On the Inference of Resource Usage Upper and Lower Bounds

Elvira Albert, Complutense University of Madrid, Spain
Samir Genaim, Complutense University of Madrid, Spain
Abu Naser Masud, Technical University of Madrid, Spain

Cost analysis aims at determining the amount of resources required to run a program in terms of its input data sizes. The most challenging step is to infer the cost of executing the *loops* in the program. This requires bounding the number of iterations of each loop and finding tight bounds for the cost of each of its iterations. This article presents a novel approach to infer *upper* and *lower* bounds from *cost relations*. These relations are an extended form of standard *recurrence equations* which can be non-deterministic, contain inexact size constraints and have multiple arguments that increase and/or decrease. We propose novel techniques to automatically transform cost relations into *worst-case* and *best-case* deterministic one-argument recurrence relations. The solution of each recursive relation provides a precise upper-bound and lower-bound for executing a corresponding loop in the program. Importantly, since the approach is developed at the level of the cost equations, our techniques are programming language independent.

Categories and Subject Descriptors: D.4.8 [**Software Engineering**]: Metrics —*Complexity measures*; *Performance measures* ; H.4.0 [**Theory of Computation**]: Semantics of Programming Languages —*Program analysis*

Additional Key Words and Phrases: Upper Bounds, Lower Bounds, Static Cost Analysis

1. INTRODUCTION

Having available information about the computational cost of programs execution, i.e., the amount of resources that the execution will require, is clearly useful for many different purposes, like for performance debugging, resource usage verification/certification and for program optimization (see, e.g., Albert et al. [2012] and its references). In general, reasoning about execution cost is difficult and error-prone. This is specially the case when programs contain *loops* (either as iterative constructs or recursions), since one needs to reason about the number of iterations that loops will perform and the cost of each of them.

Static cost analysis [Wegbreit 1975], a.k.a. *resource usage analysis*, 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 by Wegbreit dates back to 1975 [Wegbreit 1975]. It consists of two phases. In the first phase, 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. The *cost model* states the resource we are measuring. Cost analyzers are usually parametric on the cost model, e.g., cost models widely used are the number of executed instructions, number of calls to methods, amount of memory allocated, etc.

This article is a revised and extended version of [Albert et al. 2011d].

Email addresses: elvira@sip.ucm.es (Elvira Albert), samir.genaim@fdi.ucm.es (Samir Genaim), anm@clip.dia.fi.upm.es (Abu Naser Masud)

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 0000 ACM 1529-3785/0000/-ARTA $15.00

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

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

(a) Running Example

$F(q)$	$= A(0,q)$	$\{\}$
$A(i,q)$	$= 0$	$\{i \geq q\}$
$A(i,q)$	$= B(0,i,q)+A(i',q)$	$\{i+1 \leq q, i+2 \leq i' \leq i+4\}$
$B(j,i,q)$	$= 0$	$\{j \geq i\}$
$B(j,i,q)$	$= C(0,j,q)+B(j',i,q)$	$\{j+1 \leq i, j+1 \leq j' \leq j+3\}$
$C(k,j,q)$	$= 0$	$\{k \geq q+j\}$
$C(k,j,q)$	$= 1+C(k',j,q)$	$\{k' = k+1, k+1 \leq q+j\}$

(b) *CRs* for Memory Consumption

Fig. 1: Running Example and its Cost Relations

Example 1.1 (*running example*). Let us motivate our work on the contrived example depicted in Figure 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 Albert et al. [2012] generates the *CR* which appears in Figure 1b. This cost model estimates the number of objects allocated in the memory. Note that in this paper we ignore the effect that compiler optimizations might have on the resource consumption, handling this is out of the scope of this paper. 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 $\langle k, j, q\rangle$ such that $k + 1 \leq q + j$ is 1 (one object) plus that of executing the loop with $\langle k', j, q\rangle$ where $k' = k + 1$. The recursive equation for B states that executing the loop with $\langle j, i, q\rangle$ costs as executing $C(0, j, q)$ plus executing the same loop with $\langle j', i, q\rangle$ where $j + 1 \leq j' \leq j + 3$. While, in the Java program, j' can be either $j + 1$ or $j + 3$, due to the use of static analysis, the case for $j + 2$ is added in order to over approximate $j' = j + 1 \vee j' = j + 3$ by the polyhedron $j + 1 \leq j' \leq j + 3$ [Cousot and Halbwachs 1978].

In general, the first phase of cost analysis (i.e., the process of generating *CRs* from the program) heavily depends on the programming language in which the program is written. Multiple analysis have been developed for different paradigms including for functional [Wegbreit 1975; Le Metayer 1988; Rosendahl 1989; Wadler 1988; Sands 1995; Benzinger 2004; Luca et al. 2006; Hoffmann et al. 2011], logic [Debray and Lin 1993; Navas et al. 2007], and imperative [Adachi et al. 1979; Albert et al. 2012] programming languages. Importantly, the resulting *CRs* are a common target of cost analyzers, i.e., they abstract away the particular features of the original programming language and (at least conceptually) have the same form.

Though *CRs* are simpler than the programs they originate from, since all variables are of integer type, in several respects they are not as static as one would expect from the result of a static analysis. One reason is that they are recursive and thus we may need to iterate for computing their value for a given concrete input. Another reason is that even for deterministic programs, it is well known that the loss of precision introduced by the size abstraction may result in *CRs* which are non-deterministic. This happens in the above example, e.g., because j' can be either $j + 1$, $j + 2$ or $j + 3$, and they become non-deterministic choices when applying the second equation defining B. In general, for finding the worst-case and best-case cost we may need to compute and compare (infinitely) many results. For both reasons, it is clear that it is interesting to compute *closed-form* bounds for the CR, whenever this is possible, i.e., bounds which are not in recursive form. Our work focuses on such second phase of cost analysis: once CRs are generated, analyzers try to compute closed-forms bounds for them. Two main approaches exist:

(1) Since CRs are syntactically quite close to *recurrence relations* (RRs), most cost analysis frameworks rely on existing *Computer Algebra Systems* (CAS) for finding closed-forms. The main problem of this approach is that CAS only accept as input a small subset of CRs, namely they require that the equations have a single argument, one base-case and one recursive case. Thus, CAS are very precise when applicable, but handle only a restricted class of CRs, namely only some of those which have an exact solution. In practice, this seldom happens.
(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 which are able to obtain useful approximations of the best-case cost have not yet been developed.

Example 1.2. Let us see the application of the above approaches to our running example. As regards 1, we cannot use a CAS since an exact solution does not exist. Recall that, in the cost relation B, variable j' can increase by one, by two or by three at each iteration. Therefore, an exact cost function which captures the cost of any possible execution does not exist. As regards 2, we now try to obtain UBs and LBs for the relation. 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 Albert et al. [2011b] assumes the worst-case cost for all loop iterations. For instance, an UB on the cost of any iteration of B is q_0+i_0-1, where q_0 and i_0 are respectively the initial values for q and i. This corresponds to the memory allocation (number of objects) 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). However, approximating the cost of iterations by the best-case cost is not useful in order to obtain LBs since it leads to a trivial (useless) result, namely 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 [Crary and Weirich 2000]. Lack of precision can make the system fail to prove the resource usage requirements imposed by the software client. For example, it makes much difference the precision we gain by inferring $\frac{1}{2}i^2$ instead of i^2 for a given method. With the latter UB, an execution with $i{=}10$ will be rejected if we have only memory for 50 objects, while with the former one it is accepted. LBs are used for scheduling the distribution of tasks in parallel execution in such a way that it is not worth parallelizing a task unless its (lower-bound) resource consumption is sufficiently large. Precision is essential here to achieve a satisfactory scheduling.

1.1. Summary of Contributions

The main achievement in this paper is the seamless integration of both approaches described above, so that we get the best of both worlds: precision as (1), whenever possible, while applicability as close to (2) as possible. For UBs, intuitively, the precision gain stems from the fact that, instead of assuming the worst-case cost for all iterations,

Worst-Case Recurrence Relation
$P_A(N) = \frac{1}{2} * (\Vert i_0{-}1\Vert + (\Vert \frac{q_0-i_0}{2} \Vert - N + 1) * 2) * (2 * \Vert q_0{-}1 \Vert + \Vert i_0{-}1 \Vert + (\Vert \frac{q_0-i_0}{2} \Vert - N + 1) * 2 + 1) + P_A(N-1)$
$P_B(N) = \Vert q_0 + j_0 - 1\Vert + (\Vert i_0 - j_0 \Vert - N + 1) * 1 + P_B(N-1)$
$P_C(N) = 1 + P_C(N-1)$

Best-Case Recurrence Relation
$P_A(N) = \frac{1}{2} * (\Vert \frac{i_0}{3} \Vert + (\Vert \frac{q_0-i_0}{4} \Vert - N) * \frac{2}{3}) * (\Vert \frac{i_0}{3} \Vert + (\Vert \frac{q_0-i_0}{4} \Vert - N) * \frac{2}{3} + 2 * \Vert q_0 - \frac{1}{2} \Vert) + P_A(N-1)$
$P_B(N) = \Vert q_0 + j_0 \Vert + (\Vert \frac{i_0-j_0}{3} \Vert - N) * 1 + P_B(N-1)$
$P_C(N) = 1 + P_C(N-1)$

Fig. 2: RRs automatically obtained from CRs in Fig. 1b. P_A, P_B and P_C are the RRs that correspond to CRs A, B and C. Here, $\|l\| = \max(l, 0)$. The parameter N corresponds to the number of iterations of the corresponding loop, the rest (such as i_0, j_0 and q_0) are constants symbols.

we infer tighter bounds on each of them in an automatic way and then approximate the summation of a corresponding sequence. We do so by generating a novel form of *worst-case* and *best-case* RRs which can be solved by CAS. For example, we will demonstrate along the paper that the *worst-case* and *best-case* RRs shown in Figure 2 can be systematically generated from the CRs of the running example in Figure 1b. The exact solution of such RR is guaranteed to be a precise UB and LB respectively of the original CR. Technically, the main contributions of this article are:

— We propose an automatic transformation from a CR with multiple arguments and a single recursive equation, which possibly accumulates a non-constant cost at each application, into a worst-case/best-case single-argument RR that can be solved using CAS. Soundness of the transformation requires that we are able to infer the so-called *progression parameters*, which describe the relation between the contributions (to the total cost) of two consecutive applications of the recursive equations.
— As a further step, we consider CRs in which we have several recursive equations defining the same relation. We propose an automatic transformation into a worst-case/best-case RR that can be solved using CAS.
— 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, with some additional conditions for soundness.
— We report on a prototype implementation within the COSTA system [Albert et al. 2009]. Preliminary experiments on Java (bytecode) programs confirm the good balance between the accuracy and applicability of our analysis.

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.

1.2. Organization

The rest of the article is organized as follows. Section 2 recalls some preliminary notions and introduces some notations. It formalizes the notion of *cost relation* and *single-argument recurrence relation*. Section 3 informally explains the approximation that our analysis aims at achieving.

In Section 4, we present the main technical part of the article, which describes how to transform a CR into a RR for the sake of inferring UBs. We split the presentation in three parts. In Section 4.1, we first consider the simplest possible form of CR, i.e., CRs with a single recursive equation that accumulates constant costs. This allows us to focus on the non-trivial problem of transforming equations that have multiple arguments to single-argument ones. In a next step, Section 4.2 considers that the accumulated cost might be non-constant, and it proposes an automatic way to achieve a single-argument RR for this more complex case. The constant case becomes an instance of it. Finally, in Section 4.3, we handle CRs with multiple recursive equations. Once the transformation is performed, we can rely on accurate CAS to obtain UBs from the corresponding RR.

Section 5 presents the dual problem of inferring LBs from the CRs. While the problem of inferring UBs on the number of iterations of loops had been already solved in the literature and we have just adopted the solutions, the problem of inferring LBs on the number of iterations is new. The main focus of this section is then on obtaining such LBs on loop iterations. Given such bounds, the techniques proposed in Section 4 dually apply to the automatic inference of LBs from CRs.

Section 6 describes the implementation of our approach and evaluates it on a series of benchmarks programs that contain loops whose cost is not constant, e.g., sorting algorithms. In these cases, the fact that we accurately approximate the cost of each loop iteration is reflected in the more precise UB that we can obtain. Finally, Section 7 compares our approach to existing work and Section 8 concludes. Proofs of all the theorems described in this paper are provided in Appendix A.

2. PRELIMINARIES

In this section, we fix some notation and recall preliminary definitions. The sets of integer, rational, non-negative integer, and non-negative rational values are denoted respectively by $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{Z}_+$, and $\mathbb{Q}_+$. A *linear expression* over $\mathbb{Z}$ has the form $v_0 + v_1x_1 + \ldots + v_nx_n$, where $v_i \in \mathbb{Q}$, and $x_1, \ldots, x_n$ are variables that range over $\mathbb{Z}$. A *linear constraint* over $\mathbb{Z}$ has the form $l_1 \leq l_2$, where l_1 and l_2 are linear expressions. We use $l_1 = l_2$ as an abbreviation for $l_1 \leq l_2 \wedge l_2 \leq l_1$. We use $\bar{t}$ to denote a sequence of entities $t_1, \ldots, t_n$, and $vars(t)$ to refer to the set of variables that appear syntactically in an entity t. We use φ, ψ and Ψ (possibly subscripted and/or superscripted) to denote (conjunctions of) linear constraints. A set of linear constraints $\{\varphi_1, \ldots, \varphi_n\}$ denotes the conjunction $\varphi_1 \wedge \cdots \wedge \varphi_n$. A solution for φ is an assignment $\sigma : vars(\varphi) \mapsto \mathbb{Z}$ for which φ is satisfiable. The set of all solutions (assignments) of φ is denoted by $[\![\varphi]\!]$. We use $\varphi_1 \models \varphi_2$ to indicate that $[\![\varphi_1]\!] \subseteq [\![\varphi_2]\!]$. We use $\sigma(t)$ or $t\sigma$ to bind each $x \in vars(t)$ to $\sigma(x)$, $\exists\bar{x}.\varphi$ for the elimination of the variables $\bar{x}$ from φ, and $\bar{\exists}\bar{x}.\varphi$ for the elimination of all variables but $\bar{x}$ from φ. We use $t[X/Y]$ to replace all occurrences of X by Y in a syntactic object t.

2.1. Cost Relations: The Common Target of Cost Analyzers

Let us now recall the general notion of CRs as defined by Albert et al. [2011b]. The basic building blocks of CRs are the so-called *cost expressions* which are generated using this grammar:

$$e ::= \ r \ \mid \ \|l\| \ \mid \ e + e \ \mid \ e * e \ \mid \ e^r \ \mid \ \log(\|l\| + 1) \ \mid \ n^{\|l\|} \ \mid \ \max(S)$$

where $r \in \mathbb{Q}_+$, $n \in \mathbb{Q}_+$ and $n \geq 1$, l is a linear expression over $\mathbb{Z}$. S is a nonempty set of cost expressions and $\|.\| : \mathbb{Q} \to \mathbb{Q}_+$ is defined as $\|l\| = \max(\{l, 0\})$. Note that $\|.\|$ is read as "nat" (for natural numbers) and $\|l\|$ as "nat of l". Importantly, linear expressions are always wrapped by $\|.\|$ in order to avoid negative evaluations. For instance, as we will see later, an UB for $C(k, j, q)$ in Figure 1b is $\|q_0 + j_0 - k_0\|$. Without the use of $\|.\|$, the evaluation of $C(5, 5, 11)$ results in the negative cost -1 which must be lifted to zero, since it corresponds to an execution in which the `for` loop is not entered (i.e.,

$k \geq q + j$). Moreover, $\|.\|$ expressions provide a compact representation for piecewise functions, in which each $\|l\|$ is represented by two cases for $l \geq 0$ and $l < 0$. Observe that cost expressions are monotonic in their $\|.\|$ sub-expressions, i.e., replacing $\|l\| \in e$ by $\|l'\|$ such that $l' \geq l$ results in a cost expression e' such that $e' \geq e$. This property is fundamental for the correctness of our approach.

Definition 2.1 (*Cost Relation*). A CR C is defined by a set of equations of the form $\mathcal{E} \equiv \langle C(\bar{x}) = e + \sum_{i=1}^{m} D_i(\bar{y}_i) + \sum_{j=1}^{n} C(\bar{z}_j), \varphi \rangle$ where $m \geq 0$; $n \geq 0$; 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 $vars(\mathcal{E})$.

W.l.o.g., in what follows we formalize our method by making two assumptions on the CR defined above:

(1) *Direct recursion:* 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 [Albert et al. 2011b]; and
(2) *Standalone cost relations:* CRs do not depend on any other CR, i.e., the equations do not contain external calls, and thus have the form $\langle C(\bar{x}) = e + \sum_{j=1}^{n} C(\bar{z}_j), \varphi \rangle$.

The second assumption can be made 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 Figure 1b is solved to the UB $\|q_0 + j_0 - k_0\|$. Then, we continue by substituting the computed bounds in the equations which call such relation, which in turn become standalone. For instance, substituting the above UB in the relation B results in the equation $\langle B(j, i, q) = \|q + j\| + B(j', i, q), \{j < i, j+1 \leq j' \leq j+3\} \rangle$. This operation is repeated until no more CRs need to be solved. In what follows, CR refers to a standalone CR in direct recursive form, unless we explicitly state otherwise.

The evaluation of a CR C for a given valuation $\bar{v}$ (integer values), denoted $C(\bar{v})$, is based on the notion of evaluation trees [Albert et al. 2011b], which is similar to SLD trees in the context of Logic Programming [Kowalski 1974]. The set of evaluation trees for $C(\bar{v})$ is defined as follows

$$\mathcal{T}(C(\bar{v})) = \left\{ Tree(\sigma(e), [T_1, \ldots, T_n]) \left\| \begin{array}{l} (1)\ \langle C(\bar{x}) = e + \sum_{j=1}^{n} C(\bar{z}_j), \varphi \rangle \in C \\ (2)\ \sigma \in [\![\bar{v} = \bar{x} \wedge \varphi]\!] \\ (3)\ T_j \in \mathcal{T}(C(\sigma(\bar{z}_j))) \end{array} \right. \right\}$$

A possible evaluation tree for $C(\bar{v})$ is generated as follows: In (1) we chose a matching equation from those defining the CR C; In (2) we chose a solution σ for $\bar{v} = \bar{x} \wedge \varphi$, which indicates that the chosen equation is applicable; In (3) we let T_j be an evaluation tree for $C(\sigma(\bar{z}_j))$; and then we construct an evaluation tree $Tree(\sigma(e), [T_1, \ldots, T_n])$ for $C(\bar{v})$, which has $\sigma(e)$ as the root and $T_1, \ldots, T_n$ as sub-trees. Note that due to the non-deterministic choices in (1) and (2) we might have several evaluation trees for $C(\bar{v})$. Note also that trees might be infinite. The sum of all nodes of $T \in \mathcal{T}(C(\bar{v}))$ is denoted by $sum(T)$, and the set of answers for $C(\bar{v})$ is defined as $answ(C(\bar{v})) = \{sum(T) \mid T \in \mathcal{T}(C(\bar{v}))\}$. A closed-form function $C^*(\bar{x}_0) = e$ is an UB (resp. LB) for C, if for any valuation $\bar{v}$ it holds that $C^*(\bar{v}) \geq \max(answ(C(\bar{v})))$ (resp. $C^*(\bar{v}) \leq \min(answ(C(\bar{v})))$). Note that 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. This makes it possible to use CR to infer both UBs and LBs.

Example 2.2. Let us evaluate $B(0, 3, 3)$. The only matching equation is the second one for B. We choose an assignment σ. Here we have a non-deterministic choice for selecting the value of j' which can be 1, 2 or 3. 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.

Our approach for inferring UBs (resp. LBs) for a CR C heavily relies on over (resp. under) approximating the lengths of paths in the corresponding evaluation trees. The length of a path is defined as the number of edges from the root to a leaf. Note that a path in an evaluation tree is associated with a chain of calls from which its nodes have been generated, and thus, bounding the length of such chains also bounds the length of the corresponding paths. Intuitively, the length of a path indicates how many times we have applied a recursive equation.

2.2. Single-Argument Recurrence Relations

It is fundamental for this paper to understand the differences between CRs and RRs. The following have been identified as the main differences [Albert et al. 2011b], 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. The number of evaluation steps (i.e., recursive calls performed) is often a function of such several arguments (e.g., in A it depends on i and q).
(2) CRs often contain *inexact size relations*, e.g., variables range over an interval $[a, b]$ (e.g., variable j' in B). Thus, for a given input, 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 not mutually exclusive equations (and thus *non-deterministic*). This will be further explained in Section. 4.3.

As a consequence of point (2) and (3) above, an exact solution often does not exist and hence CAS just cannot be used in such cases. But, even if a solution exists, CAS do not accept all CRs as a valid input. Below, we define a class of RRs that CAS can handle.

Definition 2.3 (*Single-argument* RR). A *single-argument* RR P is defined by at most one recursive equation $\langle P(N) = E + n * P(N-1)\rangle$ where E is a function on N (and might have constant symbols) and $n \in \mathbb{Z}_+$ refers to the number of recursive calls, and a base-case equation $\langle P(0) = \lambda\rangle$ where λ is a constant symbol representing the value of the base-case.

A closed-form solution for $P(N)$, if exists, is an arithmetic expression that depends only on the variable N, the base-case constant symbol λ, and might include constant symbols that appear in E. Depending on the number of recursive calls n in the recursive equation and the expression E, such solution can be of different complexity classes (exponential, polynomial, etc.). Note that the notion of evaluation trees for CRs can be easily adapted for RRs. The only difference is that for RRs, the call $P(v)$ has only one evaluation tree which is also complete (i.e., all levels are complete), while for CRs, the call $C(\bar{v})$ might have multiple trees with any shape.

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 Albert et al. [2011b] with an example.

Consider a CR in its simplest form with one base-case equation and one recursive equation with a single recursive call:

$$\langle C(x) = 0, \{x < 0\}\rangle$$
$$\langle C(x) = \|x\| + C(x'), \{x - 3 \leq x' \leq x - 1, x \geq 0\}\rangle$$

Any evaluation tree for $C(x_0)$, where x_0 denotes the initial value of x, consists of a single path, where the leaf node has value 0 (for the base-case), and the internal nodes, that correspond to applying the recursive equation, have values $\langle e_1, \ldots, e_\kappa\rangle$. Note that the value of each e_i is determined by $\|x\|$. Clearly, $C(x_0)$ has many evaluation trees, depending on the choice of x' in each recursive call. Moreover, the depth κ of each tree, and the value of each e_i might be different from one tree to another. Our challenge is to accurately estimate the cost of C for any input, i.e., to infer a function $C^{ub}(x_0)$ (resp. $C^{lb}(x_0)$) such that $C^{ub}(x_0)$ is larger (resp. $C^{lb}(x_0)$ is smaller) than the cost $e_1 + \cdots + e_\kappa$ associated to each evaluation tree.

CAS aim at obtaining the exact cost function, and thus it is not possible to apply it to the above example since $C(x_0)$ has multiple solutions. Instead, the goal of static cost analysis is to infer approximations in terms of closed-form UBs/LBs for C. Our starting point is the general approximation for UBs proposed by Albert et al. [2011b] which has two dimensions:

(1) *Number of applications of the recursive case:* the first dimension is to infer an UB $\hat{\kappa}$ on the number of times the recursive equations can be applied (which, for loops, corresponds to the number of iterations). This bounds the depth of the tree; and
(2) *Cost of applications:* the second dimension is to infer an UB $\hat{e}$ on the cost of all loop iterations, i.e., $\hat{e} \geq e_i$ for all i.

For the above example it infers $\hat{\kappa} = \|x_0 + 1\|$ and $\hat{e} = \|x_0\|$. Then, $C^{ub}(x_0) = \hat{\kappa} * \hat{e} = \|x_0\| * \|x_0 + 1\|$ is guaranteed to be an UB for C. If the relation C had two recursive calls, then the UB would be an exponential function of the form $2^{\hat{\kappa}} * \hat{e}$. The most important point to notice is that the cost of all iterations e_i is approximated by the same worst-case cost $\hat{e}$, which is the source of imprecision of this approach that we will improve on. Technically, the above two dimensions are solved as follows:

(1) The first dimension is solved by inferring a *ranking function* $\hat{f}_C$, such that for any recursive equation $\langle C(\bar{x}) = e + C(\bar{x}_1) + \cdots + C(\bar{x}_m), \varphi\rangle$ in the CR, it satisfies $\varphi \models \hat{f}_C(\bar{x}) \geq \hat{f}_C(\bar{x}_i) + 1 \wedge \hat{f}_C(\bar{x}) \geq 0$ for all $1 \leq i \leq m$. This guarantees that when evaluating $C(\bar{x}_0)$, the length of any chain of calls to C cannot exceed $\hat{f}_C(\bar{x}_0)$. Thus, $\hat{f}_C$ bounds the length of such chains.
(2) The second dimension is solved by first inferring an *invariant* $\langle C(\bar{x}_0) \leadsto C(\bar{x}), \Psi\rangle$, where Ψ is a set of linear constraints, which describes the relation between the values that $\bar{x}$ can take in any call to C and the initial values $\bar{x}_0$. Then, it generates $\hat{e}$ as follows: each $\|l\| \in e$ is replaced by $\|\hat{l}\|$ where $\hat{l}$ is a linear expression (over $\bar{x}_0$) that satisfies $\hat{l} \geq l$. The expression $\hat{l}$ is computed using parametric integer linear programming [Feautrier 1988], i.e., maximizing l w.r.t. $\Psi \wedge \varphi$ and the parameters $\bar{x}_0$. Alternatively, $\hat{l}$ can be computed by syntactically looking for an expression $\xi \leq \hat{l}$ in $\exists \bar{x}_0 \cup \{\xi\}. \Psi \wedge \varphi \wedge \xi = l$ where ξ is a new variable. The advantage of the later approach is that it can be used with any library for manipulating linear constraints.

The use of the above automated techniques is what makes the corresponding approach widely applicable. In the rest of this paper, we let $\hat{f}_C$ denote a ranking function for a given CR C, and $\hat{e}$ denote the maximization of a cost expression e in some context. We assume that they are computed as above.

$$\begin{array}{l}\langle C(\bar{x}) = 0, \varphi_0 \rangle \\ \langle C(\bar{x}) = e + C(\bar{x}_1) + \cdots + C(\bar{x}_m), \varphi_1 \rangle\end{array}$$

Fig. 3: CR with single recursive equation.

Our challenge is to improve the precision of the above approach for solving CRs while still keeping a similar applicability for UBs and, besides, be able to apply it for inferring useful LBs. The fundamental idea is to generate a sequence of non-negative elements $\langle u_1, \ldots, u_{\hat{\kappa}} \rangle$, with $\hat{\kappa} \geq \kappa$, such that for any concrete evaluation $\langle e_1, \ldots, e_\kappa \rangle$, each e_i has a corresponding *different* u_j satisfying $u_j \geq e_i$ (observe that the subindexes might not match as $\hat{\kappa} \geq \kappa$). This guarantees soundness since then $u_1 + \cdots + u_{\hat{\kappa}}$ is an UB of $e_1 + \cdots + e_\kappa$. Moreover, it is potentially more precise since the u_i's are not required to be all equal. For the above example, we generate the sequence $\langle \|x_0\|, \|x_0 - 1\|, \ldots, 0 \rangle$. This allows inferring the UB $\frac{\|x_0\| * \|x_0+1\|}{2}$ which is more precise than $\|x_0\| * \|x_0 + 1\|$ that we have obtained before.

Technically, we compute the approximation by transforming the CR into a (worst-case) RR (as in Definition 2.3) whose exact closed-form solution is $u_1 + \cdots + u_{\hat{\kappa}}$. When e is a simple linear expression such as $e \equiv \|l\|$, the novel idea is to view $u_1, \ldots, u_{\hat{\kappa}}$ as an arithmetic sequence that starts from $u_{\hat{\kappa}} \equiv \hat{e}$ 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., $u_i = u_{i-1} + \check{d}$. When e is a complex nonlinear expression, e.g., $\|l\| * \|l'\|$, it cannot be precisely approximated using sequences. For such cases, our novel contribution is a method for approximating e by approximating its $\|.\|$ sub-expressions (which are linear) separately.

An important advantage of our approach w.r.t. previous ones [Albert et al. 2011b; Gulwani et al. 2009; Hoffmann et al. 2011], is that the problem of inferring LBs is dual. In particular, we can infer a LB $\check{\kappa}$ on the length of chains of recursive calls, the minimum value $\check{e}$ to which e can be evaluated, and then sum the sequence $\langle \ell_1, \ldots, \ell_{\check{\kappa}} \rangle$ where $\ell_i = \ell_{i-1} + \check{d}$ and $\ell_1 = \check{e}$. For the above example, we have $\check{e} = 0$, $\check{d} = 1$ and $\check{\kappa} = \|\frac{x_0+1}{3}\|$ and thus the LB we infer is: $C^{lb}(x_0) = \frac{1}{2} * \|\frac{x_0+1}{3}\| * (\|\frac{x_0+1}{3}\| + 1)$.

In addition, our techniques can be applied to cost expressions with any progression behavior that can be modeled using sequences, and not only a linear progression behavior. Indeed, in this paper we develop our techniques also for geometric progression.

4. INFERENCE OF PRECISE UPPER BOUNDS

In this section, we present our approach to accurately infer UBs on the resource consumption in the following steps:

(1) In Section 4.1, we handle a subclass of CRs which are defined by a single recursive equation and accumulate a constant cost.
(2) In Section 4.2, we handle CRs which are still defined by a single recursive equation but accumulate non-constant costs.
(3) In Section 4.3, we treat CRs with multiple overlapping equations.
(4) In sections 4.1, 4.2 and 4.3 we assume that base-case equations always contribute cost zero, and in Section 4.4 we explain how to handle non-zero base-case equations.
(5) Finally, in Section 4.5, we finish with some concluding remarks.

4.1. Cost Relations with Constant Cost

We consider CRs defined by a single recursive equation as depicted in Figure 3, where e contributes a *constant cost*, i.e., it is a constant number. As explained in Section 3, any chain of calls in C when starting from $C(\bar{x}_0)$ is at most of length $\hat{f}_C(\bar{x}_0)$. We aim at obtaining an UB for C by solving a corresponding RR P_C in which all chains of calls are of length $\hat{f}_C(\bar{x}_0)$. Intuitively, P_C can be seen as a special case of a RR such that its recursive equation has m recursive calls (as in C), where all chains of calls are of length N, and each application accumulates the constant cost e. Its solution can be then instantiated for the case of C by replacing N by $\hat{f}_C(\bar{x}_0)$.

Definition 4.1. The worst-case RR of the CR C of Figure 3, when e is constant cost, is $\langle P_C(N){=}e + m * P_C(N-1)\rangle$.

The main achievement of the above transformation is that, for CRs with constant cost expressions, we get rid of their problematic features (1) and (2) described in Section 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 4.2. *Let E be a solution for $P_C(N)$ of Definition 4.1. Then, $C^{ub}(\bar{x}_0) = E[N/\hat{f}_C(\bar{x}_0)]$ is an UB for its corresponding CR C.*

Example 4.3. The worst-case RR of the CR C of Figure 1b is $\langle P_C(N){=}1{+}P_C(N{-}1)\rangle$, which is solved using CAS to $P_C(N){=}N$ for any $N \geq 0$. The UB for C is obtained by replacing N by the corresponding ranking function $\hat{f}_C(k_0, j_0, q_0) = \|j_0 + q_0 - k_0\|$ which results in $C^{ub}(k_0, j_0, q_0) = \|j_0 + q_0 - k_0\|$.

4.2. Cost Relations with Non-Constant Cost

During cost analysis, in many cases we obtain CRs like the one of Figure 3, 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 Definition 4.1 would not be correct since in these cases e must be appropriately related to N. In particular, the main difficulty is to simulate the accumulation of the non-constant expressions e_i at the level of the RR. In this section we formalize the ideas intuitively explained in Section 3 which are based on using sequences to simulate the behavior of e.

We distinguish two cases: CRs with linear (a.k.a., arithmetic) and CRs with geometric progression behavior. In general, the cost expression e has a complex form (e.g., exponential, polynomial, etc.). Therefore, even a simple cost expression like $\|x + y\| * \|x + y\|$ does not increase arithmetically or geometrically even if the sub-expression $x + y$ does. Therefore, limiting our approach to cases in which e has a linear or geometric progression behavior would narrow its applicability. Instead, a key observation in our approach is that, it is enough to reason on the behavior of its $\|.\|$ sub-expressions, i.e., we only need to understand how each $\|l\| \in e$ changes along a sequence of calls to C, which very often have a linear or geometric progression behavior since l is a linear expression.

4.2.1. Linear Progression Behavior. This section describes how to obtain an UB for the CR of Figure 3, when e includes $\|.\|$ sub-expressions with linear progression behavior, using a RR that simulates the behavior of each $\|.\|$ sub-expression separately. We first characterize the notion of linear progression behavior of a $\|.\|$ expression.

Definition 4.4 ($\|.\|$ *with linear progression behavior*). Consider the CR C of Figure 3. We say that $\|l\| \in e$ has an increasing (resp. decreasing) linear progression behavior, if there exists a *progression parameter* $\check{d} > 0$, such that for any two consecutive contributions of e during the evaluation of $C(\bar{x}_0)$, denoted e' and e'', it holds that $l'' - l' \geq \check{d}$ (resp. $l' - l'' \geq \check{d}$) where $\|l'\| \in e'$ and $\|l''\| \in e''$ are the instances of $\|l\|$.

For the case of the CR of Figure 3, the two consecutive instances e' and e'' in the above definition refer to two consecutive nodes in the corresponding evaluation tree (a node e', and one of its children e''). Note that there might be several values for $\check{d}$ that satisfy the conditions of the above definition. For example, if a $\|.\|$ expression decreases at least by 2, then it also decreases at least by 1, and therefore both $\check{d} = 1$ and $\check{d} = 2$ satisfy the conditions of the above definition. Although taking $\check{d} = 1$ is sound, it leads to inferring less precise bounds. Therefore, our interest is in finding the *maximum* $\check{d}$ that satisfies the above definition. It is important to note that this maximum value for (the minimum decrease/increase) $\check{d}$ is different from the maximum decrease/increase. In practice, we compute such $\check{d}$ for a given $\|l\| \in e$ with an increasing (resp. decreasing) behavior as follows: let $\langle C(\bar{y}) = e' + C(\bar{y}_1) + \cdots + C(\bar{y}_m),\ \varphi'_1\rangle$ be a renamed apart instance of the recursive equation of C such that l' is the renaming of l, and for each $1 \leq i \leq m$ let $\check{d}_i$ be the result of minimizing the objective function $l' - l$ (resp. $l - l'$) with respect to $\varphi_1 \wedge \varphi'_1 \wedge \bar{x}_i = \bar{y}$ using integer programming, then $\check{d} = \min(\check{d}_1, \ldots, \check{d}_m)$.

Example 4.5. Consider again the cost relation B of Figure 1b. Replacing the call $C(0, j, q)$ by the UB $\|q + j\|$ computed in Example 4.3 results in $\langle B(j, i, q) = \|q + j\| + B(j', i, q), \varphi_1\rangle$ where $\varphi_1 = \{j < i, j + 1 \leq j' \leq j + 3\}$. A renamed apart instance of this equation is $\langle B(j_r, i_r, q_r) = \|q_r + j_r\| + B(j'_r, i_r, q_r), \varphi'_1\rangle$ where $\varphi'_1 = \{j_r < i_r, j_r + 1 \leq j'_r \leq j_r + 3\}$. Minimizing the objective function $(q_r + j_r) - (q + j)$ with respect to $\varphi_1 \wedge \varphi_1 \wedge \{j' = j_r, i = i_r, q = q_r\}$ results in $\check{d}_1 = 1$. Therefore, $\|q + j\|$ has an increasing linear progression behavior with a progression parameter $\check{d} = 1$.

As explained in Section 3, the goal is to use a linear sequence that starts from the maximum value that a given $\|l\| \in e$ can take, i.e., $\|\hat{l}\|$, and in each step decreases by the minimum distance $\check{d}$ between two consecutive instances of $\|l\|$. Let us intuitively explain how our method works by focusing on a single $\|l\| \in e$ within the relation C, assuming that it has a decreasing linear progress behavior with a progression parameter $\check{d}$. Recall that during the evaluation of an initial query $C(\bar{x}_0)$, any chain of calls has a length $\kappa \leq \hat{f}_C(\bar{x}_0)$. Let $\|l_1\|, \ldots, \|l_\kappa\|$ be the instances of $\|l\|$ contributed in each call. Our aim is to generate a sequence of elements $u_1, \ldots, u_\kappa$ such that $u_i \geq \|l_i\|$. Then, each u_i will be used instead of $\|l_i\|$ in order to over-approximate the total cost contributed by the i-th call.

Since $l_i - l_{i+1} \geq \check{d}$, for the first κ elements of the sequence $\{u_1 = \|\hat{l}\|, u_i = u_{i-1} - \check{d}\}$ it holds that $u_1 \geq l_1, \ldots, u_\kappa \geq l_\kappa$. However, this does not imply yet $u_i \geq \|l_i\|$ since when $l_i < 0$ we have $\|l_i\| = 0$ but the corresponding u_i might be negative. This mainly happens when κ is an over-approximation of the actual length of the chain of calls. Therefore, an imprecise (too large) $\hat{f}_C$ would lead to a large decrease and the smallest element $\|\hat{l}\| - \check{d} * (\hat{f}_C(\bar{x}_0) - 1)$, and possibly other subsequent ones, could be negative and would provide an incorrect result.

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, assuming that $\hat{f}_C(\bar{x}_0) = \|l'\|$, we start from $\|\hat{l} - \check{d} * l'\| + \check{d}$ which is guaranteed to be positive and greater than or equal to the smallest value $\|\hat{l}\| - \check{d} * (\hat{f}_C(\bar{x}_0) - 1)$. This mean that when the smallest value is negative, we shift the sequence and start from a positive smallest value $\check{d}$ until the biggest value $\|l'\| * \check{d} \geq \|\hat{l}\|$. Therefore, using $u_i = \|\hat{l} - \check{d} * l'\| + (\|l'\| - i + 1) * \check{d}$, it is guaranteed that $u_1 \geq \|l_1\|, \ldots, u_\kappa \geq \|l_\kappa\|$. Similar reasoning can be done when for the case in which $\|l\| \in e$ is linearly increasing by $\check{d}$. The next definition, that generalizes Definition 4.1, uses this intuition to replace each $\|.\|$ by an expression that generates its corresponding sequence at the level of RR.

Definition 4.6. Consider the CR C of Figure 3, and let $\hat{f}_C(\bar{x}_0) = \|l'\|$. Its associated worst-case RR is $\langle P_C(N) = \hat{E}_e + m * P_C(N-1)\rangle$ where $\hat{E}_e$ is obtained from e by replacing each $\|l\| \in e$ by l_{RR} such that $l_{RR} \equiv \|\hat{l} - \check{d} * l'\| + (\|l'\| - N + 1) * \check{d}$ (resp. $l_{RR} \equiv \|\hat{l} - \check{d} * l'\| + N * \check{d}$) if $\|l\|$ is linearly increasing (resp. decreasing) with a progression parameter $\check{d}$; otherwise $l_{RR} \equiv \|\hat{l}\|$.

Example 4.7. Let us see how a given $\|l\| \in e$, which is linearly decreasing by $\check{d}$, is simulated in P_C. If we apply P_C on $N = \hat{f}_C(\bar{x}_0) = \|l'\|$, i.e., on the maximum depth of the evaluation tree, then the part that corresponds to $\|l\|$ in $\hat{E}_e$ is evaluated to $\|\hat{l} - \check{d} * l'\| + \|l'\| * \check{d}$. Then, in the following iteration, when applying P_C on $N-1$, the same part is evaluated to $\|\hat{l} - \check{d} * l'\| + (\|l'\| - 1) * \check{d}$ which is smaller than the first one, and so forth. If $\|l\|$ is linearly increasing by $\check{d}$, then in the first application of P_C we get $\|\hat{l} - \check{d} * l'\| + \check{d}$, in the second one we get $\|\hat{l} - \check{d} * l'\| + 2 * \check{d}$, and so forth.

Note that, in Definition 4.6, if $\|l\| \in e$ does not have a linear progression behavior then it is replaced by $\|\hat{l}\|$, exactly as in the approach of Albert et al. [2011b]. Note also that the distinction between the decreasing and increasing cases is of great importance when the CR has more than one recursive call. This affects the number of times the largest element of the sequence is contributed: one time (in the root of the evaluation tree) in the decreasing case, and $2^{(N-1)}$ times (the last level of internal nodes in the evaluation tree) in the increasing case. 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 4.8. *Let E be a solution for $P_C(N)$ of Definition 4.6. Then $C^{ub}(\bar{x}_0) = E[N/\hat{f}_C(\bar{x}_0)]$ is an UB for its corresponding CR C.*

Example 4.9. Consider the standalone CR B of Example 4.5, and recall that $\|q + j\|$ increases linearly with a progression parameter $\check{d} = 1$. Function $\hat{f}_B(j_0, i_0, q_0) = \|i_0 - j_0\|$ is a ranking function for CR B. Maximizing $\|q + j\|$ results in $\|q_0 + i_0 - 1\|$. Then, using Definition 4.6 we generate the *worst-case* RR $P_B(N)$ depicted in Figure 2 whose solution (computed by CAS) is:

$$P_B(N) = \|q_0 + j_0 - 1\| * N + \|i_0 - j_0\| * N + \frac{N}{2} - \frac{N^2}{2}$$

By Theorem 4.8, replacing N by $\hat{f}_B(j_0, i_0, q_0)$ results in:

$$B^{ub}(j_0, i_0, q_0) = \|q_0 + j_0 - 1\| * \|i_0 - j_0\| + \frac{\|i_0 - j_0\|}{2} * (\|i_0 - j_0\| + 1)$$

Substituting this UB in the cost relation A of Figure 1b results in the CR:

$$\langle A(i, q) = \|q - 1\| * \|i\| + \frac{\|i\|}{2} * (\|i\| + 1) + A(i', q), \{i + 1 \le q, i + 2 \le i' \le i + 4\}\rangle$$

Note that in this CR the expression $\|q - 1\|$ always evaluates to the same value, while $\|i\|$ has an increasing linear progression behavior with progression parameter $\check{d} = 2$. Given that: (1) $\hat{f}_A(i_0, q_0) = \|\frac{q_0 - i_0}{2}\|$; (2) the maximization of $\|q - 1\|$ is $\|q_0 - 1\|$; and (3) the maximization of $\|i\|$ is $\|q_0 - 1\|$, by applying Definition 4.6, we generate the *worst-case* RR $P_A(N)$ depicted in Figure 2, which is solved by CAS to:

$$\begin{aligned} P_A(N) \ = \ & \tfrac{N}{6} * [4 * N^2 + 3 * \|i_0 - 1\| * (2 * N + \|i_0 - 1\| + 3) + \\ & 6 * \|q_0 - 1\| * (\|i_0 - 1\| + N + 1) + 9 * N + 5] \end{aligned}$$

By Theorem 4.8, replacing N by $\hat{f}_A(i_0, q_0)$ results in:

$$\begin{array}{rcl} A^{ub}(i_0,q_0) & = & \frac{1}{6} * \|\frac{q_0-i_0}{2}\| * (4 * \|\frac{q_0-i_0}{2}\| * \|\frac{q_0-i_0}{2}\| + 3 * \|i_0-1\| * (2 * \|\frac{q_0-i_0}{2}\| \\ & & + \|i_0-1\| + 3) + 6 * \|q_0-1\| * (\|i_0-1\| + \|\frac{q_0-i_0}{2}\| + 1) + 9 * \|\frac{q_0-i_0}{2}\| + 5) \end{array}$$

Finally, substituting $A^{ub}(0, q_0)$ in the CR F, we obtain the UB:

$$F^{ub}(q_0) = \frac{1}{6} * \|\frac{q_0}{2}\| * (4 * \|\frac{q_0}{2}\| * \|\frac{q_0}{2}\| + 6 * \|q_0-1\| * (\|\frac{q_0}{2}\| + 1) + 9 * \|\frac{q_0}{2}\| + 5)$$

whereas the approach of Albert et al. [2011b] obtains $2 * \|\frac{q_0+1}{2}\| * \|q_0-1\|^2$, which is less precise.

4.2.2. Geometric Progression Behavior. The techniques of Section 4.2.1 can solve a wide range of CRs. However, in practice, we find also CRs that do not have constant or linear progression behavior, but rather a geometric progression behavior. This is typical in programs that implement divide and conquer algorithms, where the problem (i.e., the input) is divided into sub-problems which are solved recursively.

Example 4.10. Consider the following implementation of the merge-sort algorithm:

```
void msort(int a[], int low, int hi) {
  if ( hi > low ) {
    int mid=(hi+low)/2;
    msort(a,low,mid);
    msort(a,mid+1,hi);
    merge(a,low,mid,hi);
  }
}
```

where, for simplicity, we omit the code of merge and assume that its cost, for example, is $10 * \|hi - low + 1\|$, when counting the number of executed (bytecode) instructions. Using this UB, COSTA automatically generates the following CR for msort:

$$\begin{array}{l} \langle msort(a, low, hi) = 0, \varphi_1\rangle \\ \langle msort(a, low, hi) = 20 + 10 * \|hi - low + 1\| + msort(a, low, mid) + msort(a, mid', hi), \varphi_2\rangle \end{array}$$

where

$$\begin{array}{l} \varphi_1 = \{hi \geq 0, low \geq 0, hi \leq low\} \\ \varphi_2 = \{hi \geq 0, low \geq 0, hi \geq low + 1, mid' = mid + 1, low + hi - 1 \leq 2 * mid \leq low + hi\} \end{array}$$

The constant 20 corresponds to the cost of executing the comparison, the sum and division, and invoking the methods. The constraint $low + hi - 1 \leq 2 * mid \leq low + hi$ in φ_2 is used to model the behavior of the *integer division* mid=(low+hi)/2 with linear constraints. The progression behavior of $\|hi - low + 1\|$ is geometric, i.e., if $\|l_i\|$ and $\|l_{i+1}\|$ are two instances of $\|hi - low + 1\|$ in two consecutive calls, then $l_i \geq 2 * l_{i+1} - 1$ holds, which means that the value of $\|hi - low + 1\|$ is reduced *almost* by half at each iteration. It is not reduced exactly by half since $l_i \geq 2 * l_{i+1}$ does not hold when the input array is of odd size, in such case it is divided into two sub-problems with different (integer) sizes.

The above example demonstrates that: (1) there is a practical need for handling CRs with geometric progression behavior; and (2) the geometric progression in programs that manipulate integers does not comply the standard definition $u_i = c * r^i$ of geometric series, but rather it should consider small *shifts* around those values in order to account for examples like divide-and-conquer algorithms. The following definition specifies when a $\|.\|$ expression has a geometric progression behavior.

Definition 4.11 ($\|.\|$ *with geometric progression behavior*). Consider the *CR* C of Figure 3. We say that $\|l\| \in e$ has an increasing (resp. decreasing) geometric progression behavior, if there exist progression parameters $\check{r} > 1$ and $\check{p} \in \mathbb{Q}$, such that for any two consecutive contributions of e during the evaluation of $C(\bar{x}_0)$, denoted e' and e'', it holds that $l'' \geq \check{r} * l' + \check{p}$ (resp. $l' \geq \check{r} * l'' + \check{p}$) where $\|l'\| \in e'$ and $\|l''\| \in e''$ are the instances of $\|l\|$.

Note that the above increasing and decreasing conditions could be equivalently written as $\frac{l''}{\check{r}} + \check{p} \geq l'$ and $l'' \leq \frac{l'}{\check{r}} + \check{p}$ respectively. This might be more common in the literature, however, it does not lead to a simpler formalism. Thus, we prefer to use those of the above definition to keep the notation simpler.

As in the case of $\check{d}$ in the linear progression behavior, we are interested in values for $\check{r}$ and $\check{p}$ that are as close as possible to the *minimal* progression of $\|l\|$. This happens when $\check{r}$ is maximal, and for that maximal $\check{r}$, the value of $|\check{p}|$ is minimal. In practice, computing such $\check{r}$ and $\check{p}$ for a given $\|l\| \in e$ with an increasing (resp. decreasing) behavior is done as follows: let $\langle C(\bar{y}) = e' + C(\bar{y}_1) + \cdots + C(\bar{y}_m),\ \varphi'_1\rangle$ be a renamed apart instance of the recursive equation of C such that l' is the renaming of l, then we look for $\check{r}$ and $\check{p}$ such that for each $1 \leq i \leq m$ it holds that $\varphi_1 \wedge \varphi'_1 \wedge \bar{x}_i = \bar{y} \models l' \geq \check{r} * l + \check{p}$ (resp. $\varphi_1 \wedge \varphi'_1 \wedge \bar{x}_i = \bar{y} \models l \geq \check{r} * l' + \check{p}$). This can be done using Farkas' Lemma [Schrijver 1986], which provides a systematic way to derive all implied inequalities of a given set of linear constraints. However, systematically checking the conditions taking the coefficients and the constants that appear in φ_1 as candidates for $\check{r}$ and $\check{p}$, respectively, works well in practice.

Example 4.12. For the *CR* of Example 4.10, we have that $\|hi - low + 1\|$ is decreasing geometrically, with progression parameters $\check{r} = 2$ and $\check{p} = -1$. Note that 2 and -1 explicitly appear as coefficient and constant, respectively, in φ_1.

Similarly to the case of linear progression behavior in Section 4.2.1, the progression parameters $\langle \check{r}, \check{p} \rangle$ are used in order to over-approximate the contributions of a given $\|l\| \in e$ expression along a chain of calls. For example, if $\|l\| \in e$ has a decreasing geometric progression behavior, and $\|l_1\|, \ldots, \|l_\kappa\|$ are instances of $\|l\|$ along any chain of calls where $\kappa \leq \hat{f}_C(\bar{x}_0)$, then first κ elements of the sequence

$$u_i = \frac{\|\hat{l}\|}{\check{r}^{i-1}} + \|-\check{p}\| * \sum_{j=1}^{i-1} \frac{1}{\check{r}^j}$$

satisfy $u_i \geq \|l_i\|$. We use $\|-\check{p}\|$ in order to lift the negative value $-\check{p}$ (when $\check{p} > 0$) to zero and avoid that u_i goes into negative values. The following definition extends Definition 4.6, by handling the translation of $\|.\|$ expression with geometric behavior. First, to simplify the notation, let us denote the sum $\sum_{j=1}^{i} \frac{1}{\check{r}^j}$ by $\hat{S}(i)$, which is also equal to $\frac{1}{\check{r}^i} * \frac{1}{1-\check{r}} - \frac{1}{1-\check{r}}$.

Definition 4.13. We *extend* Definition 4.6 for the geometric progression case as follow: if $\|l\| \in e$ has an increasing (resp. decreasing) geometric progression behavior, then its corresponding l_{RR} is defined as

$$l_{RR} \equiv \frac{\|\hat{l}\|}{\check{r}^{(N-1)}} + \|-\check{p}\| * \hat{S}(N-1) \quad \left[\text{resp. } l_{RR} \equiv \frac{\|\hat{l}\|}{\check{r}^{(\hat{f}_C(\bar{x}_0)-N)}} + \|-\check{p}\| * \hat{S}(f_C(\bar{x}_0) - N)\right]$$

Note that value of $\frac{\|\hat{l}\|}{\check{r}^{(N-1)}} + \|-\check{p}\| * \hat{S}(N-1)$ decreases along the iterations of P_C, i.e., when N decreases. Similarly, the value of $\frac{\|\hat{l}\|}{\check{r}^{(\hat{f}_C(\bar{x}_0)-N)}} + \|-\check{p}\| * \hat{S}(f_C(\bar{x}_0) - N)$ increases.

Algorithm 1: compute_UB

```
   Input: Standalone $CR$ $C$, as in Figure 3
   Output: Closed-form UB $C^{ub}(\bar{x}_0)$ for $C$
1  Compute a loop bound $\hat{f}_C(\bar{x}_0)$;
2  Compute an invariant $\langle C(\bar{x}_0) \leadsto C(\bar{x}), \Psi \rangle$;
3  $\hat{E}_e = e$;
4  foreach $\|l\| \in e$ do
5      Let $\hat{l}$ be the result of maximizing $l$ w.r.t $\Psi \wedge \varphi_1$ and the parameter $\bar{x}_0$;
6      Compute the progression parameters $\check{d}$ or $\langle \check{r}, \check{p} \rangle$ for $l$;
7      Compute $l_{RR}$ as in definitions 4.6 and 4.13;
8      $\hat{E}_e = \hat{E}_e[\|l\|/l_{RR}]$;
9  end
10 Solve $\langle P_C(N) = \hat{E}_e + m * P_C(N-1) \rangle$ into a closed-form expression $E$ using $CAS$;
11 $C^{ub}(\bar{x}_0) = E[N/\hat{f}_C(\bar{x}_0)]$;
```

Note also that the distinction between the decreasing and the increasing cases is fundamental, and it is for the same reasons as in Definition 4.6. 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 4.14. *Let E be a solution for $P_C(N)$ of Definition 4.6, together with the extension of Definition 4.13. Then, $C^{ub}(\bar{x}_0) = E[N/\hat{f}_C(\bar{x}_0)]$ is an UB for its corresponding CR C.*

Example 4.15. Consider the CR of Example 4.10, and recall that $\|hi - low + 1\|$ decreases geometrically with progression parameters $\check{r} = 2$ and $\check{p} = -1$ (see Example 4.12). Moreover, the ranking function for the CR $msort$ is $\hat{f}_{msort}(a_0, low_0, hi_0) = \log_2(\|hi_0 - low_0\| + 1) + 1$, and maximization of $\|hi - low + 1\|$ results in $\|hi_0 + 1\|$. According to Definition 4.13, the associated worst-case RR (after simplifying $\hat{E}_e$ for clarity) is:

$$P_{msort}(N) = 30 + 10 * \frac{\|hi_0 + 1\| - 1}{2^{(\log_2(\|hi_0 - low_0\|+1)+1-N)}} + 2 * P_{msort}(N-1)$$

Obtaining a closed-form solution for $P_{msort}(N)$ using CAS, and then replacing N by $\hat{f}_{msort}(a_0, hi_0, low_0)$ results in the following UB for the CR $msort$:

$$msort^{ub}(a_0, low_0, hi_0) = 30 + 60 * \|hi_0 - low_0\| + 10 * (\log_2(\|hi_0 - low_0\| + 1) + 1) * (\|hi_0 + 1\| - 1).$$

Algorithm 1 summarizes the process of solving a given standalone CR as that of Figure 3, as described in sections 4.1 and 4.2. As we have explained in Section 2.1, non-standalone CRs are solved in a modular way: we first apply Algorithm 1 to the CRs that do not call any other CRs (i.e., standalone), then we continue by substituting the computed bounds in the equations that call such CRs, which in turn become standalone, and thus can be solved using Algorithm 1. This process is applied until all CRs are solved.

4.3. Non-constant Cost Relations with Multiple Equations

Any approach for solving CRs that aims at being practical has to consider CRs with several recursive equations as the one depicted in Figure 4. This kind of CRs is

$$\begin{array}{l}\langle C(\bar{x}) = 0, \varphi_0 \rangle \\ \langle C(\bar{x}) = e_1 + C(\bar{x}_1) + \cdots + C(\bar{x}_{m_1}), \varphi_1 \rangle \\ \qquad\qquad\vdots \\ \langle C(\bar{x}) = e_h + C(\bar{x}_1) + \cdots + C(\bar{x}_{m_h}), \varphi_h \rangle \end{array}$$

Fig. 4: *CRs* with multiple recursive equations

very common during cost analysis, and they mainly originate from conditional statements inside loops. For instance, the instruction "**if** (x[i]>0) A **else** B" may lead to non-deterministic equations which accumulate the costs of A and B. This is because arrays are typically abstracted to their length and, hence, the condition x[i]>0 is abstracted to $true$, i.e., we do not keep this information in the corresponding CR. Hence, $\varphi_1, \ldots, \varphi_h$ are not necessarily mutually exclusive. In what follows, w.l.o.g., we assume that $m_1 \geq \cdots \geq m_h$, i.e., the first (resp. last) recursive equation has the maximum (resp. minimum) number of recursive calls among all equations.

As a first solution to the problem of inferring an UB for the CR of Figure 4, we simulate its *worst-case* behavior, whenever possible, using another CR $\hat{C}$ with a single recursive equation. We refer to $\hat{C}$ as the *worst-case* CR of C. Namely, we generate the following CR

$$\langle \hat{C}(\bar{x}) = e + \hat{C}(\bar{x}_1) + \ldots + \hat{C}(\bar{x}_{m_1}), \varphi \rangle$$

such that the evaluation trees of $\hat{C}(\bar{x}_0)$ up to depth $\hat{f}_C(\bar{x}_0)$ over-approximate the evaluation trees of $C(\bar{x}_0)$. Then, we will infer an UB on the evaluation trees of $\hat{C}(\bar{x}_0)$ up to such depth, by generating a corresponding RR, which is then guaranteed to be an UB for $C(\bar{x}_0)$. The process of constructing $\hat{C}$ will be discussed later in this section, let us start by formalizing the conditions that $\hat{C}$ should satisfy, and how we approximate its evaluation trees up to a given depth.

Definition 4.16. We say that $\langle \hat{C}(\bar{x}) = e + \hat{C}(\bar{x}_1) + \cdots + \hat{C}(\bar{x}_{m_1}), \varphi \rangle$ is a worst-case CR for the CR C of Figure 4, if for any valuation $\bar{v}$ it holds that

$$max(\{sum(T, \hat{f}_C(\bar{v})) \mid T \in \mathcal{T}(\hat{C}(\bar{v}))\}) \geq max(answ(C(\bar{v})))$$

where $sum(T, \hat{f}_C(\bar{v}))$ denotes the sum of all nodes in T up to depth $\hat{f}_C(\bar{v})$.

Intuitively, we require that when evaluating $\hat{C}(\bar{v})$ until the maximum depth of the trees of $C(\bar{v})$, i.e., until depth $\hat{f}_C(\bar{v})$, we already get a larger cost than when evaluating $C(\bar{v})$. Note that we do not require the evaluation trees of $\hat{C}(\bar{v})$ to be finite, and indeed in some cases they are not, i.e., the loops of $\hat{C}$ are possibly non-terminating. This is because, when generating $\hat{C}$, we usually generalize $\varphi_1, \ldots, \varphi_h$ into φ which might affect the termination behavior. The following definition explains how to construct a worst-case RR for $\hat{C}$, that we use to approximate its cost up to depth $\hat{f}_C(\bar{v})$.

Definition 4.17. Given the CR C of Figure 4, a corresponding worst-case CR $\hat{C}$ as in Definition 4.16, and a ranking function $\hat{f}_C(\bar{x}_0)$ for C. The worst-case RR of $\hat{C}$ is defined as $\langle P_{\hat{C}}(N) = \hat{E}_e + m_1 * P_{\hat{C}}(N-1) \rangle$, where $\hat{E}_e$ is generated as in definitions 4.6 and 4.13, with the only difference of using $\hat{f}_C(\bar{x}_0)$ instead of $\hat{f}_{\hat{C}}(\bar{x}_0)$.

Let us clarify how we compute $\hat{E}_e$ from e in the above definition. In principle, it is computed as in definitions 4.6 and 4.13 but using $\hat{f}_C(\bar{x}_0) = \|l'\|$, and not $\hat{f}_{\hat{C}}(\bar{x}_0) = \|l'\|$, in order to account only for paths of at most length $\hat{f}_C(\bar{x}_0)$. Apart from this difference, it is important to note that when computing l_{RR} for $\|l\| \in e$: (1) the progression parameters are computed using $\hat{C}$ (i.e., using the constraints φ); and (2) $\|\hat{l}\|$ is computed by considering an invariant of $\hat{C}$, i.e., $\langle \hat{C}(\bar{x}_0) \leadsto \hat{C}(\bar{x}), \Psi \rangle$.

THEOREM 4.18. *Let E be a solution for $P_{\hat{C}}(N)$ of Definition 4.17. Then, $C^{ub}(\bar{x}_0) = E[N/\hat{f}_C(\bar{x}_0)]$ is an UB for the CR C.*

In what follows, we describe how to construct a worst-case *CR* $\hat{C}$. The set of constraints φ is simply constructed as the convex-hull of $\varphi_1, \ldots, \varphi_h$, taking into account that each φ_i might include local variables that do not occur in other φ_j. Next we describe how to compute e. Observe that any cost expression (that does not include $\max$) can be normalized to the form $\Sigma_{i=1}^{n} \Pi_{j=1}^{n_i} b_{ij}$ (i.e., sum of multiplications) where each b_{ij} is a *basic cost expression* of the form $\{r, \|l\|, m^{\|l\|}, \log(\|l\|+1)\}$. This normal form allows constructing e by considering the basic components of $e_1, \ldots, e_h$. For simplicity, we assume that $e_1, \ldots, e_h$ are given in this normal form, otherwise they could be normalized first. The following definition introduces the notion of a *generalization operator* for basic cost expressions. W.l.o.g., we consider that $e_1, \ldots, e_h$ have the same number of multiplicands n, and that all multiplicands have the same number of basic cost expressions m. This is not a restriction since otherwise, we just add 1 in multiplication and 0 in sum to achieve this form.

Definition 4.19 (*generalization of cost expressions*). A generalization operator $\sqcup$ is a mapping from pairs of basic cost expressions to cost expressions such that it satisfies $a \sqcup b \geq a$ and $a \sqcup b \geq b$. The $\sqcup$-generalization of two cost expressions $e_1 = \Sigma_{i=1}^{n} \Pi_{j=1}^{m} a_{ij}$ and $e_2 = \Sigma_{i=1}^{n} \Pi_{j=1}^{m} b_{ij}$ is defined as $e_1 \sqcup e_2 = \Sigma_{i=1}^{n} \Pi_{j=1}^{m} (a_{ij} \sqcup b_{ij})$.

The above definition does not provide an algorithm for generalizing two cost expressions, but rather a general method which is parametrized in: (1) the actual generalization operator $\sqcup$; and (2) the order of the multiplicands and the order of their basic cost expressions (since we generalize basic cost expressions with the same indexes). It is important to notice that there is no best-solution for these points, and that in practice heuristic-based solutions should be used. Below we describe such a solution.

As regards (1), any generalization operator should try first to prove that $a_{ij} \geq b_{ij}$ or $a_{ij} \leq b_{ij}$, and take the bigger one as the result. Such comparison is feasible due to the simple forms of the basic cost expressions, which are also known a priori. This means that one could generate a set of rules that specify conditions under which it is guaranteed that one cost expression is bigger than another one. E.g., $\|l_1\| \geq \|l_2\|$ if $l_1 \geq l_2$. Albert et al. [2010] defined such rules for comparing cost expressions in general. When the comparison fails, a possible sound solution is to take $a_{ij} + b_{ij}$. However, this might often results in too imprecise generalization. Again, the simple structure of such expressions makes it possible to build a set of generalization rules that obtain precise results. E.g., $\|2 * y_0 + z_0\|$ and $\|y_0 + 2 * z_0\|$ can be generalized into $\|2 * y_0 + 2 * z_0\|$, by taking the maximum of the coefficients that correspond to the same variables.

Algorithms 2 summarizes the approach that we have discussed so far for solving *CRs* with multiple equations. Let us now apply it to a concrete example.

Example 4.20. Let us add the following equation to the *CR* B of Example 4.5:

$$B(j,i,q) = \|j\|^2 + B(j',i,q) \; \{j+1 \leq i, j' = j+1\}$$

Now B has multiple equations, that impose a non-deterministic choice for accumulating either $e_1 = \|q+j\|$ or $e_2 = \|j\|^2$. Next, we compute $e = e_1 \sqcup e_2 = \|q+j\| * \|j\|$, and

Algorithm 2: compute_UB_MulEqn_1

```
   Input: $CR\ C$ with multiple equations, as in Figure 4
   Output: closed-form UB $C^{ub}(\bar{x}_0)$
1  Compute a loop bound $\hat{f}_C(\bar{x}_0)$;
2  Compute $e = e_1 \sqcup \cdots \sqcup e_h$;
3  Generalize $\varphi_1, \cdots, \varphi_h$ into $\varphi$;
4  Let $\langle \hat{C}(\bar{x}) = e + \hat{C}(\bar{x}_1) + \ldots + \hat{C}(\bar{x}_{m_1}), \varphi \rangle$ be a worst-case $CR$ for $C$;
5  Compute an invariant $\langle \hat{C}(\bar{x}_0) \leadsto \hat{C}(\bar{x}), \Psi \rangle$;
6  $\hat{E}_e = e$;
7  foreach $\|l\| \in e$ do
8      Let $\hat{l}$ be the result of maximizing $l$ w.r.t $\Psi \wedge \varphi$ and the parameter $\bar{x}_0$;
9      Compute the progression parameters $\check{d}$ or $\langle \check{r}, \check{p} \rangle$ for $l$ using $\hat{C}$;
10     Compute $l_{RR}$ as in definitions 4.6 and 4.6, using $\hat{f}_C(\bar{x}_0)$;
11     $\hat{E}_e = \hat{E}_e[\|l\|/l_{RR}]$;
12 end
13 Solve $\langle P_{\hat{C}}(N) = \hat{E}_e + m_1 * P_{\hat{C}}(N-1) \rangle$ into a closed-form expression $E$ using $CAS$;
14 $C^{ub}(\bar{x}_0) = E[N/\hat{f}_C(\bar{x}_0)]$;
```

generalize the corresponding constraints into $\{j+1 \le i, j+1 \le j' \le j+3\}$. Then we construct the worst-case CR $\hat{B}$ as

$$\hat{B}(j,i,q) = \|q+j\| * \|j\| + \hat{B}(j',i,q)\ \{j+1 \le i, j+1 \le j' \le j+3\}$$

Both $\|q+j\|$ and $\|j\|$ are increasing linearly with $\check{d} = 1$. Next we compute

$$\hat{E}_e = (\|q_0+j_0-1\| + (\|i_0-j_0\| - N + 1)) * ((\|j_0-1\| + (\|i_0-j_0\| - N + 1)))$$

and generate the corresponding worst-case RR

$$\langle P_{\hat{B}}(N) = (\|q_0+j_0-1\| + \|i_0-j_0\| - N + 1) * (\|j_0-1\| + \|i_0-j_0\| - N + 1) + P_{\hat{B}}(N-1) \rangle$$

which is solved by CAS to

$$\begin{array}{rcl} P_{\hat{B}}(N) & = & \frac{N}{6} * (2 * N^2 + 6 * \|i_0-j_0\|^2 - 6 * N * \|i_0-j_0\| + 3 * \|j_0-1\| + 6 * \|i_0-j_0\| \\ & & *(\|j_0-1\|+1) + 3 * \|q_0+j_0-1\| * (2 * \|j_0-1\| + 2 * \|i_0-j_0\| + 1) - 3 * \\ & & N * (\|j_0-1\| + \|q_0+j_0-1\| + 1) + 1) \end{array}$$

Instantiating N with $\hat{f}_B(j_0,i_0,q_0) = \|i_0-j_0\|$ gives (after simplification for clarity):

$$\begin{array}{rcl} B^{ub}(j_0,i_0,q_0) & = & \frac{1}{6} * \|i_0-j_0\| * [2 * \|i_0-j_0\| * \|i_0-j_0\| + 3 * \|i_0-j_0\| * \|j_0-1\| \\ & & +3 * \|i_0-j_0\| * \|q_0+j_0-1\| + 3 * \|i_0-j_0\| + 3 * \|q_0+j_0-1\| \\ & & +6 * \|q_0+j_0-1\| * \|j_0-1\| + 3 * \|j_0-1\| + 1] \end{array}$$

The above approach works well in practice, since in many cases the cost expressions contributed by the different equations have very similar structure, and they differ only in constant expressions. However, there are some cases where this approach fails to precisely generalize expressions $e_1, \ldots, e_h$, and thus might infer imprecise UBs.

Example 4.21. Consider the following CR

$$\begin{array}{ll} C(z,y) = 0 & \{z < 1, y < 1\} \\ C(z,y) = \|z\| + C(z',y) & \{z' = z-1, z > 0\} \\ C(z,y) = \|y\| + C(z,y') & \{y' = y-1, y > 0\} \end{array}$$

We Generalize $\|z\|$ and $\|y\|$ to $\|z+y\|$, and then infer $C^{ub}(z_0,y_0) = \|z_0+y_0\| * \|z_0+y_0\|$.

In what follows we present an alternative approach for solving (some cases of) CRs with multiple equations, which is able to handle the one of the above example. The main idea is to concentrate on the contribution of each equation, independently from the rest. We start by defining the projection of a CR C on its i-th equation, which is used later to compute an UB on the contributions of the i-th equation.

Definition 4.22. Given the CR C of Figure 4, we denote by C_i the CR obtained by replacing each e_j when $j \neq i$ by 0.

Clearly, if $C_i^{ub}(\bar{x}_0)$ is an UB for CR C_i, then $C^{ub}(\bar{x}_0) = \sum_{i=1}^{h} C_i^{ub}(\bar{x}_0)$ is an UB for CR C. The challenge is to compute a precise UB for each C_i. Of course one can use the generalization-based approach to solve each C_i, however this does not lead to precise UB. E.g, for the CR of Example 4.21 we obtain $\|y_0\| * \|z_0 + y_0\| + \|z_0\| * \|z_0 + y_0\|$. This is because 0 and $\|z_0\|$, for example, are generalized to $\|z\|_0$, and thus all corresponding 0 contributions (there are $\|y_0\|$ of them in this case) will be changed to $\|z_0\|$.

Let us consider a path in an arbitrary evaluation tree of $C_i(\bar{x}_0)$, and concentrate on the contributions of a single $\|l\| \in e_i$ in this path. As we have done so far, we aim at simulating these contributions using a corresponding arithmetic or geometric sequence, and then use this sequence to generate a corresponding RR whose solution can be transformed into an UB for C_i. There are two important issues that should be taken into account: (1) a ranking function for C (which is also valid for C_i) does not precisely bound the number of instances of $\|l\| \in e_i$, since it also accounts for visits to other equations; and (2) when computing the progression parameters of $\|l\| \in e_i$, it is not safe to consider only consecutive applications of the i-th equation, since between two applications of the i-th equation we might apply any other equations.

The above two issues can be solved as follows: (1) instead of using the ranking function $\hat{f}_C(\bar{x}_0)$, we use a function $\hat{f}_{C_i}(\bar{x}_0)$ which approximates the number of applications of the i-th equation only. Inferring such function can be done by instrumenting the CR with a counter that counts the number of visits to the i-th equation, and then infer an invariant that relates this counter to $\bar{x}_0$; and (2) when inferring the progression parameters $\check{d}$ or $\langle \hat{r}, \hat{p} \rangle$, we consider the increase/decrease in two subsequent applications of the i-th equation (rather than of two consecutive ones). Again, this can be inferred by means of an appropriate invariant.

Now let us see how to use $\hat{f}_{C_i}(\bar{x}_0)$ and the progression parameters (computed as in (2) above) in order to compute a precise UB for C_i, assuming that it has at most one recursive call, i.e., $m_1 = 1$ (later we discuss this restriction): (i) we generate a worst-case RR $P_{C_i}(N) = E_{e_i} + P_{C_i}(N-1)$ where E_{e_i} is computed as in definitions 4.6 and 4.13, but using $\hat{f}_{C_i}(\bar{x}_0)$ instead of $\hat{f}_C(\bar{x}_0)$, and by computing the progression parameters as in point (2) above; (ii) we solve $P_{C_i}(N)$ into a closed-form solution E using CAS; and (iii) $C_i^{ub}(\bar{x}_0) = E[N/\hat{f}_{C_i}(\bar{x}_0)]$ is guaranteed to be a correct UB for C_i. Algorithm 3 summarizes this approach, let us apply it to the CR of Example 4.21.

Example 4.23. Consider the CR of Example 4.21. We generate C_1 and C_2

$$\begin{array}{ll}
C_1(z,y) = 0 & \{z < 1, y < 1\} \\
C_1(z,y) = \|z\| + C_1(z',y) & \{z' = z-1, z > 0\} \\
C_1(z,y) = 0 + C_1(z,y') & \{y' = y-1, y > 0\} \\
\\
C_2(z,y) = 0 & \{z < 1, y < 1\} \\
C_2(z,y) = 0 + C_2(z',y) & \{z' = z-1, z > 0\} \\
C_2(z,y) = \|y\| + C_2(z,y') & \{y' = y-1, y > 0\}
\end{array}$$

Algorithm 3: compute_UB_MulEqn_2

```
   Input: $CR\ C$ of Figure 4 with $m_1 = 1$
   Output: Close-form upper bound $C^{ub}(\bar{x}_0)$
1  Compute an invariant $\langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \Psi\rangle$;
2  for $i = 1 \to h$ do
3      Generate $CR\ C_i$ as in Definition. 4.22;
4      Compute a bound $\hat{f}_{C_i}(\bar{x}_0)$ on the number of visits to the $i$-the equation;
5      $E_{e_i}$ = $e_i$;
6      foreach $\|l\| \in e_i$ do
7          Let $\hat{l}$ be the result of maximizing $l$ w.r.t $\Psi \wedge \varphi_i$ and the parameter $\bar{x}_0$;
8          Compute $\check{d}$ or $\langle \check{r}, \check{p}\rangle$ for $l$ (considering subsequent visits to the $i$-th equation);
9          Compute $l_{RR}$ as in definitions 4.6 and 4.6, using $\hat{f}_{C_i}(\bar{x}_0)$;
10         $\hat{E}_{e_i} = \hat{E}_{e_i}[\|l\|/l_{RR}]$;
11     end
12     Solve $\langle P_{\hat{C}_i}(N) = \hat{E}_{e_i} + P_{\hat{C}_i}(N-1)\rangle$ into a closed-form expression $E_i$ using $CAS$;
13     $C_i^{ub}(\bar{x}_0) = E_i[N/\hat{f}_{C_i}(\bar{x}_0)]$ ;
14 end
15 $C^{ub}(\bar{x}_0) = C_1^{ub}(\bar{x}_0) + \cdots + C_h^{ub}(\bar{x}_0)$ ;
```

Observe that (1) $\|z\|$ and $\|y\|$ are linearly decreasing with a progression parameter $\check{d} = 1$ when considering subsequent visit to the corresponding equations; (2) the maximization of $\|z\|$ and $\|y\|$ are $\|z_0\|$ and $\|y_0\|$ respectively; and (3) the number of applications of the first (resp. second) recursive equations of C_1 (resp. C_2) is $\hat{f}_{C_1}(z_0, y_0) = \|z_0\|$ (resp. $\hat{f}_{C_2}(z_0, y_0) = \|y_0\|$). We generate the worst-case RR for C_1 as follows:

$$\langle P_{C_1}(N) = \|z_0 - z_0 * 1\| + N + P_{C_1}(N-1)\rangle$$

Note that $\|z_0 - z_0 * 1\| = 0$, we just keep it for clarity. The solution of $P_{C_1}(N)$ obtained by CAS is $P_{C_1}(N) = \frac{1}{2} * N^2 + \frac{1}{2} * N$, and replacing N by $\hat{f}_{C_1}(z_0, y_0) = \|z_0\|$ we get the following UB for C_1

$$C_1{}^{ub}(z_0, y_0) = \frac{1}{2} * \|z_0\| * \|z_0\| + \frac{1}{2} * \|z_0\|$$

Similarly, We generate the worst-case RR for C_2 as follows:

$$\langle P_{C_2}(N) = \|y_0 - y_0 * 1\| + N + P_{C_2}(N-1)\rangle$$

The solution of $P_{C_2}(N)$ obtained from CAS is $P_{C_2}(N) = \frac{1}{2} * N^2 + \frac{1}{2} * N$ and, replacing N by $\hat{f}_{C_2}(z_0, y_0) = \|y_0\|$ we get the following UB for C_2

$$C_2{}^{ub}(z_0, y_0) = \frac{1}{2} * \|y_0\| * \|y_0\| + \frac{1}{2} * \|y_0\|$$

Then, computing $C^{ub}(z_0, y_0)$ as $C_1{}^{ub}(z_0, y_0) + C_2{}^{ub}(z_0, y_0)$ results in:

$$C^{ub}(z_0, y_0) = \frac{1}{2} * \|z_0\| * \|z_0\| + \frac{1}{2} * \|z_0\| + \frac{1}{2} * \|y_0\| * \|y_0\| + \frac{1}{2} * \|y_0\|$$

which is more precise than $\|z_0 + y_0\| * \|z_0 + y_0\|$.

It is important to note that this last approach is correct only when $m_1 = 1$, it might infer incorrect UBs if $m_1 > 1$. Let us intuitively see why. Suppose we change the RR

P_{C_i} such that it has $m_1 > 1$ recursive calls, then an evaluation tree for $P_{C_i}(\hat{f}_{C_i}(\bar{v}_0))$ might include less nodes than those contributed by the i-th equation in a corresponding evaluation tree for $C(\bar{v}_0)$. This is because deeper levels in an evaluation tree has more nodes, and since we have shortened the depth, by using $\hat{f}_{C_i}(\bar{x}_0)$ instead of $\hat{f}_C(\bar{x}_0)$, we might also reduce the number of such nodes. However, this approach is still practical since with $m_1 = 1$ we can handle all programs with (possibly nested) iterative constructs and/or a single recursive call (per method).

4.4. Non-zero Base-case Cost

So far, we have considered *CRs* with only one base-case equation, and moreover, we have assumed that its contributed cost is always 0. In practice, many *CRs* that originate from real programs have several non-zero base-case equations and, besides, the cost contributed by such equations is not necessarily constant. In this section, we describe how to handle such *CRs*.

Consider the *CR* C of Figure 4 and assume that, instead of one base-case equation, it has n base-case equations, where the i-th base-case equation is defined by $\langle C(\bar{x}) = e'_i, \varphi^i_0\rangle$. In order to account for these base-case equations, we first extend the worst-case *RR* P_C of definitions 4.1, 4.6 and 4.13 to include a generic base-case equation $\langle P_C(0) = \lambda\rangle$. Due to this extension, any solution E for P_C must involve the base-case symbol λ to account for all applications of the base-case equation.

In a second step, the base-case symbol λ in E is replaced by a cost expression e_λ that involves only $\bar{x}_0$ (i.e., it does not involve the parameter of P_C), and is greater than or equal to any instance of $\hat{e}'_i$ during the evaluation of $C(\bar{x}_0)$. The cost expression e_λ is simply defined as $e_\lambda = \hat{e}'_1 \sqcup \ldots, \sqcup \hat{e}'_n$, where $\hat{e}'_i$ is the maximization of e'_i as defined in Section 3, and $\sqcup$ is a generalization operator of cost expressions.

Example 4.24. Let us replace the base-case equation $\langle B(j,i,q) = 0, \{j \geq i\}\rangle$ of Figure 1b by the equations $\langle B(j,i,q) = \|j\|, \{j \geq i\}\rangle$ and $\langle B(j,i,q) = \|i\|, \{j \geq i\}\rangle$. Maximizations of such base-case costs are, respectively, $\hat{e}'_1 = \|i_0+2\|$, $\hat{e}'_2 = \|i_0\|$ and thus their generalization is $e_\lambda = \|i_0+2\|$. Solving P_B of Example 4.9, together with a base-case equation $P_B(0) = \lambda$, results in:

$$P_B(N) = \|q_0+j_0-1\| * N + \|i_0-j_0\| * N + \frac{N}{2} - \frac{N^2}{2} + \lambda$$

Then, replacing N by the ranking function $\|i_0-j_0\|$ and λ by e_λ we get

$$B^{ub}(j_0,i_0,q_0) = \|q_0+j_0-1\| * \|i_0-j_0\| + \frac{\|i_0-j_0\|}{2} * (\|i_0-j_0\|+1) + \|i_0+2\|\,.$$

4.5. Concluding Remarks

We have presented a practical and precise approach for inferring UBs on *CRs*. When considering *CRs* with a single recursive equation, in practice, our approach achieves an optimal precision. As regards *CRs* with multiple recursive equations, we have presented a solution which is effective in practice. Note that, although we have concentrated on arithmetic and geometric behavior of $\|.\|$ expression, our techniques can be adapted to any behavior that can be modeled with sequences.

It is important to point out that in some cases the output of *CAS*, when solving a *RR*, might not comply with the grammar of cost expressions as specified in Section 2. Concretely, after normalization, it might include sub-expressions of the form $-e$ where e is a multiplication of basic cost expression. Converting them to valid cost expressions can be simply done by removing such negative parts and obviously still have a sound UB. In practice, these negative parts are asymptotically negligible when compared to the other parts of the UB, and thus, removing them does not significantly affect the precision. In addition, the negative parts can be rewritten in order to *push* the minus sign inside a $\|.\|$ expression, e.g., $\|l_1\| - \|l_2\|$ is over-approximated by $\|l_1 - l_2\|$.

5. THE DUAL PROBLEM: INFERENCE OF LOWER BOUNDS

We now aim at applying the approach from Section 4 in order to infer *lower bounds*, i.e., under-approximations of the best-case cost. In addition to the traditional applications for performance debugging, program optimization and verification, such LBs are 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 Albert et al. [2011b] 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 precise. In what follows, in Section 5.1 we develop our method for inferring LBs for CRs with single recursive equation as the one of Figure 3, and, in Section 5.2 we handle CRs with multiple recursive equations as the one of Figure 4. Section 5.3 concludes.

5.1. Cost Relations with Single Recursive Equation

As explained in Section 3, the basic ideas for inferring LBs are dual to those described in Section 4 for inferring UBs, i.e., they are based on simulating the behavior of $\|.\|$ expressions with corresponding linear or geometric sequences. For example, if a given $\|l\| \in e$ is linearly increasing with a progression parameter $\check{d} \geq 0$, then it is simulated with an arithmetic sequence that starts from the minimum value to which $\|l\|$ can be evaluated, and increases in each step by $\check{d}$. In addition, the number of elements that we consider in such sequence is an under-approximation of the length of any chain of calls when evaluating $C(\bar{x}_0)$. In what follows, we develop our approach for inferring LBs on the CR of Figure 3 as follows: we first describe how to infer a lower-bound on the length of any chain of calls; then we describe how to infer the minimum value to which an expression $\|l\|$ can be evaluated; and finally we use this information in order to build a best-case RR that under-approximates the best-case cost of the CR C.

The following definition provides a practical algorithm for inferring an under-approximation on the length of any chain of calls when evaluating $C(\bar{x}_0)$ using the CR of Figure 4, which is also applicable for the CR of Figure 3.

Definition 5.1. Given the CR of Figure 4, a lower-bound on the length of any chain of calls during the evaluation of $C(\bar{x}_0)$ denoted as $\check{f}_C(\bar{x}_0)$ is computed as follows:

(1) *Instrumentation*: Replace each head $C(\bar{x})$ by $C(\bar{x}, lb)$, each recursive call $C(\bar{x}_j)$ by $C(\bar{x}_j, lb')$, and add $\{lb' = lb + 1\}$ to each φ_i;
(2) *Invariant*: Infer an invariant $\langle C(\bar{x}_0, 0) \leadsto C(\bar{x}, lb), \Psi\rangle$ for the new CR, such that the linear constraints Ψ hold between (the variables of) the initial call $C(\bar{x}_0, 0)$ and any recursive call $C(\bar{x}, lb)$; and
(3) *Synthesis*: compute l as the result of minimizing lb w.r.t $\Psi \wedge \varphi_0$ and the parameters $\bar{x}_0$, using parametric integer programming; or alternatively, compute l by syntactically looking for $lb \geq l$ in $\exists \bar{x}_0 \cup \{lb\}.\ \Psi \wedge \varphi_0$.

Then, $\check{f}_C(\bar{x}_0) = \|l\|$.

Let us explain intuitively the different steps of the above definition. In step 1, the CR C is instrumented with an extra argument lb which computes the length of the corresponding chain of calls, when starting the evaluation from $C(\bar{x}_0, 0)$. This instrumentation reduces the problem of finding a lower-bound on the length of any chain of calls to the problem of finding a (symbolic) minimum value for lb for which the base-case equation is applicable (i.e., the chain of calls terminates). This is exactly what steps 2 and 3 do. In 2, we infer an invariant Ψ on the arguments of any call $C(\bar{x}, lb)$ encountered during the evaluation of $C(\bar{x}_0, 0)$. This is done exactly as for the invariant described in Section 3 when maximizing cost expressions. In 3, from all states described by Ψ, we are interested only in those in which the base-case equation is applicable, i.e., in $\Psi \wedge \varphi_0$. Then, within this set of states, we take the minimum value l (in terms $\bar{x}_0$) of lb. Such l is the lower-bound we are interested in.

COROLLARY 5.2. *Function* $\check{f}_C(\bar{x}_0)$ *of Definition 5.1 is a* lower-bound on the length *of any chain of calls during the evaluation of* $C(\bar{x}_0)$.

Example 5.3. Applying step 1 of Definition 5.1 on the CR B of Example 4.5 we get

$$\begin{array}{lcll} \langle B(j,i,q,lb) & = & 0 & \{j \geq i\}\rangle \\ \langle B(j,i,q,lb) & = & \|q+j\| + B(j',i,q,lb') & \{j < i, j+1 \leq j' \leq j+3, lb' = lb+1\}\rangle \end{array}$$

The invariant for this CR is $\Psi = \{j - j_0 - lb \geq 0, j_0 + 3 * lb - j \geq 0, i = i_0, q = q_0\}$. Projecting $\Psi \wedge \{j \geq i\}$ on $\langle j_0, i_0, q_0, lb\rangle$ results in $\{j_0 + 3 * lb - i_0 \geq 0\}$ which implies $lb \geq \frac{(i_0 - j_0)}{3}$, from which we can synthesize $\check{f}_B(j_0, i_0, q_0) = \|\frac{i_0 - j_0}{3}\|$. Similarly, for CRs C and A of Figure 1b we obtain $\check{f}_C(k_0, j_0, q_0) = \|q_0 + j_0 - k_0\|$ and $\check{f}_A(i_0, q_0) = \|\frac{q_0 - i_0}{4}\|$.

Inferring the minimum value to which $\|l\| \in e$ can be evaluated is done in a dual way to that of inferring the maximum value to which it can be evaluated (see Section 3). Namely, using the invariant Ψ of Definition 5.1, we syntactically look for an expression $\xi \geq \check{l}$ in $\exists \bar{x}_0 \cup \{\xi\}. \Psi \wedge \varphi_1 \wedge \xi = l$ where ξ is a new variable. As in the case of maximization, the advantage of this approach is that it can be implemented using any tool for manipulation of linear constraints (e.g., PPL [Bagnara et al. 2008]). Alternatively, we can also use parametric integer programming [Feautrier 1988] in order to minimize l w.r.t. $\Psi \wedge \varphi_1$ and the parameters $\bar{x}_0$.

Now that we have all ingredients for under-approximating the behavior of a given $\|l\| \in e$. In the following definition, we generate the best-case RR P_C of CR C. Let us first explain the idea intuitively. Let $\|l_1\|, \ldots, \|l_\kappa\|$ be the first $\kappa \leq \check{f}_C(\bar{x}_0)$ elements contributed by a given $\|l\| \in e$ along a chain of calls, and assume that $l_i \geq 0$ for all $1 \leq i \leq \kappa$. If $\|l\|$ is linearly increasing (resp. decreasing) with a progression parameter $\check{d} > 0$, then the elements of the sequence $\{\ell_1 = \|\check{l}\|, \ell_i = \ell_{i-1} + \check{d}\}$ satisfy $\ell_i \leq \|l_i\|$ (resp. $\ell_i \leq \|l_{\kappa-i+1}\|$). Similarly, if $\|l\|$ is geometrically increasing (resp. decreasing) with progression parameters $\check{r}$ and $\check{p}$, then the elements of the sequence $\{\ell_1 = \|\check{l}\|, \ell_i = \check{r} * \ell_{i-1} + \check{p}\}$ satisfy $\ell_i \leq \|l_i\|$ (resp. $\ell_i \leq \|l_{\kappa-i+1}\|$). The following definition uses these sequences in order to under-approximate the behavior of $\|l\|$. Note that the condition $l_i \geq 0$ is essential, otherwise, the sequence ℓ_i is not a sound under-approximation.

Definition 5.4. Let C be the CR of Figure 3, and $\check{f}_C(\bar{x}_0)$ a lower-bound function on the length of any chain of calls generated during the evaluation of $C(\bar{x}_0)$. Then, the best-case RR of C is $P_C(N) = \check{E}_e + m * P_C(N-1)$ where $\check{E}_e$ is obtained from e by replacing each $\|l\| \in e$ by l_{RR} where:

(1) $l_{RR} \equiv \|\check{l}\| + (\check{f}_C(\bar{x}_0) - N) * \check{d}$, if it is linearly increasing and $\check{l} \geq 0$;
(2) $l_{RR} \equiv \|\check{l}\| + (N-1) * \check{d}$, if it is linearly decreasing and $\check{l} \geq 0$;

(3) $l_{RR} \equiv \check{r}^{(\check{f}_C(\bar{x}_0)-N)} * \|\check{l}\| + \check{p} * \check{S}(\check{f}_C(\bar{x}_0) - N)$, if it is geometrically increasing and $\check{l} \geq 0$;
(4) $l_{RR} \equiv \check{r}^{(N-1)} * \|\check{l}\| + \check{p} * \check{S}(N-1)$, if it is geometrically decreasing and $\check{l} \geq 0$;
(5) $l_{RR} \equiv \|\check{l}\|$, otherwise.

where $\check{S}(i) = \frac{\check{r}^i - 1}{\check{r}-1}$

THEOREM 5.5. *Let E is a solution for $P_C(N)$ of Definition 5.4. Then, $C^{lb}(\bar{x}_0) = E[N/\check{f}_C(\bar{x}_0)]$ is a LB for $C(\bar{x}_0)$.*

An algorithm that summarizes the above approach can be derived in a very similar way to Algorithm 1, simply by considering the dual notions to $\hat{l}$ and $\hat{f}_C(\bar{x}_0)$.

Example 5.6. Consider again the LBs on the length of chains of calls as described in Example 5.3. Since $C(k_0, j_0, q_0)$ accumulates a constant cost 1, its LB cost is $\|q_0 + j_0 - k_0\|$. We now replace the call $C(0, j, q)$ in B by its LB $\|q + j\|$ and obtain the following recursive equation:

$$\langle B(j,i,q) = \|q+j\| + B(j',i,q), \{j+1 \leq i, j+1 \leq j' \leq j+3\}\rangle$$

Notice the need of the soundness requirement in Definition 5.4, i.e., $q_0 + j_0 \geq 0$ where $q_0 + j_0$ is the minimization of $q + j$ for any call to $B(j_0, i_0, q_0)$. For example, when evaluating $B(-5, 5, 0)$ the first 5 instances of $\|q + j\|$ are zero since they correspond to $\|-5\|, \ldots, \|-1\|$. Therefore, it would be incorrect to start accumulating from 0 with a difference 1 at each iteration. However, in the context of the *CRs* of Figure 1b, it is guaranteed that $q_0 + j_0 \geq 0$ (since it is always called with $j \geq 0$ and $q \geq 0$). Using Definition 5.4, we generate the *best-case RR P_B* depicted in Figure 2 which is solved by *CAS* to

$$P_B(N) = \|q_0 + j_0\| * N + \|\frac{i_0 - j_0}{3}\| * N - \frac{N^2}{2} - \frac{N}{2}$$

Then, according to Theorem 5.5

$$B^{lb}(j_0, i_0, q_0) = \frac{1}{2} * \|\frac{i_0 - j_0}{3}\| * (\|\frac{i_0 - j_0}{3}\| + 2 * \|q_0 + j_0 - \frac{1}{2}\|)$$

Substituting this LB in the *CR A* of Figure 1b results in the *CR*

$$\langle A(i,q) = \frac{1}{2} * \|\frac{i}{3}\| * (\|\frac{i}{3}\| + 2 * \|q - \frac{1}{2}\|) + A(i', q), \{i+1 \leq q, i+2 \leq i' \leq i+4\}\rangle$$

In this *CR*, the expression $2 * \|q - \frac{1}{2}\|$ is constant, while $\|\frac{i}{3}\|$ has an increasing linear progression behavior with $\check{d} = \frac{2}{3}$. According to Definition 5.4, the generated *best-case RR P_A* is depicted in Figure 2 which is solved using *CAS* to

$$\begin{aligned} P_A(N) \ = \ & \tfrac{N}{54} * (4 * N^2 + 6 * N + 18 * \|\tfrac{i_0}{3}\| * (N-1) + 18 * \|q_0 - \tfrac{1}{2}\| * (N-1) + \\ & 27 * \|\tfrac{i_0}{3}\| * \|\tfrac{i_0}{3}\| + 54 * \|\tfrac{i_0}{3}\| * \|q_0 - \tfrac{1}{2}\| - 12 * \|\tfrac{q_0 - i_0}{4}\| + 2) \end{aligned}$$

Then, according to Theorem 5.5, i.e., substituting N by $\|\frac{q_0 - i_0}{4}\|$, we obtain

$$\begin{aligned} A^{lb}(i_0, q_0) \ = \ & \tfrac{1}{54} * \|\tfrac{q_0 - i_0}{4}\| * (4 * \|\tfrac{q_0 - i_0}{4}\| * \|\tfrac{q_0 - i_0}{4}\| + 6 * \|\tfrac{q_0 - i_0}{4}\| + 18 * \|\tfrac{i_0}{3}\| \\ & * (\|\tfrac{q_0 - i_0}{4}\| - 1) + 18 * \|q_0 - \tfrac{1}{2}\| * (\|\tfrac{q_0 - i_0}{4}\| - 1) + 27 * \|\tfrac{i_0}{3}\| * \|\tfrac{i_0}{3}\| \\ & + 54 * \|\tfrac{i_0}{3}\| * \|q_0 - \tfrac{1}{2}\| - 12 * \|\tfrac{q_0 - i_0}{4}\| + 2) \end{aligned}$$

Finally, the LB of $F(q_0)$ is

$$\begin{aligned} F^{lb}(q_0) \ = \ & \tfrac{1}{54} * \|\tfrac{q_0}{4}\| * (4 * \|\tfrac{q_0}{4}\| * \|\tfrac{q_0}{4}\| + 6 * \|\tfrac{q_0}{4}\| + 18 * \|q_0 - \tfrac{1}{2}\| * \\ & (\|\tfrac{q_0}{4}\| - 1) - 12 * \|\tfrac{q_0}{4}\| + 2)\,. \end{aligned}$$

5.2. Cost Relations with Multiple Recursive Equations

We infer LBs for CRs with multiple recursive equations in a dual way to the inference of UBs, namely: we first try to generate a best-case CR $\check{C}$, for the multiple recursive CRs C in Figure 4, in a similar way to the worst-case CR $\hat{C}$. If this is not possible (or not precise enough) and $m_1 = 1$ (i.e. we have at most one recursive call) then we can use the second approach, in which we compute a LB for each C_i (the projection of C on the i-th equation), and then sum all these LBs into a sound LB for C.

Definition 5.7. We say that $\langle \check{C}(\bar{x}) = e + \check{C}(\bar{x}_1) + \cdots + \check{C}(\bar{x}_{m_1}), \varphi \rangle$ is a best-case CR for the CR C of Figure 4, if for any valuation $\bar{v}$ it holds that

$$min(\{sum(T, \check{f}_C(\bar{v})) \mid T \in \mathcal{T}(\check{C}(\bar{v}))\} \leq min(answ(C(\bar{v})))$$

where $sum(T, \check{f}_C(\bar{v}))$ denotes the sum of all nodes in T up to depth $\check{f}_C(\bar{v})$.

CR $\check{C}$ is generated in a similar way to $\hat{C}$. The only difference is that in order to generate the cost expression e, we use a *reduction operator* $\sqcap$ instead of a generalization operator. Such operator guarantees that $a \sqcap b \leq a$ and $a \sqcap b \leq b$. In practice, the reduction operator $\sqcap$ is implemented by syntactically analyzing the input cost expressions, in a similar way to the case of $\sqcup$.

THEOREM 5.8. *Given the CR C of Figure 4, a corresponding $\check{f}_C(\bar{x}_0)$ as defined in Definition 5.1, a best-case CR $\check{C}$ for C, and a solution E for the RR $\langle P_{\check{C}}(N) = \check{E}_e + m_h * P_{\check{C}}(N-1) \rangle$. Then $C^{lb}(\bar{x}_0) = E[N/\check{f}_C(\bar{x}_0)]$ is a LB for the CR C.*

Note that in the above theorem the expression $\check{E}_e$ is generated as in Definition 5.4, but using $\check{f}_C(\bar{x}_0)$ instead of $\check{f}_{\check{C}}(\bar{x}_0)$ (since applying Definition 5.4 on $\check{C}$ would use $\check{f}_{\check{C}}(\bar{x}_0)$).

Example 5.9. Consider the CR B in Example 4.20. We simulate its best-case behavior by the following single recursive equation

$$\langle \check{B}(j, i, q) = \|j\| + \check{B}(j', i, q), \{j + 1 \leq i, j + 1 \leq j' \leq j + 3\} \rangle$$

Note that (1) $\|j\|$ under-approximates both e_1 and e_2; and (2) $\|j\|$ has an increasing linear progression behavior with progression parameter $\check{d} = 1$. Using Definition 5.4, we generate the following best-case RR $P_{\check{B}}$ for $\check{B}$

$$\langle P_{\check{B}}(N) = \|j_0\| + (\|\frac{i_0 - j_0}{3}\| - N) * 1 + P_{\check{B}}(N-1) \rangle$$

which is solved by CAS to

$$P_{\check{B}}(N) = N * \|\frac{i_0 - j_0}{3}\| + N * \|j_0\| - \frac{1}{2} * N^2 - \frac{1}{2} * N$$

According to Theorem 5.8, replacing N by $\check{f}_B(j_0, i_0, q_0) = \|\frac{i_0 - j_0}{3}\|$ results in (after simplification) the following LB for B

$$B^{lb}(j_0, i_0, q_0) = \frac{1}{2} * \|\frac{i_0 - j_0}{3}\| * \|\frac{i_0 - j_0}{3} - 1\| + \|\frac{i_0 - j_0}{3}\| * \|j_0\|$$

When the best-case CR approach leads to imprecise bounds, which happens when the reduction operator obtains trivial reductions (i.e., 0), we can apply the alternative method that is based on analyzing each C_i separately. Namely, we infer a LB ${C_i}^{lb}(\bar{x}_0)$ for each CR C_i, and then $C^{lb}(\bar{x}_0) = \sum_{i=1}^{h} {C_i}^{lb}(\bar{x}_0)$ is clearly a sound LB for C. The technical details for solving each ${C_i}^{lb}(\bar{x}_0)$ are identical to those of the UB case: (1) instead of using $\check{f}_C(\bar{x}_0)$, we should use $\check{f}_{C_i}(\bar{x}_0)$ which under-approximates the number of applications of the i-th equation. This is done by modifying Definition 5.1 such that it counts only the applications of the i-th equation instead of all equations; and (2) the progression parameter $\check{d}$ is the same as in the case of UB, i.e., we consider subsequent, rather than consecutive, applications of the i-th equation. It is important to note that this approach can be applied only when $m_1 = 1$. Algorithms that summarize the above approaches can be derived in a very similar way to algorithms 2 and 3, simply by considering the dual notions.

5.3. Concluding Remarks

We have presented a practical and precise approach for inferring LBs on CRs. When considering CRs with a single recursive equation, in practice, our approach achieves an optimal precision. As regards CRs with multiple recursive equations, we have presented a solution which is effective in many cases, however, it is less effective than its UB counterpart. This is expected, as indeed, the problem of inferring LBs is far more complicated than inferring UBs. It is important to note that this is the first work that attempts to automatically infer lower-bounds for CRs that originate from real programs. Our approach for inferring LBs is not limited to $\|.\|$ expressions with linear and geometric behavior, but can be adapted to any behavior that can be modeled with sequences. Handling (multiple) base-case equations with non-zero cost can be done as for the case of UBs, but considering the dual notions.

As in the case of UBs, the output of CAS, when solving a best-case RR, might not comply with the grammar of cost expressions as specified in Section 2. Concretely, after normalization, it might include sub-expressions of the form $-e$ where e is a multiplication of basic cost expressions. Unlike the case of UBs, for LBs it is not sound to remove such expression as it results in a bigger one. Removing them requires changing other subexpression in order to compensate on $-e$. E.g., $\|x\|^2 - \|x\|$ can be rewritten to $\|x-1\| * \|x\|$.

6. EXPERIMENTS

We have implemented our techniques as a component in PUBS (Practical Upper Bound Solver) that implements the techniques proposed in Albert et al. [2011b]. PUBS is used as backend solver in COSTA (a COSt and Termination Analyzer for Java bytecode) [Albert et al. 2011b]. This means that our solver can be used to solve (i) CRs that are automatically generated by COSTA from Java (bytecode) programs; or (ii) CRs provided by the user. The obtained RRs are solved using MAXIMA [Maxima 2009]. Our implementation (and examples) can be tried out at `http://costa.ls.fi.upm.es/costa` where the user is expected to provide a Java (bytecode) programs as input, or at `http://costa.ls.fi.upm.es/pubs` where the input is given as a CR. In both cases the option `series` should be selected. In our experiments we apply our implementation on Java programs via COSTA.

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; and PascalTriangle computes and prints Pascal's Triangle.

Table I illustrates the accuracy and efficiency on the above benchmarks using the cost model "*number of executed (bytecode) instructions*". In the second column, (A) is the UB obtained using the approach of Albert et al. [2011b], and (B) and (C) are respectively the UB and LB obtained by our solver. In order to facilitate the comparison of the UBs and LBs of Table I, we also provide the graphical representations in Figure 5. As regards UBs, we improve the precision over Albert et al. [2011b] in all benchmarks.

#	UBs and LBs	T
1	**(A)** $24 \cdot \lVert a{-}1\rVert^3{+}36 \cdot \lVert a{-}1\rVert^2{+}18 \cdot \lVert a\rVert^2{+}30 \cdot \lVert a\rVert \cdot \lVert a{-}1\rVert{+}35 \cdot \lVert a{-}1\rVert{+}72 \cdot \lVert a\rVert{+}54$	1.8
	(B) $8 \cdot \lVert a{-}1\rVert^3{+}18 \cdot \lVert a\rVert^2{+}45 \cdot \lVert a{-}1\rVert^2{+}72 \cdot \lVert a\rVert{+}102 \cdot \lVert a{-}1\rVert{+}54$	3.3
	(C) $8 \cdot \lVert a{-}1\rVert^3{+}16 \cdot \lVert a\rVert^2{+}43 \cdot \lVert a{-}1\rVert^2{+}55 \cdot \lVert a\rVert{+}96 \cdot \lVert a{-}1\rVert{+}54$	2.6
2	**(A)** $24 \cdot \lVert c{-}1\rVert^3{+}18 \cdot \lVert c\rVert^2{+}36 \cdot \lVert c{-}1\rVert^2{+}30 \cdot \lVert c{-}1\rVert \cdot \lVert c\rVert{+}35 \cdot \lVert c-1\rVert{+}25 \cdot \lVert c\rVert{+}$ $48 \cdot \lVert b{-}1\rVert^2{+}46 \cdot \lVert b{-}1\rVert{+}74$	1.9
	(B) $8 \cdot \lVert c{-}1\rVert^3{+}18 \cdot \lVert c\rVert^2{+}45 \cdot \lVert c{-}1\rVert^2{+}25 \cdot \lVert c\rVert{+}102 \cdot \lVert c{-}1\rVert{+}24 \cdot \lVert b{-}1\rVert^2{+}92 \cdot \lVert b{-}1\rVert$ $+74$	3.5
	(C) $8 \cdot \lVert c{-}1\rVert^3{+}16 \cdot \lVert c\rVert^2{+}43 \cdot \lVert c{-}1\rVert^2{+}25 \cdot \lVert c\rVert{+}96 \cdot \lVert c{-}1\rVert{+}24 \cdot \lVert b{-}1\rVert^2{+}92 \cdot \lVert b{-}1\rVert$ $+74$	2.8
3	**(A)** $24 \cdot \lVert a{-}1\rVert^3{+}56 \cdot \lVert a\rVert \cdot \lVert a{-}1\rVert^2{+}18 \cdot \lVert a\rVert^2{+}46 \cdot \lVert a{-}1\rVert^2{+}75 \cdot \lVert a\rVert{+}68 \cdot \lVert a\rVert \cdot \lVert a{-}1\rVert$ $+49 \cdot \lVert a{-}1\rVert{+}62$	3.6
	(B) $8 \cdot \lVert a{-}1\rVert^3{+}28 \cdot \lVert a\rVert \cdot \lVert a{-}1\rVert^2{+}18 \cdot \lVert a\rVert^2{+}50 \cdot \lVert a{-}1\rVert^2{+}92 \cdot \lVert a\rVert \cdot \lVert a{-}1\rVert{+}75 \cdot \lVert a\rVert$ $+121 \cdot \lVert a{-}1\rVert{+}62$	4.6
	(C) $8 \cdot \lVert a{-}1\rVert^3{+}28 \cdot \lVert a\rVert \cdot \lVert a{-}1\rVert^2{+}16 \cdot \lVert a\rVert^2{+}48 \cdot \lVert a{-}1\rVert^2{+}92 \cdot \lVert a\rVert \cdot \lVert a{-}1\rVert{+}75 \cdot \lVert a\rVert$ $+115 \cdot \lVert a{-}1\rVert{+}62$	3.8
4	**(A)** $25 \cdot \lVert b\rVert^2 \cdot \lVert b-1\rVert{+}30 \cdot \lVert b\rVert^2{+}16 \cdot \lVert b\rVert{+}6$	0.1
	(B) $\frac{25}{3} \cdot \lVert b\rVert^3{+}15 \cdot \lVert b\rVert^2{+}\frac{68}{3} \cdot \lVert b\rVert{+}6$	0.2
	(C) $\frac{21}{2} \cdot \lVert b\rVert^2{+}\frac{53}{2} \cdot \lVert b\rVert{+}6$	0.0
5	**(A)** $19 \cdot \lVert b{-}1\rVert^2 + 25 \cdot \lVert b{-}1\rVert + 7$	0.1
	(B) $\frac{19}{2} \cdot \lVert b{-}1\rVert^2 + \frac{69}{2} \cdot \lVert b{-}1\rVert + 7$	0.1
	(C) $18 \cdot \lVert b{-}1\rVert + 7$	0.1
6	**(A)** $37 \cdot \lVert b+1\rVert \cdot \lVert 2b{-}1\rVert + 53 \cdot \lVert 2b{-}1\rVert + 11$	1.2
	(B) $37 \cdot \lVert b{+}1\rVert \cdot log_2(\lVert 2b{-}1\rVert{+}1) + 53 \cdot \lVert 2b{-}1\rVert + 11$	2.0
	(C) 4	1.2
7	**(A)** $27 \cdot \lVert a{-}1\rVert^2 + 16 \cdot \lVert a{-}1\rVert + 9$	0.1
	(B) $\frac{27}{2} \cdot \lVert a{-}1\rVert^2 + \frac{59}{2} \cdot \lVert a{-}1\rVert + 9$	0.2
	(C) $\frac{13}{2} \cdot \lVert a{-}1\rVert^2 + \frac{45}{2} \cdot \lVert a{-}1\rVert + 9$	0.2
8	**(A)** $30 \cdot \lVert a\rVert^2 + 27 \cdot \lVert a{-}1\rVert^2 + 33 \cdot \lVert a\rVert + 10 \cdot \lVert a{-}1\rVert + 25$	0.7
	(B) $\frac{41}{2} \cdot \lVert a\rVert^2 + 27 \cdot \lVert a{-}1\rVert^2 + 10 \cdot \lVert a{-}1\rVert + \frac{85}{2} \cdot \lVert a\rVert + 25$	0.9
	(C) $\frac{41}{2} \cdot \lVert a\rVert^2 + 27 \cdot \lVert a{-}1\rVert^2 + 10 \cdot \lVert a{-}1\rVert + \frac{85}{2} \cdot \lVert a\rVert + 25$	0.9
9	**(A)** $34 \cdot \lVert c\rVert^2 + 12 \cdot \lVert c\rVert + 8$	0.1
	(B) $17 \cdot \lVert c\rVert^2 + 29 \cdot \lVert c\rVert + 8$	0.3
	(C) $8 \cdot \lVert c\rVert^2 + 20 \cdot \lVert c\rVert + 8$	0.2

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

This improvement, in all benchmarks except MergeSort, is due to nested loops where 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. Moreover, our UBs are very close to the real cost (the difference is only in some constants). In Figure 5, it can be seen that the precision gain is greater for larger values of the inputs.

For MergeSort, we obtain a tight bound in the order of $b * log_2(b)$. Note that Albert et al. [2011b] could obtain $b * log_2(b)$ 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. As regards LBs, it can be observed from row C of each benchmark that we

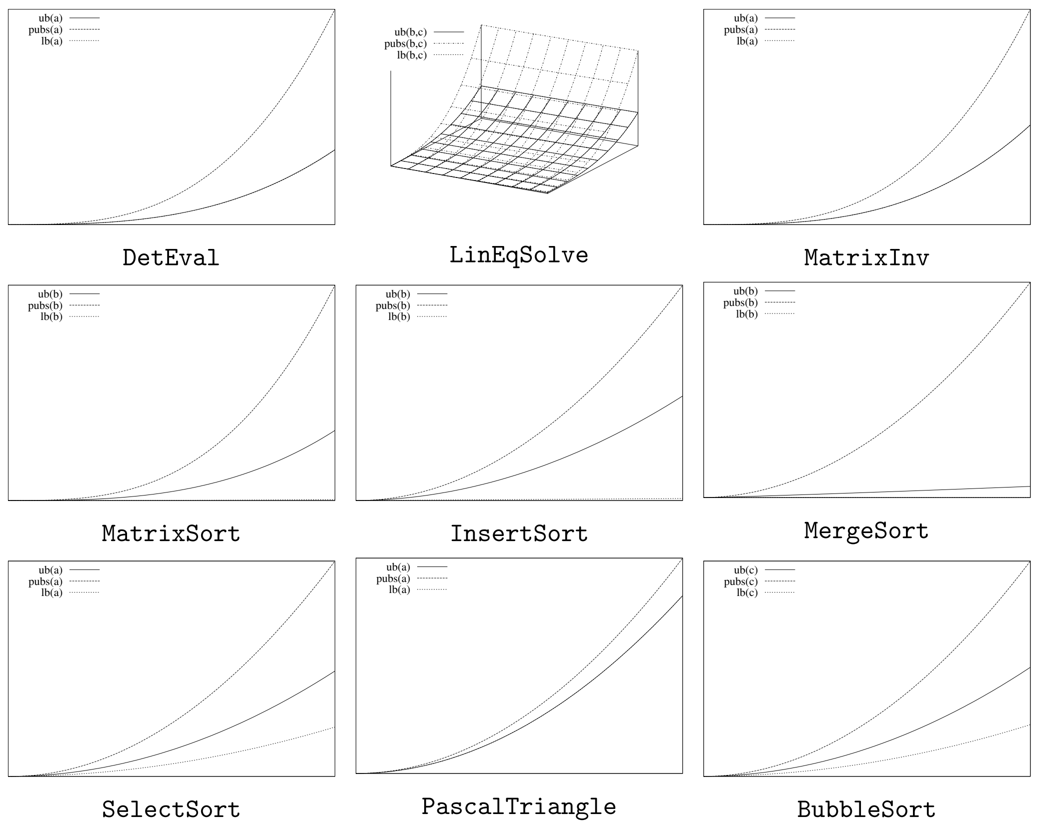

Fig. 5: Graphs for UBs and LBs of Table I. The plot `pubs` is the UB obtained by Albert et al. [2011b], the plots `ub` and `lb` are our UB and LB, respectively.

obtain non-trivial LBs in all cases, except in MergeSort. For MergeSort, the LB on loop iterations is logarithmic which cannot be inferred by our linear invariant generation tool and hence we get the trivial bound 4. Note that for InsertSort we infer a linear LB which happens when the array is sorted. In the last column of each benchmark, T shows the time (in seconds) to compute the bounds of (A),(B) and (C) from the generated CR. Our approach is slightly slower mainly due to the overhead of connecting PUBS to MAXIMA.

We have also compared experimentally our approach to the analysis of Hoffmann et al. [2011], which was developed in parallel to our work. The comparison is made on their examples, which are available at `http://raml.tcs.ifi.lmu.de`. These examples are written in a first-order functional languages (RAML). In order to perform a fair comparison, we have done the following: (i) RAML programs are first translated into equivalent CRs; and (ii) we used a cost model that counts the number of visits to a specific point (functions entries) as this can be easily defined in RAML. The detailed results are available at `http://costa.ls.fi.upm.es/pubs` in the examples section. The main conclusions drawn from the comparison are: (1) in most cases we are as precise as their analysis, and sometimes the results differ only in the constants; (2) for QuickSort our analysis fails to infer the precise bound, this is because the input list is divided into two lists of different length; and (3) for MergeSort, our analysis is more precise since they cannot infer bounds with logarithmic expressions.

7. RELATED WORK

In this section we discus the most related approaches on automatic inference of symbolic UBs [Wegbreit 1975; Albert et al. 2011b; Hofmann and Jost 2003; Hoffmann and Hofmann 2010; Hoffmann et al. 2011; Gulwani et al. 2009].

Automatic resource bound analysis dates back to the pioneer work of Wegbreit [Wegbreit 1975], where an automatic analysis for simple LISP programs were developed and implemented in a tool called Metric. Given a program and a cost model, Metric generates difference equations in several phases that capture the cost (with respect to the cost model) of the program. Then several methods were provided to solve these equations into closed-form expressions. Metric mostly deals with programs that manipulate data structures, such as lists, where termination depends on the length of such data structures. Our approach is more general, since it uses a more general setting in which variables have integer values (which can refer to sizes of data structures), and moreover allows the relations between variables to be expressed as linear constraints.

When comparing to the work of Albert et al. [2011b], although experimentally our approach is more precise (as we have seen in Section 6), we cannot prove theoretically that it is always more precise. However, for the case of CRs with a single recursive equation as described in sections 4.1 and 4.2, if we use the same ranking functions and maximization procedures as they do, then it is guaranteed that our approach is more precise. For the case of CRs with multiple recursive equations, it is not possible to formally compare them. Indeed, one could handcraft examples for which their approach infers more precise UBs. This is because for solving such cases: (1) our first alternative, which generalizes cost expression, is based on heuristics and thus might be imprecise in some cases; and (2) our second alternative, which analyzes each recursive equation separately, requires inferring the number of visits to a single equation which can be less precise than inferring ranking functions. As regards applicability, first note that when it is not possible to infer the progression parameters (in Definition 4.6 and 4.13), we basically use the approach of Albert et al. [2011b], i.e., replacing the corresponding $\|l\|$ by $\|\hat{l}\|$. Assuming that CAS is able to handle the corresponding RRs we achieve a similar applicability.

There are several techniques that are centered on the static inference of resource usage of first-order functional programs [Hofmann and Jost 2003; Hoffmann and Hofmann 2010; Hoffmann et al. 2011]. The techniques of Hofmann and Jost [2003] can infer only linear UBs, and, they have been extended by Hoffmann and Hofmann [2010] for univariate polynomial UBs. However, such polynomial cannot express bounds of the form $m*n$, and thus they are over-approximated by n^2+m^2. Hoffmann et al. [2011] developed techniques for handling multivariate polynomial UBs, such as $m*n$. All these approaches cannot handle programs whose resource usage depend on integer variables (mainly because of negative values). While these techniques can be adapted to handle CRs with simple integer linear constraints, it is not clear how it can be extended to handle CRs with unrestricted form of integer linear constraints. It is also important to note that currently these techniques cannot compute *logarithmic* or *exponential* UBs. For example, Hoffmann et al. [2011] computes $O(n^2)$ as UB for the `mergesort` program whereas we compute $O(n*log(n))$. On the other hand, these techniques are superior for examples that exhibit *amortized cost* behavior, but such examples are out of the scope of this paper since they cannot be modeled precisely with CRs [Alonso-Blas and Genaim 2012]. It is also important to note that these techniques fail to infer an UB if the cost cannot be expressed using a polynomial with degree less than a given integer k (the user has to provide this k). Our analysis does not have such restriction. Overall, we believe that our approach is more generic, in the sense that it handles *CRs* with arbitrary integer linear constraints, which might be the output of cost analysis of any programming language, and, in addition, it is not restricted to any complexity class.

The work of Gulwani et al. [2009] in the SPEED project computes *worst-case* symbolic bounds for C++ code containing loops and recursion. The loops in the input code are instrumented with counters and a linear invariant generation tool is used to compute the bound of those counters. The computed bounds for the individual loops are then composed using a counter-optimal proof structure to compute the total bound which can be nonlinear or disjoint. 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 their work the sum $\sum_{i=1}^{n} i$ is over-approximated by n^2, while our approach is able to obtain the precise solution $\frac{n^2}{2} - \frac{n}{2}$.

There are some works related on solving recurrence relations [Cohen and Katcoff 1977; Bagnara et al. 2003; Gosper 1978; Everest et al. 2003]. Cohen and Katcoff [1977] developed interactive techniques based on guessing solutions and generating functions that solve linear recurrence relations. However, their method does not always admit closed-form expressions. Decision procedures have been developed by Gosper [1978] and Everest et al. [2003] which admits closed-form expression for a subset of linear recurrence relations. Bagnara et al. [2003] extended the previous techniques to recurrence relations with multiple arguments and some nonlinear recurrences of finite order. However, in spite of this extension, still the form of the recurrence relations that can be solved are very limited, when compared to our *CRs*.

8. CONCLUSIONS AND FUTURE WORK

We have proposed a novel approach to infer precise UBs and LBs 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 (resp. best-case) *RRs* that *CAS* can accurately solve to obtain UBs (resp. LBs) on the resource consumption. The required transformation is far from trivial since it requires transforming multiple recursive non-deterministic equations involving multiple increasing and decreasing arguments into a single deterministic equation with a single decreasing argument.

Importantly, since *CRs* are universal output of cost analysis for any programming language, our approach to infer closed-form UBs and LBs is completely independent of the programming language from which *CRs* are obtained. Currently, we have applied it to *CRs* obtained from Java bytecode programs, from X10 programs [Albert et al. 2011c] and from actor-based programs [Albert et al. 2011a]. In the latter two cases, the languages have concurrency primitives to spawn asynchronous tasks and to wait for termination of tasks. In spite of being concurrent languages, the first phase of cost analysis handles the concurrency primitives and the generated *CRs* can be solved directly using our approach.

As future work, we plan to assess the scalability of our approach by analyzing larger programs, up to now the main concern has been the accuracy of the results obtained. Also, 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, our approach will directly be benefited from any improvement in this regard.

ACKNOWLEDGMENT

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

REFERENCES

ADACHI, A., KASAI, T., AND MORIYA, E. 1979. A theoretical study of the time analysis of programs. In *MFCS*, J. Becvár, Ed. Lecture Notes in Computer Science Series, vol. 74. Springer, 201–207.

ALBERT, E., ARENAS, P., GENAIM, S., GÓMEZ-ZAMALLOA, M., AND PUEBLA, G. 2011a. Cost Analysis of Concurrent OO programs. In *The 9th Asian Symposium on Programming Languages and Systems (APLAS'11)*. Vol. 7078. Springer, 238–254.

ALBERT, E., ARENAS, P., GENAIM, S., HERRAIZ, I., AND PUEBLA, G. 2010. Comparing cost functions in resource analysis. In *1st International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA'09)*. Lecture Notes in Computer Science Series, vol. 6234. Springer, 1–17.

ALBERT, E., ARENAS, P., GENAIM, S., AND PUEBLA, G. 2011b. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning 46,* 2, 161–203.

ALBERT, E., ARENAS, P., GENAIM, S., PUEBLA, G., AND ZANARDINI, D. 2009. Resource usage analysis and its application to resource certification. In *9th International School on Foundations of Security Analysis and Design (FOSAD'09)*. Number 5705 in Lecture Notes in Computer Science. Springer, 258–288.

ALBERT, E., ARENAS, P., GENAIM, S., PUEBLA, G., AND ZANARDINI, D. 2012. Cost Analysis of Object-Oriented Bytecode Programs. *Theoretical Computer Science 413,* 1, 142–159.

ALBERT, E., ARENAS, P., GENAIM, S., AND ZANARDINI, D. 2011c. Task-level analysis for a language with async/finish parallelism. In *LCTES*, J. Vitek and B. D. Sutter, Eds. ACM, 21–30.

ALBERT, E., GENAIM, S., AND MASUD, A. N. 2011d. More Precise yet Widely Applicable Cost Analysis. In *12th Verification, Model Checking, and Abstract Interpretation (VMCAI'11)*, R. Jhala and D. Schmidt, Eds. Lecture Notes in Computer Science Series, vol. 6538. Springer Verlag, 38–53.

ALONSO-BLAS, D. E. AND GENAIM, S. 2012. On the limits of the classical approach to cost analysis. In *SAS*, A. Miné and D. Schmidt, Eds. Lecture Notes in Computer Science Series, vol. 7460. Springer, 405–421.

BAGNARA, R., HILL, P. M., AND ZAFFANELLA, E. 2008. 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.

BAGNARA, R., ZACCAGNINI, A., AND ZOLO, T. 2003. The automatic solution of recurrence relations. I. Linear recurrences of finite order with constant coefficients. Quaderno 334, Dipartimento di Matematica, Università di Parma, Italy. Available at `http://www.cs.unipr.it/Publications/`.

BENZINGER, R. 2004. Automated Higher-Order Complexity Analysis. *Theoretical Computer Science 318,* 1-2, 79–103.

COHEN, J. AND KATCOFF, J. 1977. Symbolic solution of finite-difference equations. *ACM Trans. Math. Softw. 3,* 3, 261–271.

COUSOT, P. AND HALBWACHS, N. 1978. Automatic Discovery of Linear Restraints among Variables of a Program. In *The 5th ACM Symposium on Principles of Programming Languages (POPL'78)*. ACM Press, 84–97.

CRARY, K. AND WEIRICH, S. 2000. Resource Bound Certification. In *Proc. of POPL'00*. ACM, 184–198.

DEBRAY, S. K. AND LIN, N. W. 1993. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems 15,* 5, 826–875.

EVEREST, G., VAN DER POORTEN, A., SHPARLINSKI, I., AND WARD, T. 2003. *Recurrence sequences*. American Mathematical Society, Providence, R.I. :.

FEAUTRIER, P. 1988. Parametric integer programming. *RAIRO Recherche Opérationnelle 22,* 3, 243–268.

GOSPER, JR., R. W. 1978. Decision procedure for indefinite hypergeometric summation. *Proceedings of the National Academy of Sciences of the United States of America 75,* 1, 40–42.

GULWANI, S., MEHRA, K. K., AND CHILIMBI, T. M. 2009. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *The 36th Symposium on Principles of Programming Languages (POPL'09)*. ACM, 127–139.

HOFFMANN, J., AEHLIG, K., AND HOFMANN, M. 2011. Multivariate Amortized Resource Analysis. In *The 38th Symposium on Principles of Programming Languages (POPL'11)*. ACM, 357–370.

HOFFMANN, J. AND HOFMANN, M. 2010. Amortized Resource Analysis with Polynomial Potential. In *The 19th European Symposium on Programming (ESOP'10)*. Lecture Notes in Computer Science Series, vol. 6012. Springer, 287–306.

HOFMANN, M. AND JOST, S. 2003. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *30th Symposium on Principles of Programming Languages (POPL'03)*. ACM Press.

KOWALSKI, R. A. 1974. Predicate Logic as a Programming Language. In *Proceedings IFIPS*. 569–574.

LE METAYER, D. 1988. ACE: An Automatic Complexity Evaluator. *ACM Transactions on Programming Languages and Systems 10,* 2, 248–266.

LUCA, B., ANDREI, S., ANDERSON, H., AND KHOO, S.-C. 2006. Program transformation by solving recurrences. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '06)*. ACM, 121–129.

Maxima 2009. *Maxima, a Computer Algebra System.* `http://maxima.sourceforge.net`.

NAVAS, J., MERA, E., LÓPEZ-GARCÍA, P., AND HERMENEGILDO, M. 2007. User-Definable Resource Bounds Analysis for Logic Programs. In *23rd International Conference on Logic Programming (ICLP'07)*. Lecture Notes in Computer Science Series, vol. 4670. Springer.

ROSENDAHL, M. 1989. Automatic Complexity Analysis. In *4th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*. ACM Press.

SANDS, D. 1995. A Naïve Time Analysis and its Theory of Cost Equivalence. *Journal of Logic and Computation 5,* 4, 495–541.

SCHRIJVER, A. 1986. *Theory of linear and integer programming*. John Wiley & Sons, Inc., New York, NY, USA.

WADLER, P. 1988. Strictness Analysis Aids Time Analysis. In *ACM Symposium on Principles of Programming Languages (POPL'88)*. ACM Press.

WEGBREIT, B. 1975. Mechanical Program Analysis. *Communications of the ACM 18,* 9, 528–539.

A. PROOFS OF THEOREMS

A.1. Theorem 4.2

For any initial values $\bar{x}_0$, the evaluation tree T_1 of $P_C(\hat{f}_C(\bar{x}_0))$ is a complete tree such that any path (from the root to a leaf) has *exactly* $\hat{f}_C(\bar{x}_0)$ internal nodes (i.e., all nodes but the leaf) with cost e. Any evaluation tree $T_2 \in \mathcal{T}(C(\bar{x}_0))$ is a (possibly not complete) tree such that any path (from the root to a leaf) has *at most* $\hat{f}_C(\bar{x}_0)$ internal nodes, and each internal node has cost e. Thus, since the recursive equations of P_C and C have m recursive calls each, it holds that $sum(T_1) \geq sum(T_2)$ and therefore $P_C(\hat{f}_C(\bar{x}_0)) \geq C(\bar{x}_0)$.

A.2. Theorem 4.8

We prove the theorem for the case when e is linearly decreasing. The case of linearly increasing is dual. Let $T_1 \in \mathcal{T}(C(\bar{x}_0))$, and T_2 be the evaluation tree of $P_C(\hat{f}_C(\bar{x}_0))$. Observe that: (1) The leaves have cost 0; (2) The number of *internal nodes* in any path from the root to a leaf in T_1 is at most $\hat{f}_C(\bar{x}_0)$, and in T_2 is exactly $\hat{f}_C(\bar{x}_0)$; (3) The RR P_C and the CR C have the same number of recursive calls in their recursive equation; and (4) $\hat{E}_e$ and e are identical up to their $\|.\|$ components since $\hat{E}_e$ is obtained from e by replacing each $\|l\| \in e$ by $\|\hat{l} - l' * \check{d}\| + N * \check{d}$. These observations, together with the fact that cost expressions are monotonic, implies that in order to prove the theorem, it is enough to prove that for any $\|l\| \in e$ and its corresponding $L = \|\hat{l} - l' * \check{d}\| + N * \check{d}$ in $\hat{E}_e$, if $\|l_1\|, \ldots, \|l_\kappa\|$ are instances of $\|l\|$ in a given path in T_1, and $L_1, \ldots, L_{\hat{f}_C(\bar{x}_0)}$ are those of L in T_2, then $L_i \geq \|l_i\|$ for any $1 \leq i \leq \kappa$. Recall that $\kappa \leq \hat{f}_C(\bar{x}_0)$.

Base Case:. L_1 is obtained when $N = \hat{f}_C(\bar{x}_0)$, therefore $L_1 = \|\hat{l} - l' * \check{d}\| + \hat{f}_C(\bar{x}_0) * \check{d} \geq \|\hat{l}\| - \|l'\| * \check{d} + \hat{f}_C(\bar{x}_0) * \check{d} = \|\hat{l}\| \geq \|l_1\|$. Ra call that $\|l'\| = \hat{f}_C(\bar{x}_0)$.

Inductive Case:. First, we assume that $L_i \geq \|l_i\|$. Next, we consider two cases: (1) if $l_i \geq \hat{d}$, then $\|l_{i+1}\| \leq \|l_i\| - \check{d} \leq L_i - \check{d} = \|\hat{l} - l' * \check{d}\| + (\hat{f}_C(\bar{x}_0) - (i-1)) * \check{d} - \check{d} = \|\hat{l} - l' * \check{d}\| + (\hat{f}_C(\bar{x}_0) - i) * \check{d} = L_{i+1}$; and (2) if $l_i < \hat{d}$, then $\|l_{i+1}\| = 0 \leq \|\hat{l} - l' * \check{d}\| + (\hat{f}_C(\bar{x}_0) - i) * \check{d} = L_{i+1}$.

A.3. Theorem 4.14

The proof is similar to the one of Theorem 4.8. We prove the theorem for the case when e is geometrically decreasing. The case of geometrically increasing is dual. Let $T_1 \in \mathcal{T}(C(\bar{x}_0))$ and T_2 be the evaluation tree of $P_C(\hat{f}_C(\bar{x}_0))$. The observations about T_1, T_2 and the monotonicity property in the proof of Theorem 4.8 also hold for this case, and therefore it is enough to prove that for any $\|l\| \in e$ and its corresponding $L = \frac{\|\hat{l}\|}{\check{r}^{(\hat{f}_C(\bar{x}_0) - N)}} + \|{-\check{p}}\| * \hat{S}(\hat{f}_C(\bar{x}_0) - N)$ in $\hat{E}_e$, if $\|l_1\|, \ldots, \|l_\kappa\|$ are instances of $\|l\|$ in a given path in T_1, and $L_1, \ldots, L_{\hat{f}_C(\bar{x}_0)}$ are those of L in T_2, then $L_i \geq \|l_i\|$ for any $1 \leq i \leq \kappa$.

Base Case:. L_1 is obtained when $N = \hat{f}_C(\bar{x}_0)$, therefore $L_1 = \frac{\|\hat{l}\|}{\check{r}^{(\hat{f}_C(\bar{x}_0) - \hat{f}_C(\bar{x}_0))}} + \|{-\check{p}}\| * \hat{S}(\hat{f}_C(\bar{x}_0) - \hat{f}_C(\bar{x}_0)) = \|\hat{l}\| \geq \|l_1\|$ since $\hat{S}(0) = 0$.

Inductive Case:. We assume that $L_i \geq \|l_i\|$ and will prove that $L_{i+1} \geq \|l_{i+1}\|$. We have $\hat{S}(i) = \frac{1}{\check{r}^{(i)}} * \frac{1}{1-\check{r}} - \frac{1}{1-\check{r}} = \frac{1}{\check{r}} * (\frac{1}{\check{r}^{(i-1)}} * \frac{1}{1-\check{r}} - \frac{1}{1-\check{r}}) + \frac{1}{\check{r} * (1-\check{r})} - \frac{1}{(1-\check{r})} = \frac{1}{\check{r}} * \hat{S}(i-1) + \frac{1}{\check{r}}$.

We also have $L_i = \frac{\|\hat{l}\|}{\check{r}^{i-1}} + \|{-\check{p}}\| * \hat{S}(i-1)$. Then, the following equations hold:

$$\begin{array}{lll}
L_{i+1} & = & \frac{\|\hat{l}\|}{\check{r}^{i}} + \|{-\check{p}}\| * \hat{S}(i) \\
 & = & \frac{1}{\check{r}} * \frac{\|\hat{l}\|}{\check{r}^{i-1}} + \|{-\check{p}}\| * (\frac{1}{\check{r}} * \hat{S}(i-1) + \frac{1}{\check{r}}) \\
 & = & \frac{1}{\check{r}} * (\frac{\|\hat{l}\|}{\check{r}^{i-1}} + \|{-\check{p}}\| * \hat{S}(i-1)) + \|{-\check{p}}\| * \frac{1}{\check{r}} \\
 & = & \frac{1}{\check{r}} * (L_i + \|{-\check{p}}\|) \\
 & \geq & \frac{1}{\check{r}} * (\|l_i\| + \|{-\check{p}}\|) \\
 & \geq & \|l_{i+1}\| \quad \text{[Since } \|{-\check{p}}\| \geq -\check{p} \text{ and } \varphi_1 \models l_i \geq \check{r} * l_{i+1} + \check{p}]
\end{array}$$

A.4. Theorem 4.18

Intuitively, since by definition the evaluation trees of $\hat{C}$ up to depth $\hat{f}_C(\bar{x}_0)$ over-approximate those of C, then, by the construction of $\hat{E}_e$, it is guaranteed that the evaluation tree of $\langle P_C(N) = \hat{E}_e + m_h * P_C(N-1)\rangle$ up to depth $\hat{f}_C(\bar{x}_0)$ over-approximates C. Therefore, if E is a solution for P_C then $E[N/\hat{f}_C(\bar{x}_0)]$ is an UB for $C(\bar{x}_0)$.

A.5. Theorem 5.5

In order to prove the above theorem, it is enough to prove that if the costs contributed by $C(\bar{x}_0)$ and $P_C(N)$ along any corresponding chain of calls are $e_1, \cdots, e_{\kappa_0}$ and $\check{E}_{e_1}, \cdots, \check{E}_{e_\kappa}$ respectively, it holds that $\check{E}_{e_i} \leq e_i$ for all $1 \leq i \leq \kappa$ and $\kappa \leq \kappa_0$. Since N is instantiated by $\check{f}_C(\bar{x}_0)$ to the solution E of $P_C(N)$, any chain of calls of $P_C(N)$ is exactly $\check{f}_C(\bar{x}_0)$ (i.e. $\kappa = \check{f}_C(\bar{x}_0)$). According to Corollary 5.2, $\check{f}_C(\bar{x}_0)$ is the *lower-bound* on the length of any chain of $C(\bar{x}_0)$ and hence $\kappa \leq \kappa_0$ holds in general. Again, since cost expression e (and hence its corresponding RR expression $\check{E}_e$) follows the monotonicity property, in order to prove $\check{E}_{e_i} \leq e_i$, it is enough to prove the relation for their $\|.\|$ sub-component. That means, if $\|l_1\|, \cdots, \|l_{\kappa_0}\|$ are instances of $\|l\| \in e$ in the chain of calls $e_1, \cdots, e_{\kappa_0}$ and $L_1, \cdots, L_{\check{f}_C(\bar{x}_0)}$ are the instances of the replacements of $\|l\|$ in $\check{E}_e$ according to Definition 5.4 along the chain of calls $\check{E}_{e_1}, \cdots, \check{E}_{e_\kappa}$, it is enough to prove that $L_i \leq \|l_i\|$ for all $1 \leq i \leq \check{f}_C(\bar{x}_0)$.

Base Case:. The comparison of L_1 and $\|l_1\|$ are done by the following case analysis as done for the replacement of $\|l\|$ in Definition 5.4.

(1) We obtain L_1 when $N = \check{f}_C(\bar{x}_0)$ since $\|l\|$ is linearly increasing. $L_1 = \|\check{l}\| + (\check{f}_C(\bar{x}_0) - \check{f}_C(\bar{x}_0)) * \check{d} = \|\check{l}\| \leq \|l_1\|$.
(2) In this case $N = 1$ since $\|l\|$ is linearly decreasing and $L_1 = \|\check{l}\| + (1-1) * \check{d} = \|\check{l}\| \leq \|l_1\|$.
(3) Here, $N = \check{f}_C(\bar{x}_0)$ and $L_1 = \check{r}^{(\check{f}_C(\bar{x}_0) - \check{f}_C(\bar{x}_0))} * \|\check{l}\| + \check{p} * \check{S}(\check{f}_C(\bar{x}_0) - \check{f}_C(\bar{x}_0)) = \|\check{l}\| \leq \|l_1\|$ since $\check{S}(0) = 0$.
(4) Here, $N = 1$ and $L_1 = \check{r}^{(1-1)} * \|\check{l}\| + \check{p} * \check{S}(1-1) = \|\check{l}\| \leq \|l_1\|$.
(5) $\|\check{l}\| \leq \|l_1\|$.

Inductive Case:. Here we assume that $L_i \leq \|l_i\|$ and will prove that $L_{i+1} \leq \|l_{i+1}\|$. We do the similar case analysis.

(1) For L_i, we have $N = \check{f}_C(\bar{x}_0) - i + 1$. So, $L_i = \|\check{l}\| + (\check{f}_C(\bar{x}_0) - (\check{f}_C(\bar{x}_0) - i + 1)) * \check{d} = \|\check{l}\| + (i-1) * \check{d}$. Then $L_{i+1} = \|\check{l}\| + (\check{f}_C(\bar{x}_0) - \check{f}_C(\bar{x}_0) + i) * \check{d} = \|\check{l}\| + i * \check{d} = \|\check{l}\| + (i-1) * \check{d} + \check{d} = L_i + \check{d} \leq \|l_i\| + \check{d} \leq \|l_{i+1}\|$ since $\check{d}$ is the minimum distance of $\|l\|$ and $\check{l} \geq 0$.
(2) Here, we have $N = i$ for L_i and $N = i + 1$ for L_{i+1}. Then the proof is similar to the proof of (1).

(3) For L_i and L_{i+1}, $N = \check{f}_C(\bar{x}_0) - i + 1$ and $N = \check{f}_C(\bar{x}_0) - i$ respectively. Thus we obtain $L_i = \check{r}^{(i-1)} * \|\check{l}\| + \check{p} * \check{S}(i-1)$ and $L_{i+1} = \check{r}^i * \|\check{l}\| + \check{p} * \check{S}(i)$. We also have $\check{S}(i) = \frac{\check{r}^i - 1}{\check{r} - 1} = \check{r} * \frac{\check{r}^{i-1} - 1}{\check{r} - 1} + 1 = \check{r} * \check{S}(i-1) + 1$. Then $L_{i+1} = \check{r}^i * \|\check{l}\| + \check{p} * \check{S}(i) = \check{r} * (\check{r}^{(i-1)} * \|\check{l}\| + \check{p} * \check{S}(i-1)) + p = \check{r} * L_i + \check{p} \leq \check{r} * \|l_i\| + \check{p} \leq \|l_{i+1}\|$ [since $\check{l} \geq 0$ and $\|l\|$ has the geometric progression behavior as defined in Definition 4.11].

(4) For L_i and L_{i+1}, $N = i$ and $N = i + 1$ respectively and the proof is similar to the proof of (3).

(5) $\|\check{l}\| \leq \|l_{i+1}\|$.

A.6. Theorem 5.8

Intuitively, since by definition the evaluation trees of $\check{C}$ up to depth $\check{f}_C(\bar{x}_0)$ under-approximate those of C, then, by the construction of $\check{E}_e$, it is guaranteed that the evaluation tree of $\langle P_C(N) = \check{E}_e + m_h * P_C(N-1) \rangle$ up to depth $\check{f}_C(\bar{x}_0)$ under-approximates C. Therefore, if E is a solution for P_C then $E[N/\check{f}_C(\bar{x}_0)]$ is a LB for $C(\bar{x}_0)$.