C(\bar{x}_0)$. The next definition uses this intuition to replace each nat by an expression that generates its corresponding sequence at the level of RR.

Definition 5. *Consider the CR C of Eq. 1 where e has a linear behavior. Let $\hat{f}_C(\bar{x}_0) = \mathsf{nat}(l')$ be its corresponding ranking function. We define its associated*

worst-case RR as $\langle P_C(x) = E_e + P_C(x-1) + \cdots + P_C(x-1)\rangle$ *where* E_e *is obtained from* e *by replacing each* $\mathsf{nat}(l) \in e$ *by* $\mathsf{nat}(\hat{l} - \check{d} * l') + x * \check{d}$.

Definition 5 generalizes Def. 3 and an equivalent theorem to Theorem 1 holds.

Theorem 2. *If* E *is a solution for* $P_C(x)$ *then* $E[x/\hat{f}_C(\bar{x}_0)]$ *is an UB for* $C(\bar{x}_0)$.

Example 4. Following Ex. 3, we have that $\check{d}$=1. Since $\mathsf{nat}(n_0{+}i_0{-}1)$ is an UB of the cost $\mathsf{nat}(n{+}j)$ accumulated in B, and $\hat{f}_B(j_0, i_0, n_0){=}\mathsf{nat}(i_0{-}j_0)$, according to Def. 5, we have $\langle P_B(x){=}\mathsf{nat}(n_0{+}i_0{-}1{-}(i_0-j_0)*1){+}x*1{+}P_B(x-1)\rangle$ which is solved by CAS to $P_B(x){=}\mathsf{nat}(n_0+j_0-1)*x{+}x*(x{+}1)/2$. Thus, $B(j_0, i_0, n_0) = P_B(x)[x/\mathsf{nat}(i_0-j_0)]$. Similarly, for A we obtain the RR $P_A(x) = (q{+}2x)(q/2{+}x){+}r(q{+}2x){+}q/2{+}x{+}P_A(x{-}1)$ where $q = \mathsf{nat}(i_0{-}2)$ and $r = \mathsf{nat}(n_0{-}1)$, which is solved to $P_A(x) = qx^2{+}qrx{+}rx{+}2/3x^3{+}rx^2{+}3/2x^2{+}5/6x{+}1/2q^2x{+}3/2qx$. Thus, $A(i_0, n_0) = P_A(x)[x/\mathsf{nat}((n_0 - i_0)/2)]$. Finally, for F, we obtain the UB $F(n_0) = y(4y^2{+}6zy{+}9y{+}6z{+}5)/6$, whereas [2] provides $2{*}\mathsf{nat}(n_0/2{+}1/2){*}z^2$, where $y = \mathsf{nat}(n_0/2)$ and $z = \mathsf{nat}(n_0 - 1)$, which is much less precise.

Our approach can be also applied when nat expressions are increasing or decreasing geometrically, i.e., when $\mathsf{nat}(l_{i+1}) \leq k * \mathsf{nat}(l_i)$ for some positive rational k called common ratio. This is the case in a CR like $\langle C(n){=}\mathsf{nat}(n){+}C(n/2), \{n \geq 1\}\rangle$, which is similar to what we obtain when analyzing the recursive implementation of merge-sort algorithm (mergesort has two recursive calls). In such geometric case, the counterpart condition to Def. 4 checks if there exists a minimum ratio $\check{k}$ such that $\varphi_2 \wedge \varphi_2' \wedge \bar{x}_i = \bar{y} \models l \geq \check{k} * l'$. Then, in a counterpart definition to Def. 5, we replace such $\mathsf{nat}(l) \in e$ by $\mathsf{nat}(\hat{l}) * \check{k}^{m-x} \in E_e$ where $m = \hat{f}_C(\bar{x}_0)$. Intuitively, we accumulate $\mathsf{nat}(\hat{l})$ when $x = \hat{f}_C(\bar{x}_0)$, and, at each subsequent step, the expression is geometrically reduced by the ratio. For the above CR, we obtain a linear UB $C(n_0) = 2 * \mathsf{nat}(n_0)$, whereas techniques described in [2, 9] would obtain $C(n_0) = \mathsf{nat}(n_0) * log_2(\mathsf{nat}(n_0 + 1))$. Note that here our approach improves even the complexity order. Using a similar construction, for merge-sort (see experiments), we are able to infer the upper bound $63\mathsf{nat}(a+1)log_2(\mathsf{nat}(2a-1)+1) + 50\mathsf{nat}(2a-1)$ on the number of executed instructions. For conciseness, rest of the paper formalizes the arithmetic case, but all results are directly applicable to geometric progressions as described above.

4.3 Non-deterministic Non-constant Cost Relations

Any approach for solving *CRs* that aims at being practical has to consider *CRs* with several recursive equations as shown in equation 2. This kind of *CRs* is very common during cost analysis and they mainly originate from conditional statements inside loops.

$$\begin{array}{l} \langle C(\bar{x}) = e_0, \varphi_0\rangle \\ \langle C(\bar{x}) = e_1 + C(\bar{x}_1) + \cdots + C(\bar{x}_{k_1}), \varphi_1\rangle \\ \quad\vdots \\ \langle C(\bar{x}) = e_h + C(\bar{x}_1) + \cdots + C(\bar{x}_{k_h}), \varphi_h\rangle \end{array} \tag{2}$$

For instance, the instruction `if (x[i]>0) {A;} else {B;}`, may lead to two

non-deterministic equations which accumulate the costs of `A` and `B`. This is because arrays are typically abstracted to their length and, hence, the guard `x[i]>0` is abstracted to `true`, i.e., we do not keep this information on the *CR*. Thus, $\varphi_0, \ldots, \varphi_h$ are not necessarily mutually exclusive. W.l.o.g., we assume that $k_1 \geq \cdots \geq k_h$, i.e., the first (resp. last) recursive equation has the maximum (resp. minimum) number of recursive calls among all equations. We also assume that $\hat{f}_C(\bar{x}_0) = \mathsf{nat}(l')$ is a *global* ranking function for this *CR*, i.e., a ranking function for all equations.

In non-deterministic *CRs*, the costs contributed by a chain of recursive calls might not be instances of the same cost expression, but rather of different expressions $e_1, \ldots, e_h$, i.e., the equations might interleave. Namely, we might apply one equation and for another call another different equation. The worst-case cost might originate from such interleaving sequences (see [2]). Thus, when inferring how a given $\mathsf{nat}(l) \in e_i$ changes, we have to consider subsequent instances of $\mathsf{nat}(l)$ which are not necessarily consecutive. For this, we infer an invariant that holds between two subsequent (not necessarily consecutive) applications of the same equation, similar to what [2] does, and then we compute the distance $\check{d}$ between its subsequent instances as in Def. 4 but considering this invariant.

As a first solution, similarly to Def. 5, for each expression e_i, we can generate a corresponding E_i by replacing each $\mathsf{nat}(l)$ by $\mathsf{nat}(\hat{l}-\check{d}*l')+x*\check{d}$ where $\check{d}$ is the distance for $\mathsf{nat}(l)$. Clearly, if e is a closed-form solution for the *RR* $P_C(x)=\max(E_1,\ldots,E_h)+P_C(x-1)+\ldots+P_C(x-1)$ with k_1 recursive calls, then $e[x/\hat{f}_C(\bar{x}_0)]$ is an UB for C (because in each application we take the worst-case). Unfortunately, *CAS* fail to solve *RRs* which involve (non-constant) max expressions. Therefore, this approach is not practical. Clearly, in the case that one of $e_1, \ldots, e_h$ is provable to be the maximum, this approach works since we can eliminate the max operator. Unfortunately, even comparing simple cost expressions is difficult and in many cases not feasible [1]. In what follows, we describe a practical solution to this problem, which is based on finding an expression E which does not include max and is always larger than or equal to $\max(E_1, \ldots, E_h)$. This way, we can replace the max by E and still get an UB for C.

First, observe that any cost expression (which does not include max) can be normalized to the form $\Sigma_{i=1}^{n} \Pi_{j=1}^{m_i} b_{ij}$ (i.e., sum of multiplications) where each b_{ij} is a *basic element* of the following form $\{r, \mathsf{nat}(l), n^{\mathsf{nat}(l)}, \log(\mathsf{nat}(l))\}$. We assume that all $e_1, \ldots, e_h$ of Eq. 2 are given in this form. For simplicity, we assume that all expressions have the same number of multiplicands, and all multiplicands have the same number of basic expressions (if not, we just add 1 in multiplication and 0 for sum). Now since e_i is in a normal form, the corresponding E_i will be also in a corresponding normal form. Given two cost expressions e_1 and e_2, and their corresponding E_1 and E_2, the following definition describes how to generate an expression E such that it is larger than (or equal to) both E_1 and E_2.

Definition 6. *Given two expressions* $E_i = a_{11} \cdots a_{1m_1} + \cdots + a_{n1} \cdots a_{1m_n}$ *and* $E_j = b_{11} \cdots b_{1m_1} + \cdots + b_{n1} \cdots b_{1m_n}$. *We define the generalization of* E_i *and* E_j *as* $E_i \sqcup E_j = c_{11} \cdots c_{1m_1} + \cdots + c_{n1} \cdots c_{1m_n}$ *where* $c_{ij} = b_{ij}$ *if we can prove that* $b_{ij} \geq a_{ij}$, $c_{ij} = a_{ij}$ *if we can prove that* $a_{ij} \geq b_{ij}$, *otherwise* $c_{ij} = a_{ij} + b_{ij}$.

Although we need to compare expressions when constructing $E_i \sqcup E_j$ (namely b_{ij} to a_{ij}), this comparison is on the basic elements rather than on the whole expression and hence it is far simpler. By construction, we guarantee that $E_i \sqcup E_j$ is always greater than or equal to both E_i and E_j. Clearly, the quality of $E_i \sqcup E_j$ (i.e., how tight it is) depends on the ordering of the summands and the elements of each summand (i.e., multiplicands) in both E_i and E_j. In order to obtain tighter bounds, we use some heuristics like ordering the elements inside each multiplication in increasing complexity in such a way that we always try to compare basic elements of the same complexity order. Besides, we try to compare basic elements that involve the same variable.

Definition 7. *Let C be the CR of Eq. 2, $\hat{f}_C(\bar{x}_0)$=$\mathsf{nat}(l')$ its corresponding ranking function, and E_i generated from e_i by replacing each $\mathsf{nat}(l)$∈e_i by $\mathsf{nat}(\hat{l}-d*l')+x*\check{d}$ where $\check{d}$ is the distance of $\mathsf{nat}(l)$. The corresponding worst-case RR is $\langle P_C(x) = E_1 \sqcup \cdots \sqcup E_h + P_C(x-1) + \cdots + P_C(x-1)\rangle$ with k_1 recursive calls.*

Theorem 3. *If E is a solution for $P_C(x)$ then $E[x/\hat{f}_C(\bar{x}_0)]$ is an UB for $C(\bar{x}_0)$.*

Example 5. Let us add the contrived recursive equation $B(j,i,n) = \mathsf{nat}(n+15) + B(j',i,n) + B(j'',i,n)$ $\{j<i, j'=j+1, j''=j+2\}$ to the *CR* B. It has two recursive calls and a non-deterministic choice for accumulating either $e_1 = \mathsf{nat}(n+j)$ or $e_2 = \mathsf{nat}(n+15)$. The function $f_B(j_0,i_0,n_0) = \mathsf{nat}(i_0 - j_0)$ is a ranking function for all equations. Next, we compute $E_1 \sqcup E_2$ where $E_1 = \mathsf{nat}(n_0 + i_0 - 1 - (i_0 - j_0)) + x$ and $E_2 = \mathsf{nat}(n_0 + 15 - (i_0 - j_0)) + x$. A naive generalization results in $\mathsf{nat}(n_0+i_0-1-(i_0-j_0)) + \mathsf{nat}(n_0+15-(i_0-j_0)) + x$, but syntactically analyzing the expressions and employing the above heuristics, we automatically obtain a tighter bound $\mathsf{nat}(n_0+j_0+15)+x$. Now we generate $\langle P_B(x) = \mathsf{nat}(n_0+j_0+15) + x + P_B(x-1) + P_B(x-1)\rangle$ which can be solved to $\langle P_B(x) = 2^x(q+2) - q - x - 2\rangle$ for $q = \mathsf{nat}(n_0 + j_0 + 15)$ and therefore $B(j_0,i_0,n_0) = P_B(x)[x/\mathsf{nat}(i_0-j_0)]$.

5 The Dual Problem: Lower Bounds

We now aim at applying the approach from Sec. 4 in order to infer *lower bounds*, i.e., under-approximations of the best-case cost. Such LBs are typically useful in granularity analysis to decide if tasks should be executed in parallel. This is because the parallel execution of a task incurs various overheads, and therefore the LB cost of the task can be useful to decide if it is worth executing it concurrently as a separate task. Due in part to the difficulty of inferring under-approximations, a general framework for inferring LBs from *CR* does not exist. When trying to adapt the UB framework of [2] to LB, we only obtain trivial bounds. This is because the minimization of the cost expression accumulated along the execution is in most cases zero and, hence, by assuming it for all executions we would obtain a trivial (zero) LB. In our framework, even if the minimal cost could be zero, since we do not assume it for all iterations, but rather only for the first one, the resulting LB is non-trivial.

Existing approaches typically assume that the length of chains of recursive calls depends on a single decreasing argument. We first propose a new technique to inferring LBs on the length of such chains, which does not have this restriction. Essentially, we add a counter to the equations in the *CR* and infer an invariant which involves this counter. The invariant is indeed the same one used later to obtain $\check{l}$. The minimum value of this counter when we enter a non-recursive case is a LB on the length of those chains.

Definition 8. *Given the CR of Eq. 2, we compute* $\check{f}_C(\bar{x}_0) = \mathsf{nat}(l)$ *which is a* lower bound on the length *of any chain of recursive calls when starting from* $C(\bar{x}_0)$ *in three steps: (1) Replace each head* $C(\bar{x})$ *by* $C(\bar{x}, lb)$ *and each recursive call* $C(\bar{x}_j)$ *by* $C(\bar{x}_j, lb{+}1)$*; (2) Infer an invariant* $\langle C(\bar{x}_0, 0) \rightsquigarrow C(\bar{x}, lb), \Psi\rangle$ *for the new CR; (3) Syntactically look for* $lb \geq l$ *in* $\Psi \wedge \varphi_0$ *(projected on* $\bar{x}_0$ *and* lb*).*

Example 6. Applying step (1) on the *CR* B results in $\langle B(j,i,n,lb) = 0, \{j \leq i\}\rangle$ and $\langle B(j,i,n,lb) = \mathsf{nat}(n{+}j){+}B(j',i,n,lb{+}1), \{j{<}i, j{+}1{\leq}j'{\leq}j{+}3\}\rangle$. The invariant Ψ for this *CR* is $\{j{-}j_0{-}lb{\geq}0, j_0{+}3lb{-}j{\geq}0, i{=}i_0, n{=}n_0\}$. Projecting $\Psi{\wedge}\{j{\geq}i\}$ on $\langle j_0, i_0, n_0, lb\rangle$ results in $\{lb{\geq}0, j_0{+}3lb{-}i_0{\geq}0\}$ which implies $lb \geq (i_0 - j_0)/3$. Similarly $\check{f}_C(k_0, j_0, n_0){=}\mathsf{nat}(n_0{+}j_0 - k_0)$ and $A(i_0, n_0){=}\mathsf{nat}(\frac{n_0 - i_0}{4})$.

We present the approach directly for the non-deterministic *CR* of Eq. 2. As in Def. 6, we can reduce the expressions $E_1, \ldots, E_h$ in order to get an expression which is guaranteed to be smaller than or equal to $\min(E_1, \ldots, E_h)$.

Definition 9. *Given the expressions* E_i *and* E_j *in Def. 6, we define their* reduction *as* $E_i \sqcap E_j = c_{11} \cdots c_{1m_1} + \cdots + c_{n1} \cdots c_{1m_n}$ *where* $c_{ij} = b_{ij}$ *if we can prove that* $b_{ij} \leq a_{ij}$*,* $c_{ij} = a_{ij}$ *if we can prove that* $a_{ij} \leq b_{ij}$*, otherwise* $c_{ij} = 0$*.*

The case of $c_{ij} = 0$ can be improved to obtain a tighter LB by relying on heuristics, similarly to what we have discussed in Sec. 4.3. As intuitively explained in Sec. 3, the main idea is to simulate each $\mathsf{nat}(l)$ by a sequence that starts from $\mathsf{nat}(\check{l})$ and increases in each iteration by the minimal distance $\check{d}$.

Definition 10. *Let* C *be the CR of Eq. 2 such that for each* $\mathsf{nat}(l) \in e_i$ *it holds that* $\check{l} \geq 0$*, and let* E_i *be the expression generated from* e_i *by replacing each* $\mathsf{nat}(l)$ *by* $\mathsf{nat}(\check{l}) + (x-1) * \check{d}$*. The corresponding best-case RR is* $\langle P_C(x) = E_1 \sqcap \cdots \sqcap E_h + P_C(x-1) + \cdots + P_C(x-1)\rangle$ *with* k_h *recursive calls.*

In the above definition, it can be observed that, for the sake of soundness, we require that for each $\mathsf{nat}(l)$ it holds that $\check{l} \geq 0$. Intuitively, when such expressions take negative values, by definition of nat, they evaluate to zero and there can be a sequence of zeros until the evaluation becomes positive. Our under-approximation would be unsound in this case, because it assumes as minimum value zero and then starts to increase it by the minimum distance. Thus, for some values, the approximation could be actually bigger than the actual value.

Theorem 4. *If* E *is a solution for* $P_C(x)$*, then* $E[x/\check{f}_C(\bar{x}_0)]$ *is a LB for* $C(\bar{x}_0)$*.*

#	UBs and LBs	T
1	$24\eta(a-1)^3+36\eta(a-1)^2+27\eta(a)^2+39\eta(a)\eta(a-1)+35\eta(a-1)+72\eta(a)+54$	1240
	$8\eta(a-1)^3+27\eta(a)^2+\frac{99}{2}\eta(a-1)^2+\frac{231}{2}\eta(a-1)+72\eta(a)+54$	1395
	$8\eta(a-2)^3+46\eta(a-2)^2+105\eta(a-2)+55\eta(a-1)+54$	204
2	$24\eta(c-1)^3+36\eta(c-1)^2+28\eta(c)^2+\eta(c-1)(40\eta(c)+35)+25\eta(c)+48\eta(b-1)^2+46\eta(b-1)+74$	1270
	$8\eta(c-1)^3+28\eta(c)^2+50\eta(c-1)^2+25\eta(c)+117\eta(c-1)+24\eta(b-1)^2+70\eta(b-1)+74$	1425
	$8\eta(c-2)^3+48\eta(c-2)^2+25\eta(c-1)+111\eta(c-2)+24\eta(b-2)^2+70\eta(b-2)+74$	247
3	$24\eta(a-1)^3+56\eta(a)\eta(a-1)^2+27\eta(a)^2+46\eta(a-1)^2+75\eta(a)+77\eta(a)\eta(a-1)+49\eta(a-1)+62$	3617
	$8\eta(a-1)^3+28\eta(a)\eta(a-1)^2+27\eta(a)^2+\frac{109}{2}\eta(a-1)^2+75\eta(a)+66\eta(a)\eta(a-1)+\frac{269}{2}\eta(a-1)+62$	3890
	$18\eta(a-2)^3+81\eta(a-2)^2+75\eta(a-1)+144\eta(a-2)+62$	415
4	$25\eta(b)\eta(c)\eta(c-1)+30\eta(b)\eta(c)+16\eta(b)+6$	130
	$25/2\eta(b)\eta(c-b)^2+25\eta(b)^2\eta(c-b)+25/2\eta(b)^3+40\eta(b)^2+135/2\eta(b)\eta(c-b)+87/2\eta(b)+6$	200
	$21/2\eta(b-1)^2+21\eta(b-1)\eta(c-b)+53/2\eta(b-1)+6$	60

#	UBs and LBs	T
5	$19\eta(a-1)^2+25\eta(a-1)+7$	44
	$19/2\eta(a-1)^2+69/2\eta(a-1)+7$	63
	$18\eta(a-2)+7$	10
7	$27\eta(a-1)^2+16\eta(a-1)+9$	103
	$27/2\eta(a-1)^2+59/2\eta(a-1)+9$	120
	$13/2\eta(a-2)^2+45/2\eta(a-2)+9$	25
9	$34\eta(a)\eta(a-1)+12\eta(a)+8$	174
	$17\eta(a)^2+29\eta(a)+8$	197
	$8\eta(a-1)^2+20\eta(a-1)+8$	24

#	UBs and LBs	T
6	$43\eta(a)\eta(2a-3)+53\eta(2a-3)+17$	2127
	$63\eta(a+1)log_2(\eta(2a-1)+1)+50\eta(2a-1)$	2100
	0	40
8	$16\eta(a)^2+27\eta(a-1)^2+31\eta(a)+10\eta(a-1)+25$	200
	$27/2\eta(a)^2+27\eta(a-1)^2+10\eta(a-1)+67/2\eta(a)+25$	247
	$5/2\eta(a-1)^2+10\eta(a-2)+67/2\eta(a-1)+25$	60
10	$2^{\eta(a-1)}(5\eta(a-1)+21)+5\eta(a)-5\eta(a-1)-7$	104
	$31*2^{\eta(a-1)}+5\eta(a)-5\eta(a-1)-17$	144
	$31*2^{\eta(a-2)}+5\eta(a-1)-5\eta(a-2)-17$	34

Table 1: 1. DetEval(a) 2. LinEqSolve(a,b,c) 3. MatrixInv(a) 4. MatrixSort(a,b,c) 5. InsertSort(a) 6. MergeSort(a) 7. SelectSort(a) 8. PascalTriangle(a) 9. BubbleSort(a) 10. NestedRecIter(a).

Example 7. Consider the LBs on iterations of Ex. 6. Since $C(k_0, j_0, n_0)$ accumulates a constant cost 1, its LB cost is $\mathsf{nat}(n_0+j_0-k_0)$. We now replace the call $C(0, j, n)$ in B by its LB $\mathsf{nat}(n+j)$ and obtain the equation: $B(j, i, n) = \mathsf{nat}(n+j) + B(j', i, n) \quad \{j<i, j+1\leq j'\leq j+3\}$. Notice the need of the soundness requirement in Th. 3, i.e., $\mathsf{nat}(n+j)\geq 0$. E.g., when evaluating $B(-5, 5, 0)$ the first 4 instances of $\mathsf{nat}(n+j)$ are zero since they correspond to $\mathsf{nat}(-5), \ldots, \mathsf{nat}(-1)$. Therefore, it would be incorrect to start accumulating from 0 with a difference 1 at each iteration. After solving A and B in the same way, the computed final LB for $F(n)$ is: $\frac{1}{3}\mathsf{nat}(n)\mathsf{nat}(\frac{n}{4}-1)+\frac{1}{18}\mathsf{nat}(\frac{n}{4}-1)\mathsf{nat}(\frac{n}{4}-1)+\frac{1}{6}\mathsf{nat}(\frac{n}{4}-1)$.

6 Experiments and Conclusions

We have implemented our approach in COSTA, a COSt and Termination Analyzer for Java bytecode. The obtained *RRs* are solved using MAXIMA [12] or PURRS [4]. As benchmarks, we use classical examples from complexity analysis and numerical methods: DetEval evaluates the determinant of a matrix; LinEqSolve solves a set of linear equations; MatrixInverse computes the inverse of an input matrix; MatrixSort sorts the rows in the upper triangle of a matrix; InsertSort, SelectSort, BubbleSort, and MergeSort implement sorting algorithms; PascalTriangle computes and prints Pascal's Triangle; NestedRecIter is an interesting programming pattern we found in the Java libraries with a spacial form

of nested loops that uses recursion and a simple iteration for loop. Our implementation (and examples) can be tried out at `http://costa.ls.fi.upm.es` by enabling the option `series` in the manual configuration.

Table 1 illustrates the accuracy and efficiency on the above benchmarks using the cost model "*number of executed (bytecode) instructions*". We abbreviate nat(x) as $\eta(x)$. The second column shows: in the top row the UB obtained by [2], next the UB obtained by us and at the bottom our LB. Unfortunately, there are no other cost analysis tools for imperative languages available to compare experimentally to (e.g., SPEED [9]). As regards UBs, we improve the precision over [2] in all benchmarks. This improvement, in all benchmarks except MergeSort and NestedRecIter, is due to nested loops were the inner loops bounds depend on the outer loops counters. In these cases, we accurately bound the cost of each iteration of the inner loops, rather than assuming the worst-case cost. For MergeSort, we obtain a tight bound in the order of $a{*}log(a)$. Note that [2] could obtain $a{*}log(a)$ only for simple cost models that count the visits to a specific program point but not for number of instructions, while ours works with any cost model. For NestedRecIter, we improve the complexity order over [2] from $a{*}2^a$ to 2^a. As regards LBs, it can be observed from the last row of each benchmark that we have been able to prove the positive nat condition and obtain non-trivial LBs in all cases except MergeSort. For MergeSort, the lower bound on loop iterations is a logarithmic which cannot be inferred by our linear invariant generation tool and hence we get trivial bound 0. Note that for InsertSort we infer a linear LB which happens when the array is sorted. Column T shows the time (in milliseconds) to compute the bounds from the generated CR. Our approach is slightly slower than [2] mainly due to the overhead of connecting COSTA to the external CAS.

7 Conclusions

When comparing our approach (for UBs) to [9], since the underlying cost analysis framework is fundamentally different from ours, it is not possible to formally compare the resulting upper bounds in all cases. However, by looking at small examples, we can see why our approach can be more precise. For instance, in [9] the worst-case time usage $\sum_{i=1}^{n} i$ is over-approximated by n^2, while our series-based approach is able to obtain the precise solution. For such polynomial cases, the approach of [10] can compute also the exact solution. However, this approach is restricted to univariate polynomial bounds, while ours can be applied to obtain general polynomial, exponential and logarithmic bounds as well.

Finally, to conclude, we have proposed a novel approach to infer precise upper/lower bounds of *CRs* which, as our experiments show, achieves a very good balance between the accuracy of our analysis and its applicability. The main idea is to automatically transform *CRs* into a simple form of worst-case/best-case *RRs* that *CAS* can accurately solve to obtain upper/lower bounds on the resource consumption. The required transformation is far from trivial since it requires transforming non-deterministic equations involving multiple increas-

ing/decreasing arguments into deterministic equations with a single decreasing argument.

Acknowledgements. 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 *DOVES* project, the HI2008-0153 (Acción Integrada) project, the UCM-BSCH-GR58/08-910502 Research Group and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS* project.

References

1. E. Albert, P. Arenas, S. Genaim, I. Herraiz, and G. Puebla. Comparing cost functions in resource analysis. In *FOPARA'09*, volume 6234 of *LNCS*. Springer, 2010.
2. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 2010. To appear.
3. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *ESOP*, LNCS 4421, pages 157–172. Springer, 2007.
4. R. Bagnara, A. Pescetti, A. Zaccagnini, and E. Zaffanella. PURRS: Towards Computer Algebra Support for Fully Automatic Worst-Case Complexity Analysis. Technical report, 2005. `arXiv:cs/0512056` available from http://arxiv.org/.
5. Amir M. Ben-Amram. Size-change termination, monotonicity constraints and ranking functions. In *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 109–123. Springer, 2009.
6. P. Feautrier C. Alias, A. Darte and L. Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *SAS*, Lecture Notes in Computer Science. Springer, 2010.
7. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. POPL*. ACM, 1978.
8. K. Crary and S. Weirich. Resource bound certification. In *POPL'00*. ACM Press, 2000.
9. S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139. ACM, 2009.
10. J. Hoffmann and M. Hofmann. Amortized Resource Analysis with Polynomial Potential. In *Proc. of ESOP*, volume 6012 of *Lecture Notes in Computer Science*, pages 287–306. Springer, 2010.
11. Kim Marriot and Peter Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.
12. Maxima.sourceforge.net. Maxima, a Computer Algebra System. Version 5.21.1 (2009). http://maxima.sourceforge.net/.
13. B. Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9), 1975.