Precise Cost Analysis via Local Reasoning

Diego E. Alonso-Blas, Puri Arenas, and Samir Genaim

DSIC, Complutense University of Madrid (UCM), Spain

Abstract. The classical approach to static cost analysis is based on first transforming a given program into a set of cost relations, and then solving them into closed-form upper-bounds. The quality of the upper-bounds and the scalability of such cost analysis highly depend on the precision and efficiency of the solving phase. Several techniques for solving cost relations exist, some are efficient but not precise enough, and some are very precise but do not scale to large cost relations. In this paper we explore the gap between these techniques, seeking for ones that are both precise and efficient. In particular, we propose a novel technique that first splits the cost relation into several *atomic* ones, and then uses precise local reasoning for some and less precise but efficient reasoning for others. For the precise local reasoning, we propose several methods that define the cost as a solution of a universally quantified formula. Preliminary experiments demonstrate the effectiveness of our approach.

1 Introduction

Static Cost analysis (a.k.a. resource usage analysis) aims at *statically* determining the amount of resources (e.g., memory, execution steps, etc.) required to execute a given program safely, i.e., without running out of resources. Applications of cost analysis range from detecting performance bottlenecks at the development stage, to providing resource consumption guarantees at runtime.

Several cost analysis frameworks exist [4,13,15,10]. Although different in their underlying techniques, they all report the cost as a closed-form upper-bound function (*UB* for short) in terms of the input parameters. This paper uses the classical approach of Wegbreit [18], in particular its extension for JAVA bytecode [4], where the analysis is carried out in two phases: (1) the input program is transformed into a set of *cost relations* (*CRs* for short) that define its cost; and (2) the *CRs* are solved into *UBs*. While the first phase depends on the programming language in which the program is written [4,11,12,9,17], the second phase is common to all analyses that are based on this approach. In this paper *we focus on the second phase*, i.e., on developing techniques for solving *CRs*. However, we provide enough details to clarify how *CRs* are related to programs.

Example 1. The JAVA class depicted in Fig. 1 implements a dynamic array, where field `data` is used to store its elements, and field `size` represents the number of such elements. Method `add` adds the elements of the array `elems` to the dynamic array. When the array `data` is full (L6), it is replaced by a new one of double

```
1 class DynamicArray {
2   int[ ] data;
3   int size;
4   void add(int[ ] elems) {
5     for (int i=0; i<elems.length; i++) {
6       if (data.length == size) {
7         int[ ] tmp = new int[2*data.length];
8         copy(tmp,size,data);
9         data = tmp;
10      }
11      data[size] = elems[i];
12      size++;
13    }
14  }
15  int r;
16  void qsort() {
17    qs(0, size-1);
18  }
19  void qs(int from, int to) {
20    if (to - from < r)
21      insertionSort(from,to);
22    else {
23      int m=partition(from,to);
24      qs(from,m-1);
25      qs(m+1,to);
26    }
27  }
28 }
```

$add(e,d,s) = for(e,d,s,i)$	$\varphi_0 = \{e \geq 0, d \geq 0, s \geq 0, i = 0\}$
$for(e,d,s,i) = 0$	$\varphi_1 = \{i \geq e\}$
$for(e,d,s,i) = 2 \cdot \mathsf{nat}(s) + 2 + for(e,d',s',i')$	$\varphi_2 = \{i < e, s = d, d' = 2 \cdot d, s' = s+1, i' = i+1\}$
$for(e,d,s,i) = 2 + for(e,d,s',i')$	$\varphi_3 = \{i < e, d > s, s' = s+1, i' = i+1\}$

$qsort(s,r) = qs(f,t,r)$	$\psi_0 = \{s \geq 0, f = 0, t = s-1\}$
$qs(f,t,r) = \mathsf{nat}(t-f)^2$	$\psi_1 = \{t - f < r, r \geq 0\}$
$qs(f,t,r) = \mathsf{nat}(t-f) + qs(f,m',r) + qs(m'',t,r)$	$\psi_2 = \{t - f \geq r, r \geq 0, f \leq m \leq t, m' = m-1, m'' = m+1\}$

Fig. 1. Above, JAVA code of a DynamicArray class. Below, the *CRs* of the methods

size (L7-9). Methods qsort and qs sort the array using a variation of *Quick Sort*, which resorts to *Insertion Sort* when the segment to be sorted is shorter than a threshold defined by field r. Methods copy, partition, and insertionSort are omitted.

Below the JAVA code we show the corresponding *CRs*, generated using a cost model that counts array accesses. Let us explain the *CR* of method add. Variables e, d, s, and i stand for the lengths of arrays elems and data and the values of size and i. Expression $\mathsf{nat}(e)$ is an abbreviation for $\max\{e, 0\}$. The first equation states that the cost of $add(e,d,s)$ is as that of $for(e,d,s,i)$. The constraints on the right impose conditions and relations on the variables. The second equation is for the case of exiting the loop ($i \geq e$). The third one is for the case in which the array is resized. In such case the cost is $2 \cdot \mathsf{nat}(s)$ (the cost assumed for copy), plus 2 (the accesses at L11), plus the cost of the remaining iterations $for(e,d',s',i')$. Note that $d' = 2 \cdot d$ states that the size of array data is doubled. The fourth equation describes the case in which the array is not resized. The equations of qsort are defined similarly. We note that $\mathsf{nat}(t-f)^2$ and $\mathsf{nat}(t-f)$ correspond to the cost of insertionSort and partition respectively. The constraint $f \leq m \leq t$ in ψ_2 is an input-output summary inferred for the value m returned by method partition. Methods add and qsort, respectively, have linear and quadratic worst-case complexity. □

Early works on cost analysis [11,9] relied on Computer Algebra Systems (CAS) for solving *CRs*. They can only handle cases in which the *CRs* can be

transformed into *recurrence equations* (the only valid input for CAS). This, however, is a very limited subset because *CRs* allow using constraints to define complex applicability conditions and relations between the variables. To overcome this limitation, recent works [3,6] have developed dedicated tools for solving *CRs* into *UBs*. They are mostly based on the use of program analysis techniques. These works are our starting point.

The techniques of [3] are based on assuming worst-case behaviour for all loop iterations. It is very efficient and can handle a wide class of *CRs*. To solve the *CR for*, this technique infers that $2{+}2{\cdot}\mathsf{nat}(e{+}s{-}1)$ is an *UB* on the cost of any iteration of *for*, and it infers that there is at most $\mathsf{nat}(e{-}i)$ iterations of *for*, from which it concludes that $\mathsf{nat}(e{-}i){\cdot}(2{+}2{\cdot}\mathsf{nat}(e{+}s{-}1))$ is an *UB* for *CR for*. Note that this is a quadratic *UB* while the actual cost is linear. In the case of qsort, the loss of precision is even bigger. It first infers that $\mathsf{nat}(t{-}f)^2$ is an *UB* on the cost of each call to qs, and that there are at most $2^{\mathsf{nat}(t-f)}$ of such calls. Then, it concludes that $(t{-}f)^2{\cdot}2^{\mathsf{nat}(t-f)}$ is an *UB* for *CR qs*, while the actual cost is quadratic.

The above imprecision issue, among others, was addressed in [6] where precise and novel techniques for solving *CRs* were proposed. They are based on defining the cost as a solution of a corresponding first-order universally quantified formula. This method, as expected, would obtain the most precise *UBs* for the *CRs for* and *qs*, however, it has two major limitations: (1) a template *UB* has to be provided by the user; and (2) the use of a quantifier elimination procedure for real numbers renders the technique impractical.

In this paper we explore the gap between [3] and [6], seeking for solving techniques with efficiency close to [3] and precision close to [6]. Concretely, we develop a novel technique that breaks down the input *CR* into *atomic CRs* of simpler form, solves each of them separately, and then combines the results into an *UB* for the original *CR*. Our main observation is that it is enough to solve few atomic *CRs* precisely, while solving the others as in [3], without affecting the overall precision. We also propose several methods for precisely solving atomic *CRs*, which are based on the idea of specifying the cost using universally quantified formulas as in [6]. However, we do not require the user to provide any template, and, importantly, the generated formulas have almost a linear form for which quantifier elimination can be done efficiently. Our prototype implementation and experiments [1] demonstrate the effectiveness of this approach.

This paper is organised as follows. Sec. 2 provides the required background on *CRs*. Sec. 3 is the technical core of the paper. Sec. 4 describes a prototype implementation and preliminary experiments. Finally, in Sec. 5 we conclude and discuss related work.

2 Cost Relations: Syntax and Semantics

In this section we recall some basic notions related to *CRs* [3]. The sets of real, rational, and integer values are denoted by $\mathbb{R}$, $\mathbb{Q}$, and $\mathbb{Z}$, respectively. $\mathbb{R}^+$, $\mathbb{Q}^+$, and $\mathbb{Z}^+$ denote their non-negative subsets. Variables are denoted by x, y, z, and

w, possibly subscripted. Values from $\mathbb{R}$, $\mathbb{Q}$, and $\mathbb{Z}$ are denoted, respectively, by r, q, and v. A sequence of elements of type t is denoted by $\bar{t}$. The set of variables of t is denoted by $vars(t)$. An assignment $\sigma : \mathcal{V} \mapsto \mathcal{D}$ maps variables from $\mathcal{V}$ to values from $\mathcal{D}$ and $\sigma(\bar{t})$ denotes the replacement of any $x \in vars(\bar{t})$ by $\sigma(x)$.

A *linear expression* has the form $q_0{+}q_1{\cdot}x_1{+}\cdots{+}q_n{\cdot}x_n$. A *linear constraint* has the form $l_1{\leq}l_2$, $l_1{=}l_2$, or $l_1{\geq}l_2$, where l_1 and l_2 are linear expressions and $vars(l_1){\cup}vars(l_2) \subseteq \mathbb{Z}$. The constraints $l_1{>}l_2$ and $l_1{<}l_2$ abbreviate $l_1{\geq}l_2{+}1$ and $l_1{+}1{\leq}l_2$, respectively. We use φ, ϕ, and ψ, possibly subscripted, to denote conjunctions (often written as sets) of linear constraints. We say that φ is *satisfiable* if there is an assignment σ for $vars(\varphi)$ such that $\sigma(\varphi)$ is true, denoted as $\sigma \models \varphi$. If $\sigma \models \varphi$ for every assignment σ for $vars(\varphi)$ then φ is a *valid* formula.

Definition 1 (cost expression). *A* cost expression e *is defined as:*

$$e ::= q \mid \mathsf{nat}(l) \mid \log_a(1{+}\mathsf{nat}(l)) \mid a^{\mathsf{nat}(l)}{-}1 \mid e{+}e \mid e{\cdot}e$$

where $q \in \mathbb{Q}^+$, $\mathsf{nat}(l) = \max\{l, 0\}$, $a > 1 \in \mathbb{Z}^+$, *and* l *is a linear expression.*

Note that we use $a^{\mathsf{nat}(l)}{-}1$, instead of simply $a^{\mathsf{nat}(l)}$, for the sake of simplifying the formal presentation (we explain this after Lemma 3).

Definition 2 (cost relation). *A* cost relation *is a set of* cost equations *of the form* $\langle C(\bar{x}) = e + \sum_{j=1}^{k} D_j(\bar{y}_j), \varphi\rangle$, *where* C *and* D_j *are cost relation symbols.*

Intuitively, a cost equation $\langle C(\bar{x}) = e + \sum_{j=1}^{k} D_j(\bar{y}_j), \varphi\rangle$ states that the cost of $C(\bar{x})$ is e plus the sum of the costs of $D_1(\bar{y}_1), \ldots, D_k(\bar{y}_k)$. The linear constraint φ specifies the values of $\bar{x}$ for which the equation is applicable, and defines relations among the different variables. Since *CRs* usually originate from programs, it is often helpful to think of each *CR* symbol as a (non-deterministic) procedure, in which case we say that C calls $D_1, \ldots, D_k$.

Without loss of generality, in what follows we assume that the input *CR* includes a single *CR* symbol. Namely, in Def. 2 we have $D_j{=}C$. We call such *CRs stand-alone*. To handle *CRs* with more than one *CR* symbol, we rely on the compositional approach of [3] which we briefly explain next. In a first step, the input *CR* is transformed into a form in which all recursions are direct, i.e. an equation that defines C can either call itself directly, or other *CR* symbols that do not call C (directly or indirectly). In a second step, the *CRs* are solved iteratively, where in each iteration we solve those that do not depend on any other symbols (there must be at least one), and then substitute the result in the calling contexts. In the rest of the paper *CR* refers to a stand-alone *CR*.

To define the cost assigned by C to a concrete input $\bar{v}$, we use evaluation trees. A (possibly infinite) tree will be denoted by $node(r, \langle T_1, \ldots, T_k\rangle)$, where $r \in \mathbb{R}^+$ is the value of the root and $T_1, \ldots, T_k$ are sub-trees.

Definition 3 (evaluation tree). *Given a* CR C *and an input* $\bar{v}$, *we say that* $node(r, \langle T_1, \ldots, T_k\rangle)$ *is an* evaluation tree *for* $C(\bar{v})$ *iff there exists an equation* $\mathcal{E} \equiv \langle C(\bar{x}) = e{+}\sum_{j=1}^{k} C(\bar{y}_j), \varphi\rangle$ *and* $\sigma : vars(\mathcal{E}) \mapsto \mathbb{Z}$ *such that: (1)* $\sigma(x_i){=}v_i$ *and* $\sigma{\models}\varphi$; *(2)* $r = \sigma(e)$; *and (3) each* T_i *is an evaluation tree for* $C(\sigma(\bar{y}_i))$.

Intuitively, when viewing C as a procedure, an evaluation tree can be seen as a *recursion tree* where the call $C(\bar{v})$ is evaluated as follows: we pick an equation that defines C and an assignment σ that satisfies the equation's constraints; we evaluate $\sigma(e)$ into r, and we recursively call each $C(\sigma(\bar{y}_i))$. Note that an evaluation tree can be infinite. Note also that $C(\bar{v})$ might have several evaluation trees, due to the nondeterminism induced by choosing an equation for C and a satisfying assignment σ for φ. The set of all evaluation trees for $C(\bar{v})$ is denoted by $\mathit{Trees}(C(\bar{v}))$. The set of all possible costs for $C(\bar{v})$ is then defined as $\mathit{Answers}(C(\bar{v}))=\{\mathsf{Sum}(T) \mid T\in\mathit{Trees}(C(\bar{v}))\}$, where $\mathsf{Sum}(T)$ is the sum of all nodes of T. Our interest is to approximate *CRs* by mean of closed-form *UBs* functions, i.e., functions of the form $f(\bar{x})=e$, where $\mathit{vars}(e) \subseteq \bar{x}$.

Definition 4 (upper bound). *A function $C^+ : \mathbb{Z}^n \mapsto \mathbb{R}^+$ is an* UB *for a* CR *C, iff for any input $\bar{v} \in \mathbb{Z}^n$ and cost $r \in \mathit{Answers}(C(\bar{v}))$ we have $C^+(\bar{v}) \geq r$.*

Next we overview the approach of [3] for solving a *CR* into an *UB*. Suppose we have two functions $h(\bar{x})=e_1$ and $g(\bar{x})=e_2$, where e_1 and e_2 are cost expressions, such that for any $T\in\mathit{Trees}(C(\bar{v}))$ the following holds (i) $h(\bar{v})$ is an *UB* on the depth of T; and (ii) $g(\bar{v})$ is an *UB* on the value of any node of T. Now assuming that d is the maximum number of recursive calls in any equation of C, i.e., the maximum branching factor of its evaluation trees, then $C^+(\bar{x})=g(\bar{x})\cdot\mathcal{N}$ where $\mathcal{N}=h(\bar{x})$ if $d=1$, and $\mathcal{N}=d^{h(\bar{x})}$ if $d>1$. Technically, in [3], $h(\bar{x})$ is computed by inferring a linear *ranking function* [8] that bounds the recursion depth of C, and $g(\bar{x})$ is computed by relying on *linear invariants*.

Example 2. Consider the *CR for* in Fig. 1. The technique of [3] infers $h(e,d,s,i) = \mathsf{nat}(e-i)$ and $g(e,d,s,i) = 2+2\cdot\mathsf{nat}(e+s-1)$. Then, since the branching factor is $d=1$, it reports the *UB* $\mathit{for}^+(e,d,s,i)=\mathsf{nat}(e-i)\cdot(2+2\cdot\mathsf{nat}(e+s-1))$. For *CR qs*, it infers $h(f,t,r)=\mathsf{nat}(t-f)$ and $g(f,t,r)=\mathsf{nat}(t-f)^2$. Then, since the branching factor is $d=2$, it reports the *UB* $qs^+(f,t,r)=\mathsf{nat}(t-f)^2\cdot 2^{\mathsf{nat}(t-f)}$. □

Maximisation procedure. We rely on the technique of [3] that generates $g(\bar{x})$ as we explain next. Let e be the cost expression that is contributed by an equation of C, and let b be a cost sub-expression of e. As explained in Def. 3, when generating the nodes of an evaluation tree $T\in\mathit{Trees}(C(\bar{v}))$, we evaluate $\sigma(e)$ to r. This evaluation requires computing $\sigma(b)$. We call $\sigma(b)$ an instance of b. We reuse the techniques of [3] to infer a cost expression $\hat{b}(\bar{x})$ that satisfies the following: for any input $\bar{v}$, $T\in\mathit{Trees}(C(\bar{v}))$ and any instance $\sigma(b)$ of b in T, we have $\hat{b}(\bar{v}) \geq \sigma(b)$. Intuitively, $\hat{b}(\bar{x})$ is a function that bounds each contribution of b to the total cost. We call $\hat{b}(\bar{x})$ the *maximisation* of b, and, in our implementation, we compute it reusing the components of [3].

3 Solving Cost Relations in Closed-Form Upper-Bounds

In this section we present our approach for solving a *CR* C into an *UB*. We assume that C is defined by m equations of the form $\langle C(\bar{x}) = e_i + \sum_{j=1}^{k_i} C(\bar{y}_{ij}), \varphi_i\rangle$,

$$
\begin{array}{|ll|l|}
\hline
for_1 & for(e,d,s,i) = 0 & \varphi_1 = \{i \geq e\} \\
 & for(e,d,s,i) = 2 \cdot \mathsf{nat}(s) + for(e,d',s',i') & \varphi_2 = \{i < e, s = d, d' = 2 \cdot d, s' = s+1, i' = i+1\} \\
 & for(e,d,s,i) = for(e,d,s',i') & \varphi_3 = \{i < e, d > s, s' = s+1, i' = i+1\} \\
\hline
for_2 & for(e,d,s,i) = 0 & \varphi_1 = \{i \geq e\} \\
 & for(e,d,s,i) = 2 + for(e,d',s',i') & \varphi_2 = \{i < e, s = d, d' = 2 \cdot d, s' = s+1, i' = i+1\} \\
 & for(e,d,s,i) = for(e,d,s',i') & \varphi_3 = \{i < e, d > s, s' = s+1, i' = i+1\} \\
\hline
for_3 & for(e,d,s,i) = 0 & \varphi_1 = \{i \geq e\} \\
 & for(e,d,s,i) = for(e,d',s',i') & \varphi_2 = \{i < e, s = d, d' = 2 \cdot d, s' = s+1, i' = i+1\} \\
 & for(e,d,s,i) = 2 + for(e,d,s',i') & \varphi_3 = \{i < e, d > s, s' = s+1, i' = i+1\} \\
\hline
qs_1 & qs(f,t,r) = \mathsf{nat}(t-f)^2 & \psi_1 = \{t - f < r, r \geq 0\} \\
 & qs(f,t,r) = qs(f,m',r) + qs(m'',t,r) & \psi_2 = \{t - f \geq r, r \geq 0, f \leq m \leq t, \\
 & & \quad m' = m-1, m'' = m+1\} \\
\hline
qs_2 & qs(f,t,r) = 0 & \psi_1 = \{t - f < r, r \geq 0\} \\
 & qs(f,t,r) = \mathsf{nat}(t-f) + qs(f,m',r) + & \psi_2 = \{t - f \geq r, r \geq 0, f \leq m \leq t, \\
 & \quad qs(m'',t,r) & \quad m' = m-1, m'' = m+1\} \\
\hline
\end{array}
$$

Fig. 2. The sparse *CRs* of *for* and *qs* of Fig. 1.

$1 \leq i \leq m$. Our approach is presented in two steps: we reduce the problem of solving C to solving *atomic CRs*, and then we focus on solving atomic *CRs*.

Observe that cost expressions, as in Def. 1, can be normalised into the form $P_1 + \cdots + P_h$, where each P_i is a *product* of cost expressions $b_{i1}, \ldots, b_{ip_i}$ with $b_{ij} \in \{q, \mathsf{nat}(l), \log_a(1+\mathsf{nat}(l)), a^{\mathsf{nat}(l)}-1\}$. For simplicity, since q is non-negative, we assume it is given as $\mathsf{nat}(q)$. We assume that each e_i in C is given in this form. Let $P_C = \{P_1, \ldots, P_t\}$ be the multiset of all non-zero product cost expressions that appear in C (i.e., the products of $e_1, \ldots, e_m$). We define C_i as the *CR* obtained from C by removing all $P_j \in P_C$ with $j \neq i$. Namely, in C_i there is exactly one equation that contributes P_i, the others contribute 0. We call such *CRs sparse* and the equation that includes P_i is called the *main equation*.

Example 3. Consider the *CRs for* and *qs* in Fig. 1. Their products are respectively $P_{for} = \{2 \cdot \mathsf{nat}(s), 2, 2\}$ and $P_{qs} = \{\mathsf{nat}(t-f) \cdot \mathsf{nat}(t-f), \mathsf{nat}(t-f)\}$. Their corresponding sparse *CRs* are depicted in Fig. 2. □

Observation 1 *If $C_i^+(\bar{x})$ is an* UB *for the sparse* CR *C_i, for all $1 \leq i \leq t$, then $C^+(\bar{x}) = C_1^+(\bar{x}) + \cdots + C_t^+(\bar{x})$ is an* UB *for C.*

The above observation explains how an *UB* for C can be obtained from *UBs* for its sparse *CRs* $C_1, \ldots, C_t$. Thus, we can focus on solving sparse *CRs*. We first explain the idea intuitively. Assume that $b_{i1} \cdot b_{i2}$ is the product in the main equation of C_i. Given an arbitrary $T \in Trees(C_i(\bar{v}))$, the cost of each of its nodes is either 0 or an instance of $b_{i1} \cdot b_{i2}$. Let $\sigma_1(b_{i1} \cdot b_{i2}), \ldots, \sigma_h(b_{i1} \cdot b_{i2})$ be the instances of $b_{i1} \cdot b_{i2}$ in $T \in Trees(C_i(\bar{v}))$, then the cost of T is $S = \sum_{j=1}^{h} \sigma_j(b_{i1} \cdot b_{i2})$. As explained in Sec. 2, we can compute a function $\hat{b}_{i1}(\bar{x})$ such that $\hat{b}_{i1}(\bar{v}) \geq \sigma_j(b_{i1})$ for each $1 \leq j \leq h$. Using $\hat{b}_{i1}(\bar{v})$ we bound S as follows:

$$S = \sum_{j=1}^{h} \sigma_j(b_{i1} \cdot b_{i2}) \leq \sum_{j=1}^{h} \hat{b}_{i1}(\bar{v}) \cdot \sigma_j(b_{i2}) = \hat{b}_{i1}(\bar{v}) \cdot \sum_{j=1}^{h} \sigma_j(b_{i2})$$

Now assume that we have a function $f^+(\bar{x})$ such that $f^+(\bar{v}) \geq \sum_{j=1}^{h} \sigma_j(b_{i2})$, then $S \leq \hat{b}_{i1}(\bar{v}) \cdot f^+(\bar{v})$. Thus, since the above reasoning is done for an arbitrary T, we can conclude that $\hat{b}_{i1}(\bar{x}) \cdot f^+(\bar{x})$ is an *UB* for C_i. Now to compute $f^+(\bar{x})$, we consider a *CR* C_{i2} that is obtained from C_i by replacing $b_{i1} \cdot b_{i2}$ by b_{i2}. Clearly, $C_{i2}(\bar{v}) = \sum_{j=1}^{h} \sigma_j(b_{i2})$, and thus any *UB* for C_{i2} defines a valid $f^+(\bar{x})$. This reduces the problem of solving C_i to that of solving C_{i2}, which is simpler since its main equation includes a basic cost expression. Note that, in a similar way, we could build C_{i1} using $\hat{b}_{i2}(\bar{x})$ and then use it to find an *UB* for C_i.

Formally, given a sparse *CR* C_i with a product $b_{i1} \cdot \cdots \cdot b_{ip_i}$ in its main equation, we define the *atomic CR* C_{ij} as the one obtained from C_i by replacing its product by b_{ij} (i.e., removing all b_{ik} with $k \neq j$).

Example 4. Consider the sparse *CRs* depicted in Fig. 2. The following are possible atomic *CRs* for for_1 and qs_1

for_{12}	
$for(e,d,s,i)=0$	φ_1
$for(e,d,s,i)=\mathsf{nat}(s)+for(e,d',s',i')$	φ_2
$for(e,d,s,i)=for(e,d,s',i')$	φ_3

qs_{11}	
$qs(f,t,r)=\mathsf{nat}(t-f)$	ψ_1
$qs(f,t,r)=qs(f,m',r)+qs(m'',t,r)$	ψ_2

in which $\mathsf{nat}(s)$ and $\mathsf{nat}(t-f)$ are selected as basic cost expressions. *CRs* for_2, for_3 and qs_2 are already atomic. They correspond to for_{21}, for_{31} and qs_{21}. □

Lemma 1. *Let C_i be a* sparse CR*, $b_{i1} \cdot \cdots \cdot b_{ip_i}$ the product in its main equation, and C_{ij} an atomic* CR *of C_i. If $C^+_{ij}(\bar{x})$ is an* UB *for $C_{ij}(\bar{x})$, then $C^+_i(\bar{x}) = C^+_{ij}(\bar{x}) \cdot \prod_{k \neq j} \hat{b}_{ik}(\bar{x})$ is an* UB *for C_i.*

The above lemma allows focusing on finding an *UB* for a single atomic C_{ij} and then combine the result into an *UB* for C_i. To put this into practice we need to address the following issues: (1) how to select the basic cost expression j from the products in order to build C_{ij}; and (2) how to compute an *UB* for C_{ij}. In secs. 3.1 and 3.2 we discuss several methods for addressing the second issue. The first issue is discussed later in Sec. 3.4.

Let us first position our approach in the spectrum of related approaches [3,6]. Solving C_i using the techniques of [3] we obtain the *UB* $(\prod_{k=1}^{p_i} \hat{b}_{ik}(\bar{x})) \cdot \mathcal{N}$. Interestingly, this *UB* can be explained using our novel view of Lemma 1, which is different from that of [3], as follows: we can consider $\hat{b}_{ij}(\bar{x}) \cdot \mathcal{N}$ as an *UB* for C_{ij}, and then use it as in Lemma 1 to obtain $(\prod_{k=1}^{p_i} \hat{b}_{ik}(\bar{x})) \cdot \mathcal{N}$. Since, unlike [3], we focus on solving atomic *CRs*, we develop dedicated techniques (i.e., techniques that work only for atomic *CRs*) that are able to obtain an *UB* far more precise than $\hat{b}_{ij}(\bar{x}) \cdot \mathcal{N}$ (we will usually eliminate the $\mathcal{N}$ factor). Solving C_i using the techniques of [6] requires defining an *UB* template to be used during the solving process. If C_i does not admit an *UB* that matches the supplied templates, then this technique will fail. Moreover, using arbitrary templates renders this approach impractical since it is based on the use of quantifier elimination procedure. Our techniques for solving atomic *CRs* are actually inspired by those

of [6]. However, since we focus on a simpler form of *CRs*, we always use linear templates for which the quantifier elimination procedure is efficient. In summary, our approach uses [6] to precisely reason on the *local* cost of a single simple cost expression b_{ij}, and then uses [3] to combine this local cost into an *UB* for C_i.

To simplify our notation, in what follows, we assume a given atomic *CR* D with m equations of the form $\langle D(\bar{x}) = e_i + \sum_{j=1}^{k_i} D(\bar{y}_{ij}), \varphi_i \rangle$, where $e_1 = b$ is a basic cost expression, and $e_i = 0$ for all $2 \leq i \leq m$. Note that the main equation of D is the first one. We denote by $\bar{w}_i$ the set of variables in the i-th equation.

3.1 The Tree-Sum Method

We first explain this method for the case in which $b = \mathsf{nat}(l)$, and then we show how to extend it to handle any basic cost expression b. In many cases, in particular in examples that require amortised analysis, the sum of all instances of b in any $T \in \mathit{Trees}(D(\bar{v}))$ can be bounded by a linear expression. Thus, we seek an *UB* for D of the form $\alpha(\bar{x}) = q_0 + q_1 \cdot x_1 + \cdots + q_n \cdot x_n$, where $q_i \in \mathbb{Q}$. The way we search for $\alpha(\bar{x})$ is based on the use of universally quantified formulas as in [6]. We first define a verification condition which ensures that a given $\alpha(\bar{x})$ is a valid *UB* for D. Then, using a quantifier elimination procedure, we turn this verification condition into a synthesis procedure that actually infers $\alpha(\bar{x})$.

Lemma 2. *Let* $\alpha(\bar{x}) = q_0 + q_1 \cdot x_1 + \cdots + q_n \cdot x_n$*, and define:*

$$\Psi_1 \triangleq \forall \bar{w}_1 \ : \ \varphi_1 \rightarrow \mathsf{nat}(\alpha(\bar{x})) \geq \mathsf{nat}(l) + \textstyle\sum_{j=1}^{k_1} \mathsf{nat}(\alpha(\bar{y}_{1j}))$$
$$\Psi_2 \triangleq \textstyle\bigwedge_{i=2}^{m} \forall \bar{w}_i \ : \ \varphi_i \rightarrow \mathsf{nat}(\alpha(\bar{x})) \geq \sum_{j=1}^{k_i} \mathsf{nat}(\alpha(\bar{y}_{ij}))$$

If $\Psi_1 \wedge \Psi_2$ *is valid, then* $\mathsf{nat}(\alpha(\bar{x}))$ *is an* UB *for the atomic* CR D.

Intuitively, Ψ_1 requires that $\mathsf{nat}(\alpha(\bar{x}))$ covers the cost of the main equation, i.e., it covers the local cost $\mathsf{nat}(l)$ and the cost of the recursive calls. Similarly, Ψ_2 requires that $\mathsf{nat}(\alpha(\bar{x}))$ covers the cost of the other equations (in this case the local cost is 0). Our main interest is in inferring such $\alpha(\bar{x})$ rather than verifying the correctness of a given one. Turning the verification condition into an inference procedure can be done, using a quantifier elimination procedure, as follows:

1. we generate $\Psi_1 \wedge \Psi_2$ using a *template* function $\alpha(\bar{x})$ in which $q_0, \ldots, q_n$ are variables, i.e., *unknown*;
2. we eliminate the universal quantifiers from $\Psi_1 \wedge \Psi_2$. This results in a set of constraints Θ over the variables $q_0, \ldots, q_n$; and
3. any solution of Θ (i.e., values for $q_0, \ldots, q_n$ that satisfy Θ) defines a valid *UB* $\mathsf{nat}(\alpha(\bar{x}))$. We simply pick a solution.

Note that if Θ is not satisfiable then there is no $\alpha(\bar{x})$ satisfying $\Psi_1 \wedge \Psi_2$. In such case we say that the Tree-Sum method is not applicable for D. The main subtle point in the above inference procedure is how to eliminate the universal quantifiers, which is computationally expensive in general. However, since the formula $\Psi_1 \wedge \Psi_2$ have a very specific form (almost linear), in Sec. 3.3 we show how this can be done efficiently. For now we just assume the existence of a procedure that implements steps (2) and (3) above.

Example 5. Consider the *CR* for_{12}, as defined in Ex. 4, and let $\alpha(e,d,s,i) = q_0 + q_1{\cdot}e + q_2{\cdot}d + q_3{\cdot}s + q_4{\cdot}i$. The corresponding Ψ_1 and Ψ_2 are:

$$\Psi_1 \triangleq \forall \bar{w}_2 : \varphi_2 \rightarrow \mathsf{nat}(q_0+q_1{\cdot}e+q_2{\cdot}d+q_3{\cdot}s+q_4{\cdot}i) \geq \mathsf{nat}(s)+\mathsf{nat}(q_0+q_1{\cdot}e+q_2{\cdot}d'+q_3{\cdot}s'+q_4{\cdot}i')$$
$$\Psi_2 \triangleq \forall \bar{w}_3 : \varphi_3 \rightarrow \mathsf{nat}(q_0+q_1{\cdot}e+q_2{\cdot}d+q_3{\cdot}s+q_4{\cdot}i) \geq \mathsf{nat}(q_0+q_1{\cdot}e+q_2{\cdot}d+q_3{\cdot}s'+q_4{\cdot}i')$$

Solving $\Psi_1 \wedge \Psi_2$, i.e finding values for $q_0, \ldots, q_4$, gets $q_0{=}{-}2$, $q_1{=}2$, $q_2{=}{-}1$, $q_3{=}2$, and $q_4{=}{-}2$, which means that $for^+_{12}(e,d,s,i){=}\mathsf{nat}(2{\cdot}s{+}2{\cdot}e{-}2{\cdot}i{-}d{-}2)$ is an *UB* for *CR* for_{12}. Then, to get an *UB* for *CR* for_1 we apply Lemma 1 which results in $for^+_1(e,d,s,i){=}2{\cdot}\mathsf{nat}(2{\cdot}s{+}2{\cdot}e{-}2{\cdot}i{-}d{-}2)$. Similarly, generating the formulas for for_{21} and for_{31} and solving them, we get the *UBs* $for^+_2(e,d,s,i){=}\mathsf{nat}(2{\cdot}e{-}2{\cdot}i)$ and $for^+_3(e,d,s,i){=}\mathsf{nat}(2{\cdot}e{-}2{\cdot}i)$. Finally, we can use Obs. 1 to add them in $for^+(e,d,s,i){=}2{\cdot}\mathsf{nat}(2{\cdot}s{+}2{\cdot}e{-}2{\cdot}i{-}d{-}2){+}2{\cdot}\mathsf{nat}(2{\cdot}e{-}2{\cdot}i)$ as *UB* for *for*. Substituting this *UB* in the equation of *add* in Fig. 1, we get the expected linear bound $add^+(e,d,s) = 2{\cdot}\mathsf{nat}(2{\cdot}s{+}2{\cdot}e{-}d{-}2){+}2{\cdot}\mathsf{nat}(2{\cdot}e)$ for method `add`. □

Now we turn to the general case in which b is an arbitrary basic cost expression, not necessarily $\mathsf{nat}(l)$. In such cases, in addition to $\mathsf{nat}(l)$, b can be of the form $\log_a(1+\mathsf{nat}(l))$ or $a^{\mathsf{nat}(l)}-1$. Recall that when it is $q \in \mathbb{Q}^+$, we have implicitly assumed it was written as $\mathsf{nat}(q)$. Note that in all cases b has an embedded $\mathsf{nat}(l)$ expression. Let E be the *CR* obtained from D by replacing b by its embedded $\mathsf{nat}(l)$. Then the following lemma explains how to obtain an *UB* for D from that of E. Computing an *UB* for E is done as above.

Lemma 3. *Let* $\mathsf{nat}(\alpha(\bar{x}))$ *be an* UB *for* E*, and let*

$$D^+(\bar{x}) = \begin{cases} \mathsf{nat}(\alpha(\bar{x})) & b = \mathsf{nat}(l) \\ 1.5{\cdot}\mathsf{nat}(\alpha(\bar{x})) & b = \log_a(1+\mathsf{nat}(l)) \\ a^{\mathsf{nat}(\alpha(\bar{x}))}-1 & b = a^{\mathsf{nat}(l)}-1 \end{cases}$$

Then, $D^+(\bar{x})$ *is an* UB *for* D*.*

It is worth mentioning here the reason for which we use $a^{\mathsf{nat}(l)}-1$ as a basic cost expression, instead of $a^{\mathsf{nat}(l)}$. This allows *precisely* lifting the *UB* of E to an *UB* of D (in the last case of D^+), which is not possible when using $a^{\mathsf{nat}(l)}$.

Example 6. Let us finish this section by trying to analyse the *CR* *qs* using the Tree-Sum method. For qs_{11}, we first generate:

$$\Psi_1 \triangleq \forall \bar{w}_1 \ : \ \psi_1 \rightarrow \mathsf{nat}(q_0+q_1{\cdot}f+q_2{\cdot}t+q_3{\cdot}r) \geq \mathsf{nat}(t{-}f)$$
$$\Psi_2 \triangleq \forall \bar{w}_2 \ : \ \psi_2 \rightarrow \mathsf{nat}(q_0+q_1{\cdot}f+q_2{\cdot}t+q_3{\cdot}r) \geq \mathsf{nat}(q_0+q_1{\cdot}f+q_2{\cdot}m'+q_3{\cdot}r){+}\mathsf{nat}(q_0+q_1{\cdot}m''+q_2{\cdot}t+q_3{\cdot}r)$$

Solving $\Psi_1 \wedge \Psi_2$ results in $q_0 = 0$, $q_1 = -1$, $q_2 = 1$, and $q_3 = 0$. Thus, $\mathsf{nat}(t-f)$ is an *UB* for qs_{11}. Using Lemma 1, we get $qs^+_1(f,t,r){=}\mathsf{nat}(t{-}f)^2$. Solving qs_{21} with the Tree-Sum method does not yield any result because the generated formula is not valid. This is expected since qs_{21} does not have a linear bound. In Sec. 3.2 we develop further methods to handle such cases. □

3.2 The Level-Sum Method

In this section we describe our method for solving atomic *CRs* that exhibit a *divide and conquer* like behaviour. As we have seen in Ex. 6, the Tree-Sum method fails to handle such examples. We first explain it for the case of $b = \mathsf{nat}(l)$, and then extend it to an arbitrary basic cost expression.

We start with some notation. Given an evaluation tree $T \in \mathit{Trees}(D(\bar{v}))$, a node in T is called *primary* if it is generated by the main equation. Note that the cost of all other nodes in T is 0. The *primary-depth* of a primary node is the number of primary nodes on the path from the root to that node (both included). The primary-depth of T, denoted by $\mathit{pdepth}(T)$, is the maximum among the primary depths of all its primary nodes. The sum of (the cost of) all primary nodes of primary-depth i is denoted by $\mathsf{SumLevel}(T, i)$.

We say that $\mathsf{nat}(\alpha(\bar{x}))$ is an UB *on the primary-depth* of D, if for any input $\bar{v}$ and $T \in \mathit{Trees}(D(\bar{v}))$ we have $\mathsf{nat}(\alpha(\bar{v})) \geq \mathit{pdepth}(T)$. We say that it is an UB *on the Level-Sum* of D, if for any input $\bar{v}$, $T \in \mathit{Trees}(D(\bar{v}))$, and $1 \leq i \leq \mathit{pdepth}(T)$ we have $\mathsf{nat}(\alpha(\bar{v})) \geq \mathsf{SumLevel}(T, i)$.

Lemma 4. *Let* $\mathsf{nat}(\alpha_1(\bar{x}))$ *and* $\mathsf{nat}(\alpha_2(\bar{x}))$ *be* UBs *on the primary-depth and Level-Sum of* D*, respectively. Then,* $\mathsf{nat}(\alpha_1(\bar{x}))\cdot\mathsf{nat}(\alpha_2(\bar{x}))$ *is an* UB *for* D*.*

The correctness of the above lemma follows from the fact that only primary nodes can have non-zero cost. Intuitively, the above lemma handles divide and conquer examples since, in such examples, the input is distributed between the recursive calls. Thus, the cost of all levels is similar and can be expressed as a linear function on the initial input. Moreover, using the primary-depth, instead of depth, allows ignoring those levels that do not contribute to the cost. Note that the above lemma also reduces the problem of solving D, to that of finding $\mathsf{nat}(\alpha_1(\bar{x}))$ and $\mathsf{nat}(\alpha_2(\bar{x}))$ that bound its primary-depth and Level-Sum. We start with bounding the primary-depth.

Lemma 5. *Let* $\alpha(\bar{x}) = q_0 + q_1 \cdot x_1 + \cdots + q_n \cdot x_n$*, and define:*

$$\Phi_1 \triangleq \begin{cases} \forall \bar{w}_1 \ : \ \varphi_1 \quad \rightarrow \mathsf{nat}(\alpha(\bar{x})) \geq 1 & \text{if } k_1 = 0 \\ \bigwedge_{j=1}^{k_1} \forall \bar{w}_1 \ : \ \varphi_1 \rightarrow \mathsf{nat}(\alpha(\bar{x})) \geq 1 + \mathsf{nat}(\alpha(\bar{y}_{1j})) & \text{if } k_1 \geq 1 \end{cases}$$

$$\Phi_2 \triangleq \bigwedge_{i=2}^{m} \bigwedge_{j=1}^{k_i} \forall \bar{w}_i \ : \ \varphi_i \rightarrow \mathsf{nat}(\alpha(\bar{x})) \geq \mathsf{nat}(\alpha(\bar{y}_{ij}))$$

If $\Phi_1 \wedge \Phi_2$ *is valid, then* $\mathsf{nat}(\alpha(\bar{x}))$ *is an* UB *on the primary-depth of* D*.*

Intuitively, the primary-depth corresponds to the number of applications of the main equation, in a sequence of recursive calls. This is reflected in Φ_1 and Φ_2 as follows. In Φ_1, we treat applications of the main equation. If the main equation is non-recursive, i.e., $k_1 = 0$, then we require that $\mathsf{nat}(\alpha(\bar{x}))$ covers that single application. In case it is recursive, i.e., $k_1 \geq 1$, then we require that $\mathsf{nat}(\alpha(\bar{x}))$ covers that application and further ones that might arise through each recursive call. In Φ_2, we treat applications of other equations. In such case we require that $\mathsf{nat}(\alpha(\bar{x}))$ covers applications of the main equation that might arise through each recursive call. Note that each recursive call is considered separately, since we count primary nodes in each path rather than the whole tree.

It is worth noting that if we apply Φ_1 to all equations instead of only the main one, then $\mathsf{nat}(\alpha(\bar{x}))$ bounds the depth of any evaluation tree rather than the primary-depth. Similar techniques, based on inference of (linear) ranking functions, were used in [3] to bound the depth of the evaluation trees.

Example 7. Applying Lemma 5 to bound the primary-depth of qs_{21} (of Ex. 4) results in $\Phi_2 = \mathit{true}$ and Φ_1 as the conjunction of the following formulas:

$$\forall \bar{w}_2 \ : \ \psi_2 \rightarrow \mathsf{nat}(q_0{+}q_1{\cdot}f{+}q_2{\cdot}t{+}q_3{\cdot}r) \geq 1{+}\mathsf{nat}(q_0{+}q_1{\cdot}f{+}q_2{\cdot}m'{+}q_3{\cdot}r)$$
$$\forall \bar{w}_2 \ : \ \psi_2 \rightarrow \mathsf{nat}(q_0{+}q_1{\cdot}f{+}q_2{\cdot}t{+}q_3{\cdot}r) \geq 1{+}\mathsf{nat}(q_0{+}q_1{\cdot}m''{+}q_2{\cdot}t{+}q_3{\cdot}r)$$

Both originate from the recursive equation of qs_2. They respectively correspond to the first and second calls. Solving $\Phi_1 \wedge \Phi_2$ results in $q_0 = 1$, $q_1 = -1$, $q_2 = 1$, $q_3 = 0$, which induces the *UB* $\mathsf{nat}(t{-}f{+}1)$ on the primary-depth of qs_{21}. □

Now we turn to bounding the Level-Sum of D.

Lemma 6. *Let* $\alpha(\bar{x}) = q_0{+}q_1{\cdot}x_1{+}\cdots{+}q_n{\cdot}x_n$, *and define:*

$$\Pi_1 \triangleq \forall \bar{w}_1 : \varphi_1 \rightarrow \quad \mathsf{nat}(\alpha(\bar{x})) \geq \mathsf{nat}(l)$$
$$\Pi_2 \triangleq \textstyle\bigwedge_{i=1}^{m} \forall \bar{w}_i : \varphi_i \rightarrow \quad \mathsf{nat}(\alpha(\bar{x})) \geq \sum_{j=1}^{k_i} \mathsf{nat}(\alpha(\bar{y}_{ij}))$$

If $\Pi_1 \wedge \Pi_2$ *is valid, then* $\mathsf{nat}(\alpha(\bar{x}))$ *is an* UB *on the Level-Sum of* D.

Intuitively, Π_1 requires that $\mathsf{nat}(\alpha(\bar{x}))$ covers the local cost of the main equation at any level, and Π_2 requires that it also covers the next level. Combining these conditions, and applying inductive reasoning, one can conclude that $\mathsf{nat}(\alpha(\bar{x}))$ is actually an *UB* on the Level-Sum of D.

Example 8. Consider again qs_{21} (of Ex. 4). Its corresponding formulas are:

$$\Pi_1 \triangleq \forall \bar{w}_2 \ : \ \psi_2 \rightarrow \mathsf{nat}(q_0{+}q_1{\cdot}f{+}q_2{\cdot}t{+}q_3{\cdot}r) \geq \mathsf{nat}(t{-}f)$$
$$\Pi_2 \triangleq \forall \bar{w}_2 \ : \ \psi_2 \rightarrow \mathsf{nat}(q_0{+}q_1{\cdot}f{+}q_2{\cdot}t{+}q_3{\cdot}r) \geq \mathsf{nat}(q_0{+}q_1{\cdot}f{+}q_2{\cdot}m'{+}q_3{\cdot}r){+}\mathsf{nat}(q_0{+}q_1{\cdot}m''{+}q_2{\cdot}t{+}q_3{\cdot}r)$$

Solving $\Pi_1 \wedge \Pi_2$ results in $q_0{=}0$, $q_1{=}-1$, $q_2{=}1$, $q_3{=}0$. This induces the bound $\mathsf{nat}(t{-}f)$ on the Level-Sum. Combining this bound with that in Ex. 7, on the primary depth, we obtain $\mathsf{nat}(t{-}f){\cdot}\mathsf{nat}(t{-}f{+}1)$ as an *UB* for qs_{21}, which is also an *UB* for qs_2. Combining this, using Obs. 1, with the bound of qs_1 computed in Ex. 6, we get $qs^{+}(f,t,r){=}\mathsf{nat}(t{-}f){\cdot}\mathsf{nat}(t{-}f{+}1){+}\mathsf{nat}(t{-}f){\cdot}\mathsf{nat}(t{-}f)$. Substituting this *UB* in the equation of *qsort* in Fig. 1 we obtain $qsort^{+}(s,r){=}\mathsf{nat}(s{-}1){\cdot}\mathsf{nat}(s){+}\mathsf{nat}(s{-}1){\cdot}\mathsf{nat}(s{-}1)$, which is the expected bound for method qsort. □

Turning the verification condition to inference procedure, both in Lemma 5 and Lemma 6, is done as we explained in Sec. 3.1. Handling the general case in which b is an arbitrary basic cost expression, is done exactly as the case of Tree-Sum (see Lemma 3). Note that this affects only the *UB* on the Level-Sum.

Finally, we note that [3] proposed a technique for solving *CRs* with a divide and conquer behaviour, however, it is limited to cases in which: (1) the cost of all levels is non-increasing; and (2) the cost expression of each equation is linear. Note that, *CR* qs_1, for example, does not satisfy both conditions.

3.3 Solving the Universally Quantified Formulas

In this section we describe how we solve the universally quantified formulas of Lemma 2, Lemma 5, and Lemma 6. Namely, starting from a template linear function $\alpha(\bar{x}) = q_0 + q_1 \cdot x_1 + \ldots + q_n \cdot x_n$, we find rational values for $q_0, \ldots, q_n$ for which the corresponding formula is valid. Note that our formulas are conjunctions of universally quantified formulas of the following form:

$$\forall \bar{w} : \ \varphi \rightarrow \mathsf{nat}(l_0) \geq q + \mathsf{nat}(l_1) + \ldots + \mathsf{nat}(l_n) \tag{1}$$

where φ defines a closed polyhedron, $q \in \{0, 1\}$, and each l_i is either a linear function over $\bar{w}$, or a template function $\alpha(\bar{x}) = q_0 + q_1 \cdot x_1 + \cdots + q_n \cdot x_n$ such that $\bar{x} \subseteq \bar{w}$ and $q_i \notin \bar{w}$ (i.e., each q_i is existentially quantified). Our goal is to solve these formulas using linear programming (LP) techniques.

Consider a formula as in (1), but without the nat-expressions, i.e., of the form $\forall \bar{w} : \varphi \rightarrow l_0 \geq q + l_1 + \ldots + l_n$. It is known that there is a complete algorithm, based on the use of LP [8], able to solve such a formula. Our aim is to transform formulas as (1) to a nat-free as above, and then solve them using this algorithm. Recall that $\mathsf{nat}(l_i) = \max\{l_i, 0\}$. This means that $\mathsf{nat}(l_i)$ can be eliminated by explicitly considering the cases for $l_i \geq 0$ and $l_i \leq 0$ (we use $l_i \geq 0$ and not $l_i > 0$ since in LP constraints must be non-strict). For example, eliminating $\mathsf{nat}(l_0)$ can be done by rewriting (1) as:

$$\begin{array}{ll} \forall \bar{w} : \ \varphi \wedge l_0 \geq 0 \rightarrow l_0 \geq q + \mathsf{nat}(l_1) + \ldots + \mathsf{nat}(l_n) & \bigwedge \\ \forall \bar{w} : \ \varphi \wedge l_0 \leq 0 \rightarrow 0 \ \geq q + \mathsf{nat}(l_1) + \ldots + \mathsf{nat}(l_n) & \end{array}$$

This process can be applied iteratively to eliminate each $\mathsf{nat}(l_i)$. There is still one problem that prevents us from directly applying the LP techniques: when l_i is a template function, the constraints $l_0 \geq 0$ and $l_0 \leq 0$ are not linear. To overcome this problem, assuming that eliminating the nat-expression results in a formula ξ, we generated ξ' be the by simply removing all non-linear constraints from ξ. Since all non-linear constraints in ξ appear in the left-hand sides of the implications, we observe that $\xi' \rightarrow \xi$. This means that we can solve ξ', using the LP based algorithm, instead of ξ. Although we scarify completeness, this approach performs well in practice as demonstrated by our experiments.

3.4 Concluding Remarks

Let us conclude this section describing how all pieces, that have been described so far, connects together to infer an *UB* for C.

Solving CR C. This is as done according to the following steps: (1) generating the sparse *CRs* $C_1, \ldots, C_t$ of C; (2) solving each C_i into an *UB* as described below; and (3) combining these *UBs*, as in Obs. 1, into an *UB* for C.

Solving a sparse CR C_i. This step requires solving, using the methods described in secs. 3.1 and 3.2, one C_{ij} of the corresponding atomic *CR* which might fail for some j and succeed for some others. We iterate over all possible j=$1, \ldots, p_i$, and if all fail then we solve C_i using the approach of [3].

Solving an atomic CR C_{ij}. This is done by trying the methods of secs. 3.1 and 3.2, in this order. Note that in [1] we describe some additional methods.

Table 1. Experimental comparison with PUBS [3]. The times on the right (in secs) correspond to analysing a *CR* that connects all benchmarks together (see Sec. 4).

Entry	$\mathcal{O}(ub)$ – **new**	$\mathcal{O}(ub)$ – PUBS	**Eq**	$\mathbf{T}_n$	$\mathbf{T}_p$	**Ov**
add(a,b,c)	$\mathsf{nat}(a)+\mathsf{nat}(2a-b+2c)$	$\mathsf{nat}(a)\cdot\mathsf{nat}(a+c)$	11	0.15	0.11	1.34
qsort(a,b,c)	$\mathsf{nat}(a)^2$	$2^{\mathsf{nat}(a)}\cdot(\mathsf{nat}(a)+\mathsf{nat}(b))$	28	0.61	0.27	2.26
sum(a)	$\mathsf{nat}(a)$	$2^{\mathsf{nat}(a)}\cdot\mathsf{nat}(a)$	36	0.88	0.33	2.63
dac(a,b)	$\mathsf{nat}(a)^2+\mathsf{nat}(a-b)$	$2^{\mathsf{nat}(b)}\cdot\mathsf{nat}(a)$	45	1.24	0.40	3.13
log(a,b)	$\mathsf{nat}(b)+\mathsf{nat}(a)\cdot\mathsf{log}(\mathsf{nat}(b))$	$\mathsf{nat}(b)\cdot\mathsf{nat}(a)$	54	1.71	0.47	3.63
once(a,b)	$\mathsf{nat}(a)+\mathsf{nat}(b)$	$\mathsf{nat}(b)\cdot\mathsf{nat}(a)$	62	1.98	0.57	3.47
twice(a,b)	$\mathsf{nat}(a)+\mathsf{nat}(b)$	$\mathsf{nat}(b)\cdot\mathsf{nat}(a)$	70	2.29	0.69	3.33
full(a,b)	$\mathsf{nat}(a)\cdot\mathsf{nat}(b)$	$\mathsf{nat}(a)\cdot\mathsf{nat}(b)^2$	78	2.74	0.84	3.26
eratos(a)	$\mathsf{nat}(a)$	$\mathsf{nat}(a)^2$	91	3.16	0.94	3.37
peak(a)	$\mathsf{nat}(a)$	$\mathsf{nat}(a)\cdot\mathsf{log}(\mathsf{nat}(a))$	96	3.43	1.01	3.38
stack(a,b,c)	$\mathsf{nat}(b)\cdot\mathsf{nat}(c)+\mathsf{nat}(b)^2$	$\mathsf{nat}(c)\cdot\mathsf{nat}(b)^2$	107	3.95	1.19	3.32
rotate(a,b)	$\mathsf{nat}(a)+\mathsf{nat}(b)+\mathsf{nat}(a-b)$	$\mathsf{nat}(a)\cdot\mathsf{nat}(a-b)$	120	4.84	1.62	2.99
maxsum(a,b)	$\mathsf{nat}(b)\cdot\mathsf{log}(\mathsf{nat}(b))$	$\mathsf{nat}(b)^2$	138	7.67	2.12	3.62
mayor(a)	$\mathsf{nat}(a)\cdot\mathsf{log}(\mathsf{nat}(a))$	$\mathsf{nat}(a)\cdot\mathsf{log}(\mathsf{nat}(a))$	163	13.21	3.20	4.13
msort(a,b,c,d)	$\mathsf{nat}(d-c)\cdot\mathsf{log}(\mathsf{nat}(d-c))$	$\mathsf{nat}(d-c)^2$	173	13.72	3.81	3.60
mergexp(a)	$\mathsf{nat}(a)$	$\mathsf{nat}(a)\cdot\mathsf{log}(\mathsf{nat}(a))$	187	14.65	4.02	3.65
enque(a,b,c,d)	$\mathsf{nat}(c+d)+\mathsf{nat}(a+c)$	$\mathsf{nat}(c+d)\cdot\mathsf{nat}(a+c)$	199	16.34	4.68	3.49
deque(a,b,c)	$\mathsf{nat}(a)+\mathsf{nat}(c)$	$\mathsf{nat}(c)\cdot\mathsf{nat}(a)$	208	17.40	4.91	3.55
infinity(a)	$\mathsf{nat}(a)$	Failed: No RF	219	18.38	5.07	3.63

4 Implementation and Experiments

We have implemented our techniques as an extension of PUBS [3], the solver used in COSTA [4] for solving *CRs* generated from JAVA programs. This allows us to evaluate our approach directly on JAVA programs. We evaluate accuracy and scalability on a set of benchmarks that we collected from related literature, or were written to demonstrate some powerful features of our approach. Although the programs are not large, they exhibit challenging behaviour for cost analysis. The benchmarks and the implementation are available online [1].

In Table 1 we evaluate the accuracy of our approach by comparing it to PUBS [3]. We applied both approaches on each benchmark using a cost model that measure memory consumption or visits to an specific program point (depending on what was more interesting for each benchmark). Each line includes (from left to right) the entry method and its parameters, the *UB* inferred by our approach and the *UB* inferred by PUBS. For readability, bounds are given in asymptotic form [2]. In all examples our approach obtains *UBs* that are asymptotically more accurate than those obtained by PUBS. Moreover, our *UBs* approach obtains precise asymptotic *UBs*, i.e., they exactly reflect the actual cost.

To analyse scalability, we have merged all our benchmarks into a single program as follows: the benchmark in row i was modified to include a call (in one of its loops) to the program at row $i-1$. This means that the i-th benchmark executes at least i nested loops. The runtime (in seconds) of analysing each such (modifed) benchmark is depicted in columns $\mathbf{T}_n$ (current approach) and

$\mathbf{T}_p$ (PUBS) of Table 1. Columns **Eq** and **Ov** are, respectively, the total number of equations and the overhead ($\mathbf{T}_n/\mathbf{T}_p$) introduced by our approach.

We have also compared our approach to [6]. For all benchmarks of Table 1, it did not obtain an *UB* within the one minute time limit. This is expected since it is based on a general procedure for real quantifier elimination.

5 Conclusions and Related Work

In this paper we have developed a novel approach for solving *CRs* into precise closed-form *UBs*. It is based on the idea of dividing the *basic cost expressions* of a given *CR* C into two parts: (a) those for which we employ precise reasoning to track their behaviour along the execution; and (b) those for which we simply use their worst case behavior. Then, we show how such different bounds can be combined into an *UB* for C. For part (b) we rely on existing techniques [4] to *maximise* cost expressions. For part (a) we first model the contribution of the corresponding cost expressions using universally quantified formulas, and then, a precise *UB* on their costs can be obtain by eliminating the universal quantifiers. Note that while quantifier elimination is a very expensive procedure in general, in our case, since the formulas are of a very specific form, they can be solved efficiently. Our method has been implemented within COSTA [4], and preliminary experiments demonstrate its superiority on previous methods for solving *CRs*.

Related work. The most related works to ours are [4,6] which aim at solving *CRs* into closed-form *UBs*. In Sec. 4 we have seen that, in practice, our approach is more precise than [4] and more efficient than [6]. Detailed discussion on similarities and differences is provided along Sec. 3. Note that although the method described so far is usually more precise than [3], as we have seen in Sec. 4, there are some examples for which the use of the last case of Lemma 3 causes a loss of precision. E.g., replacing $\mathsf{nat}(s)$ by $2^{\mathsf{nat}(s)}$ in for_{12} of Ex. 4, the approach of [3] obtains $\mathsf{nat}(e-i)\cdot 2^{\mathsf{nat}(s+e-1)}$ while we obtain $2^{\mathsf{nat}(2(s+e-i-1)-d)}$. In [5], the techniques of [4] were improved to handle cases in which the cost can be modeled with arithmetic or geometric sequences. This approach is complementary to ours, in the sense that it cannot handle our benchmarks and we cannot handle some of their examples (when basic cost expressions require non-linear bounds).

There are some works that aim at inferring loop bounds on the visits to a given program point [14,19]. They are mostly related to our Lemma 5. These approaches are not limited to linear bounds, however, they cannot handle recursive programs with more than one recursive call. Our techniques can benefit from these approaches when each cost equation has at most one recursive call. Cost analysis techniques that are based on *amortised analysis* [15,16], could, in principle, handle some of our examples when the bounds are polynomial, and the data are over the non-negative integers. Solving *CRs* using template functions and real quantifier elimination has been considered before in [7]. Finally, several cost analysis frameworks [9,11] that are based on generating *CRs* can benefit from our advances in solving *CRs*.

Acknowledgements This work was funded partially by the projects FP7-ICT-610582, TIN2008-05624, TIN2012-38137, PRI-AIBDE-2011-0900 and S2009TIC-1465. Diego Esteban Alonso-Blas is supported by the PhD scholarship program of the Complutense University.

References

1. Companion Web-Page. `http://costa.ls.fi.upm.es/amor/`.
2. E. Albert, P. Arenas, D. Alonso, S. Genaim, and G. Puebla. Asymptotic Resource Usage Bounds. In *APLAS*, volume 5904 of *LNCS*, pages 294–310. Springer, 2009.
3. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *J. Autom. Reasoning*, 46(2):161–203, 2011.
4. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. *Theor. Comput. Sci.*, 413(1):142–159, 2012.
5. E. Albert, S. Genaim, and A.N. Masud. On the Inference of Resource Usage Upper and Lower Bounds. *ACM Trans. Comput. Log.* To appear.
6. D. E. Alonso-Blas and S. Genaim. On the Limits of the Classical Approach to Cost Analysis. In *SAS*, volume 7460 of *LNCS*, pages 405–421. Springer, 2012.
7. H. Anderson, S.C Khoo, S. Andrei, and B. Luca. Calculating Polynomial Runtime Properties. In *APLAS*, volume 3780 of *LNCS*, pages 230–246. Springer, 2005.
8. R. Bagnara, F. Mesnard, A. Pescetti, and E. Zaffanella. A new look at the automatic synthesis of linear ranking functions. *Inf. Comput.*, 215:47–67, 2012.
9. R. Benzinger. Automated higher-order complexity analysis. *Theor. Comput. Sci.*, 318(1-2):79–103, 2004.
10. N. Danner, J. Paykin, and J.S. Royer. A Static Cost Analysis for a Higher-Order Language. In *PLPV*, pages 25–34. ACM, 2013.
11. S. K. Debray and N. Lin. Cost Analysis of Logic Programs. *ACM Trans. Program. Lang. Syst.*, 15(5):826–875, 1993.
12. B. Grobauer. Cost Recurrences for DML Programs. In *ICFP*, pages 253–264. ACM, 2001.
13. S. Gulwani, J.K. Mehra, and T.M. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *POPL*, pages 127–139. ACM, 2009.
14. S. Gulwani and F. Zuleger. The Reachability-Bound Problem. In *PLDI*, pages 292–304. ACM, 2010.
15. J. Hoffmann, , K. Aehlig, and M. Hofmannn. Multivariate Amortized Resource Analysis. *ACM Trans. Program. Lang. Syst.*, 34(3):14:1–14:62, 2012.
16. H. R. Simões, P. B. Vasconcelos, M. Florido, S. Jost, and K. Hammond. Automatic Amortised Analysis of Dynamic Memory Allocation for Lazy Functional Programs. In *ICFP*, pages 165–176. ACM, 2012.
17. P. B. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *IFL*, volume 3145 of *LNCS*, pages 86–101. Springer, 2003.
18. B. Wegbreit. Mechanical Program Analysis. *Commun. ACM*, 18(9):528–539, 1975.
19. F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound Analysis of Imperative Programs with the Size-Change Abstraction. In *SAS*, volume 6887 of *LNCS*, pages 280–297. Springer, 2011.