Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis

Elvira Albert[1], Puri Arenas[1], Samir Genaim[2], and Germán Puebla[2]

[1] DSIC, Complutense University of Madrid, E-28040 Madrid, Spain
[2] CLIP, Technical University of Madrid, E-28660 Boadilla del Monte, Madrid, Spain

Abstract. The classical approach to automatic cost analysis consists of two phases. Given a program and some measure of cost, we first produce *recurrence relations* (RRs) which capture the cost of our program in terms of the size of its input data. Second, we convert such RRs into *closed form* (i.e., without recurrences). Whereas the first phase has received considerable attention, with a number of cost analyses available for a variety of programming languages, the second phase has received comparatively little attention. In this paper we first study the features of RRs generated by automatic cost analysis and discuss why existing computer algebra systems are not appropriate for automatically obtaining closed form solutions nor upper bounds of them. Then we present, to our knowledge, the first practical framework for the fully automatic generation of reasonably accurate upper bounds of RRs originating from cost analysis of a wide range of programs. It is based on the inference of *ranking functions* and *loop invariants* and on *partial evaluation*.

1 Introduction

The aim of *cost analysis* is to obtain *static* information about the execution cost of programs w.r.t. some cost *measure*. Cost analysis has a large application field, which includes resource certification [11,4,16,9], whereby code consumers can reject code which is not guaranteed to run within the resources available. The resources considered include processor cycles, memory usage, or *billable events*, e.g., the number of text messages or bytes sent on a mobile network.

A well-known approach to automatic cost analysis, which dates back to the seminal work of [25], consists of two phases. In the first phase, given a program and some cost measure, we produce a set of equations which captures the cost of our program in terms of the size of its input data. Such equations are generated by converting the iteration constructs of the program (loops and recursion) into recurrences and by inferring *size relations* which approximate how the size of arguments varies. This set of equations can be regarded as *recurrence relations* (RRs for short). Equivalently, it can be regarded as *time bound programs* [22]. The aim of the second phase is to obtain a non-recursive representation of the equations, known as *closed form*. In most cases, it is not possible to find an exact solution and the closed form corresponds to an upper bound.

M. Alpuente and G. Vidal (Eds.): SAS 2008, LNCS 5079, pp. 221–237, 2008.

There are a number of cost analyses available which are based on this approach and which can handle a range of programming languages, including functional [7,18,22,23,24,25], logic [12,20], and imperative [1,3]. While in all such analyses the first phase is studied in detail, the second phase has received comparatively less attention. Basically, there are three different approaches for the second phase. One approach, which is conceptually linked viewing equations as time bound programs, was proposed in [18] and advocated in [22]. It is based on existing source-to-source transformations which convert recursive programs into non-recursive ones. The second approach consists in building restricted recurrence solvers using standard mathematical techniques, as in [12,25]. The third approach consists in relying on existing *computer algebra systems* (CASs for short) such as Mathematica®, MAXIMA, MAPLE, etc., as in [3,7,23,24].

The problem with the three approaches above is that they assume a rather limited form of equations which does not cover the essential features of equations actually generated by automatic cost analysis. In the rest of the paper, we will concentrate on viewing equations as recurrence relations and will use the term *Cost Relation* (CR for short) to refer to the relations produced by automatic cost analysis. In our own experience with [3], we have detected that existing CASs are, in most cases, not capable of handling CRs. We argue that automatically converting CRs into the format accepted by CASs is unfeasible. Furthermore, even in those cases where CASs can be used, the solutions obtained are so complicated that they become useless for most practical purposes. An altogether different approach to cost analysis is based on type systems with resource annotations which does not use equations. Thus, it does not need to obtain closed forms, but it is typically restricted to linear bounds [16]. The need for improved mechanisms for obtaining upper bounds was already pointed out in Hickey and Cohen [14]. A relevant work in this direction is PURRS [5], which has been the first system to provide, in a fully automatic way, non-asymptotic upper and lower bounds for a wide class of recurrences. Unfortunately, and unlike our proposal, it also requires CRs to be deterministic. Marion et. al. [19,8] use a kind of polynomial ranking functions, but the approach is limited to polynomial bounds and can only handle a rather restricted form of CRs.

We believe that the lack of automatic tools for the above second phase is a major reason for the diminished use of automatic cost analysis. In this paper we study the features of CRs and discuss why existing CASs are not appropriate for automatically bounding them. Furthermore, we present, to our knowledge, the first practical framework for the fully automatic inference of reasonably accurate upper bounds for CRs originating from a wide range of programs. To do this, we apply semantic-based transformation and analysis techniques, including inference of *ranking functions*, *loop invariants* and the use of *partial evaluation*.

1.1 Motivating Example

Example 1. Consider the Java code in Fig. 1. It uses a class List for (non sorted) linked lists of integers. Method del receives an input list without repetitions l,

```
void del(List l, int p, int a[], int la, int b[], int lb){
  while (l!=null) {
    if (l.data<p) {
      la=rm_vec(l.data, a, la);
    } else {
      lb=rm_vec(l.data, b, lb);
    }
    l=l.next;
  }
}

int rm_vec(int e, int a[], int la){
  int i=0;
  while (i<la && a[i]<e)  i++;
  for (int j=i; j<la−1; j++) a[j]=a[j+1];
  return la−1;
}
```

(1) $Del(l, a, la, b, lb)=1+C(l, a, la, b, lb)$
$\{b\geq lb, lb\geq 0, a\geq la, la\geq 0, l\geq 0\}$
(2) $C(l, a, la, b, lb)=2$ $\{a\geq la, b\geq lb, b\geq 0, a\geq 0, l=0\}$
(3) $C(l, a, la, b, lb)=$
$25+D(a, la, 0)+E(la, j)+C(l', a, la-1, b, lb)$
$\{a\geq 0, a\geq la, b\geq lb, j\geq 0, b\geq 0, l>l', l>0\}$
(4) $C(l, a, la, b, lb)=$
$24+D(b, lb, 0)+E(lb, j)+C(l', a, la, b, lb-1)$
$\{b\geq 0, b\geq lb, a\geq la, j\geq 0, a\geq 0, l>l', l>0\}$
(5) $D(a, la, i)=3$ $\{i\geq la, a\geq la, i\geq 0\}$
(6) $D(a, la, i)=8$ $\{i<la, a\geq la, i\geq 0\}$
(7) $D(a, la, i)=10+D(a, la, i+1)$ $\{i<la, a\geq la, i\geq 0\}$
(8) $E(la, j)=5$ $\{j\geq la-1, j\geq 0\}$
(9) $E(la, j)=15+E(la, j+1)$ $\{j<la-1, j\geq 0\}$

Fig. 1. Java Code and the Result of Cost Analysis

an integer value p (the *pivot*), two sorted arrays of integers a and b, and two integers la and lb which indicate, respectively, the number of positions occupied in a and b. The array a (resp. b) is expected to contain values which are smaller than the pivot p (resp. greater or equal). Under the assumption that all values in l are contained in either a or b, the method del removes all values in l from the corresponding arrays. The auxiliary method rm_vec removes a given value e from an array a of length la and returns its new length, la−1.

We have applied the cost analysis in [3] on this program in order to approximate the cost of executing the method del in terms of the number of executed bytecode instructions. For this, we first compile the program to bytecode and then analyze the resulting bytecode. Fig. 1 (right) presents the results of analysis, after performing *partial evaluation*, as we will explain in Sec. 6, and inlining equality constraints (e.g., inlining equality $lb'=lb-1$ is done by replacing the occurrences of lb' by $lb-1$). In the analysis results, the data structures in the program are abstracted to their sizes: l represents the maximal *path-length* [15] of the corresponding dynamic structure, which in this case corresponds to the length of the list, a and b are the lengths of the corresponding arrays, and la and lb are the integer values of the corresponding variables. There are nine equations which define the relation Del, which corresponds to the cost of the method del, and three auxiliary recursive relations, C, D, and E. Each of them corresponds to a loop (C: while loop in del; D: while loop in rm_vec; and E: for loop in rm_vec). Each equation is annotated with a set of constraints which capture *size relations* between the values of variables in the left hand side (lhs) and those in the right hand side (rhs). In addition, size relations may contain applicability conditions (i.e., *guards*) by providing constraints which only affect variables in the lhs. Let us explain the equations for D. Eqs. (5) and (6) are base cases which correspond to the exits from the loop when i≥la and a[i]≥e, respectively. Note that the condition a[i]≥e does not appear in the size relation of Eq. (6) nor (7). This is because the array a has been abstracted to its length. Thus, the value in a[i] is no longer observable. For our cost measure , we count 3 bytecode instructions in Eq. (5) and 8 in Eq. (6). The cost of executing an iteration of the loop is

captured by Eq. (7), where the condition $i<la$ must be satisfied and variable i is increased by one at each recursive call. □

1.2 Cost Relations vs. Recurrence Relations

CRs differ from standard RRs in the following ways:

(a) *Non-determinism.* In contrast to RRs, CRs are possibly non-deterministic: equations for the same relation are not required to be mutually exclusive. Even if the programming language is deterministic, *size abstractions* introduce a loss of precision: some guards which make the original program deterministic may not be observable when using the size of arguments instead of their actual values. In Ex. 1, this happens between Eqs. (3) and (4) and also between (6) and (7).

(b) *Inexact size relations.* CRs may have size relations which contain constraints (not equalities). When dealing with realistic programming languages which contain non-linear data structures, such as trees, it is often the case that size analysis does not produce exact results. E.g., analysis may infer that the size of a data structure strictly decreases from one iteration to another, but it may be unable to provide the precise reduction. This happens in Ex. 1 in Eqs. (3) and (4).

(c) *Multiple arguments.* CRs usually depend on several arguments that may increase (variable i in Eq. (7)) or decrease (variable l in Eq. (2)) at each iteration. In fact, the number of times that a relation is executed can be a combination of several of its arguments. E.g., relation E is executed $la-j-1$ times.

Point (a) was detected already in [25], where an explicit `when` operator is added to the RR language to introduce non-determinism, but no complete method for handling it is provided. Point (b) is another source of non-determinism. As a result, CRs do not define functions, but rather relations. Given a relation C and input values $\overline{v}$, there may exist multiple results for $C(\overline{v})$. Sometimes it is possible to automatically convert relations with several arguments into relations with only one. However, in contrast to our approach, it is restricted to very simple cases such as when the CR only count constant cost expressions.

Existing methods for solving RRs are insufficient to bound CRs since they do not cover points (a), (b), and (c) above. On the other hand, CASs can solve complex recurrences (e.g., coefficients to function calls can be polynomials) which our framework cannot handle. However, this additional power is not needed in cost analysis, since such recurrences do not occur as the result of cost analysis.

An obvious way of obtaining upper bounds in non-deterministic CRs would be to introduce a maximization operator. Unfortunately, such operator is not supported by existing CAS. Adding it is far from trivial, since computing the maximum when the equations are not mutually exclusive requires taking into account multiple possibilities, which results in a highly combinatorial problem. Another possibility is to convert CRs into RRs. For this, we need to remove equations from CRs as well as sometimes to replace inexact size relations by exact ones while preserving the worst-case solution. However, this is not possible in general. E.g., in Fig. 1, the maximum cost is obtained when the execution interleaves Eqs. (3) and (4), and therefore we cannot remove either of them.

2 Cost Relations: Evaluation and Upper Bounds

Let us introduce some notation. We use x, y, z, possibly subscripted, to denote variables which range over integers ($\mathbb{Z}$), v, w denote integer values, a, b natural numbers ($\mathbb{N}$) and q rational numbers ($\mathbb{Q}$). We denote by $\mathbb{Q}^+$ (resp. $\mathbb{R}^+$) the set of non-negative rational (resp. real) numbers. We use $\overline{t}$ to denote a sequence of entities $t_1, \ldots, t_n$, for some $n{>}0$. We sometimes apply set operations on sequences. Given $\overline{x}$, an *assignment* for $\overline{x}$ is a sequence $\overline{v}$ (denoted by $[\overline{x}/\overline{v}]$). Given any entity t, $t[\overline{x}/\overline{v}]$ stands for the result of replacing in t each occurrence of x_i by v_i. We use $vars(t)$ to refer to the set of variables occurring in t. A *linear expression* has the form $q_0{+}q_1x_1{+}\cdots{+}q_nx_n$. A *linear constraint* has the form l_1 *op* l_2 where l_1 and l_2 are linear expressions and $op \in \{=, \leq, <, >, \geq\}$. A *size relation* φ is a set of linear constraints (interpreted as a conjunction). The operator $\bar{\exists}\overline{x}.\varphi$ eliminates from φ all variables except for $\overline{x}$. We write $\varphi_1 \models \varphi_2$ to indicate that φ_1 implies φ_2. The following definition presents our notion of *basic cost expression*.

Definition 1 (basic cost expression). Basic cost expressions *are of the form:* $\mathtt{exp}{::=}a|\mathsf{nat}(l)|\mathtt{exp}{+}\mathtt{exp}|\mathtt{exp}{*}\mathtt{exp}|\mathtt{exp}^a|\log_{\mathtt{a}}(\mathtt{exp})|a^{\mathtt{exp}}|\max(S)|\frac{\mathtt{exp}}{a}|\mathtt{exp}{-}a$*, where* $a{\geq}1$*,* l *is a linear expression,* S *is a non empty set of cost expressions,* $\mathsf{nat}{:}\mathbb{Z}{\to}\mathbb{Q}^+$ *is defined as* $\mathsf{nat}(v){=}\max(\{v, 0\})$*, and* $\mathtt{exp}$ *satisfies that for any assignment* $\overline{v}$ *for* $vars(\mathtt{exp})$ *we have that* $\mathtt{exp}[vars(\mathtt{exp})/\overline{v}] \in \mathbb{R}^+$*.*

Basic cost expressions are symbolic expressions which indicate the resources we accumulate and are the non-recursive building blocks for defining cost relations. They enjoy two crucial properties: (1) by definition, they are always evaluated to non negative values; (2) replacing a sub-expression $\mathsf{nat}(l)$ by $\mathsf{nat}(l')$ such that $l'{\geq}l$, results in an upper bound of the original expression.

A *cost relation* C of arity n is a subset of $\mathbb{Z}^n * \mathbb{R}^+$. This means that for a single tuple $\overline{v}$ of integers there can be multiple solutions in $C(\overline{v})$. We use C and D to refer to cost relations. Cost analysis of a program usually produces multiple, interconnected, cost relations. We refer to such sets of cost relations as *cost relation systems* (CRSs for short), which we formally define below.

Definition 2 (Cost Relation System). *A* cost relation system $\mathcal{S}$ *is a set of equations of the form* $\langle C(\overline{x}){=}\mathtt{exp}{+}\sum_{i=0}^{k} D_i(\overline{y}_i),\ \varphi\rangle$ *with* $k{\geq}0$*, where* C *and all* D_i *are cost relations, all variables* $\overline{x}$ *and* $\overline{y}_i$ *are distinct variables;* $\mathtt{exp}$ *is a basic cost expression; and* φ *is a size relation between* $\bar{x}$ *and* $\bar{x}{\cup}vars(\mathtt{exp}){\cup}\bar{y_i}$*.*

In contrast to standard definitions of RRs, the variables which occur in the rhs of the equations in CRSs do not need to be related to those in the lhs by equality constraints. Other constraints such as $\leq$ and $<$ can also be used. We denote by $rel(\mathcal{S})$ the set of cost relations which are defined in $\mathcal{S}$. Also, $def(\mathcal{S}, C)$ denotes the subset of the equations in $\mathcal{S}$ whose lhs is of the form $C(\overline{x})$. W.l.o.g. we assume that all equations in $def(\mathcal{S}, C)$ have the same variable names in the lhs. We assume that any CRS $\mathcal{S}$ is self-contained in the sense that all cost relations which appear in the rhs of an equation in $\mathcal{S}$ must be in $rel(\mathcal{S})$.

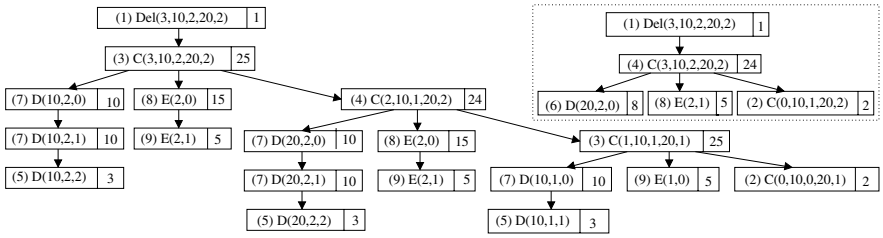

Fig. 2. Two Evaluation Trees for $Del(3, 10, 2, 20, 2)$

We now provide a semantics for CRSs. Given a CRS $\mathcal{S}$, a *call* is of the form $C(\overline{v})$, where $C{\in}rel(\mathcal{S})$ and $\overline{v}$ are integer values. Calls are evaluated in two phases. In the first phase, we build an *evaluation tree* for the call. In the second phase we obtain a *value* in $\mathbb{R}^+$ by adding up the constants which appear in the nodes of the evaluation tree. We make evaluation trees explicit since, as discussed below, our approximation techniques are based on reasoning about the number of nodes and the values in the nodes in such evaluation trees. Evaluation trees are obtained by repeatedly expanding nodes which contain calls to relations. Each expansion is performed w.r.t an appropriate instantiation of a rhs of an applicable equation. If all leaves in the tree contain basic cost expressions then there is no node left to expand and the process terminates. We will represent evaluation trees using nested terms of the form $node(Call,Local\_Cost,Children)$, where $Local\_Cost$ is a constant in $\mathbb{R}^+$ and $Children$ is a sequence of evaluation trees.

Definition 3 (evaluation tree). *Given a CRS $\mathcal{S}$ and a call $C(\overline{v})$, a tree node $(C(\overline{v}), e, \langle T_1, \ldots, T_k\rangle)$ is an* evaluation tree *for $C(\overline{v})$ in $\mathcal{S}$, denoted $\mathsf{Tree}(C(\overline{v}), \mathcal{S})$ if: 1) there is a renamed apart equation $\langle C(\overline{x}){=}\mathtt{exp}{+}\sum_{i=0}^{k} D_i(\overline{y}_i),\ \varphi\rangle \in \mathcal{S}$ s.t. φ' is satisfiable in $\mathbb{Z}$, with $\varphi'{=}\varphi[\overline{x}/\overline{v}]$, and 2) there exist assignments $\overline{w}, \overline{v}_i$ for $vars(\mathtt{exp}), \overline{y}_i$ respectively s.t. $\varphi'[vars(\mathtt{exp})/\overline{w},\ \overline{y}_i/\overline{v}_i]$ is satisfiable in $\mathbb{Z}$, and 3) $e{=}\mathtt{exp}[vars(\mathtt{exp})/\overline{w}]$, T_i is an evaluation tree $\mathsf{Tree}(D_i(\overline{v}_i), \mathcal{S})$ with $i = 0, \ldots, k$.*

In step 1 we look for an equation $\mathcal{E}$ which is applicable for solving $C(\overline{v})$. Note that there may be several equations which are applicable. In step 2 we look for assignments for the variables in the rhs of $\mathcal{E}$ which satisfy the size relations associated to $\mathcal{E}$. This a non-deterministic step as there may be (infinitely many) different assignments which satisfy all size relations. Finally, in step 3 we apply the assignment to $\mathtt{exp}$ and continue recursively evaluating the calls. We use $Trees(C(\overline{v}), \mathcal{S})$ to denote the set of all evaluation trees for $C(\overline{v})$. We define $Answers(C(\overline{v}), \mathcal{S}){=}\{\mathsf{Sum}(T) \mid T{\in}Trees(C(\overline{v}), \mathcal{S})\}$, where $\mathsf{Sum}(T)$ traverses all nodes in T and computes the sum of the cost expressions in them.

Example 2. Fig. 2 shows two possible evaluation trees for $Del(3, 10, 2, 20, 2)$. The tree on the left has maximal cost, whereas the one on the right has minimal cost. A node in either tree contains a call (left box) and its local cost (right box) and it is linked by arrows to its children. We annotate calls with a number in

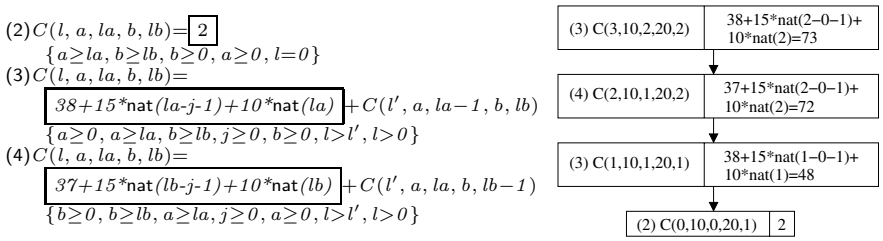

Fig. 3. Self-Contained CR for relation C and a corresponding evaluation tree

parenthesis to indicate the equation which was selected for evaluating such call. Note that, in the recursive call to C in Eqs. (3) and (4), we are allowed to pick any value l' s.t. $l'<l$. In the tree on the left we always assign $l'=l-1$. This is what happens in actual executions of the program. In the tree on the right we assign $l'=l-3$ in the recursive call to C. The latter results in a minimal approximation, however, it does not correspond to any actual execution. This is a side effect of using safe approximations in static analysis: information is correct in the sense that at least one of the evaluation trees must correspond to the actual cost, but there may be other trees with different cost. In fact, there are an infinite number of evaluation trees for our example call, as step 2 can provide an infinite number of assignments to variable j which are compatible with the constraint $j\geq 0$ in Eqs. (3) and (4). This shows that approaches like [13] based on evaluation of CRSs are not of general applicability. Nevertheless, it is possible to find an upper bound for this call since though the number of trees is infinite, infinitely many of them produce equivalent results. □

2.1 Closed Form Upper Bounds for Cost Relations

Let C be a relation over $\mathbb{Z}^n * \mathbb{R}^+$. A function $U{:}\mathbb{Z}^n \to \mathbb{R}^+$ is an *upper bound* of C iff $\forall \overline{v} \in \mathbb{Z}^n, \forall a \in Answers(C(\overline{v}), \mathcal{S}), U(\overline{v}) \geq a$. We use C^+ to refer to an upper bound of C. A function $f{:}\mathbb{Z}^n \to \mathbb{R}^+$ is in *closed form* if it is defined as $f(\overline{x})=\mathtt{exp}$, with $\mathtt{exp}$ a basic cost expression s.t. $vars(\mathtt{exp}) \subseteq \overline{x}$. An important feature of CRSs, inherited from RRs, is their *compositionality*, which allows computing upper bounds of CRSs by concentrating on one relation at a time. I.e., given a cost equation for $C(\overline{x})$ which calls $D(\overline{y})$, we can replace the call to $D(\overline{y})$ by $D^+(\overline{y})$. The resulting relation is trivially an upper bound of the original one. E.g., suppose that we have the following upper bounds: $E^+(la, j)=5+15*\mathsf{nat}(la-j-1)$ and $D^+(a, la, i)=8+10*\mathsf{nat}(la-i)$. Replacing the calls to D and E in equations (3) and (4) by D^+ and E^+ results in the CRS shown in Fig. 3.

The compositionality principle only results in an effective mechanism if all recursions are *direct* (i.e., all cycles are of length one). In that case we can start by computing upper bounds for cost relations which do not depend on any other relations, which we refer to as *standalone cost relations* and continue by replacing

the computed upper bounds on the equations which call such relations. In the following, we formalize our method by assuming standalone cost relations and in Sec. 6 we provide a mechanism for obtaining direct recursion automatically.

Existing approaches to compute upper bounds and asymptotic complexity of RRs, usually applied by hand, are based on reasoning about evaluation trees in terms of their size, depth, number of nodes, etc. They typically consider two categories of nodes: (1) *internal* nodes, which correspond to applying recursive equations, and (2) *leaves* of the tree(s), which correspond to the application of a base (non-recursive) case. The central idea then is to count (or obtain an upper bound on) the number of leaves and the number of internal nodes in the tree separately and then multiply each of these by an upper bound on the cost of the base case and of a recursive step, respectively. For instance, in the evaluation tree in Fig. 3 for the standalone cost relation C, there are three internal nodes and one leaf. The values in the internal nodes, once performed the evaluation of the expressions are 73, 72, and 48, therefore 73 is the worst case. In the case of leaves, the only value is 2. Therefore, the tightest upper bound we can find using this approximation is $3{\times}73{+}1{*}2{=}221 \geq 73{+}72{+}48{+}2{=}193$.

We now extend the approximation scheme mentioned above in order to consider all possible evaluation trees which may exist for a call. In the following, we use $|S|$ to denote the cardinality of a set S. Also, given an evaluation tree T, $\mathit{leaf}(T)$ denotes the set of leaves of T (i.e., those without children) and $\mathit{internal}(T)$ denotes the set of internal nodes (all nodes but the leaves) of T.

Proposition 1 (node-count upper bound). *Let C be a cost relation and let $C^+(\overline{x}) = \mathit{internal}^+(\overline{x}) * \mathit{costr}^+(\overline{x}) + \mathit{leaf}^+(\overline{x}) * \mathit{costnr}^+(\overline{x})$, where $\mathit{internal}^+(\overline{x})$, $\mathit{costr}^+(\overline{x})$, $\mathit{leaf}^+(\overline{x})$ and $\mathit{costnr}^+(\overline{x})$ are closed form functions defined on $\mathbb{Z}^n \to \mathbb{R}^+$. Then, C^+ is an upper bound of C if for all $\overline{v} \in \mathbb{Z}^n$ and for all $T \in \mathit{Trees}(C(\overline{v}), \mathcal{S})$, it holds: (1) $\mathit{internal}^+(\overline{v}) \geq |\mathit{internal}(T)|$ and $\mathit{leaf}^+(\overline{v}) \geq |\mathit{leaf}(T)|$; (2) $\mathit{costr}^+(\overline{v})$ is an upper bound of $\{e \mid \mathit{node}(\_, e, \_) \in \mathit{internal}(T)\}$ and (3) $\mathit{costnr}^+(\overline{v})$ is an upper bound of $\{e \mid \mathit{node}(\_, e, \_) \in \mathit{leaf}(T)\}$.*

3 Upper Bounds on the Number of Nodes

In this section we present an automatic mechanism to obtain safe $\mathit{internal}^+(\overline{x})$ and $\mathit{leaf}^+(\overline{x})$ functions which are valid for any assignment for $\overline{x}$. The basic idea is to first obtain upper bounds b and $h^+(\overline{x})$ on, respectively, the *branching factor* and *height* (the distance from the root to the deepest leaf) of all corresponding evaluation trees, and then use the number of internal nodes and leaves of a *complete* tree with such branching factor and height as an upper bound. Then,

$$\mathit{leaf}^+(\overline{x}) = b^{h^+(\overline{x})} \qquad \mathit{internal}^+(\overline{x}) = \begin{cases} h^+(\overline{x}) & b{=}1 \\ \frac{b^{h^+(\overline{x})}-1}{b-1} & b{\geq}2 \end{cases}$$

For a cost relation C, the branching factor b in any evaluation tree for a call $C(\overline{v})$ is limited by the maximum number of recursive calls which occur in a single equation for C. We now propose a way to compute an upper bound for

the height, h^+. Given an evaluation tree $T \in \mathit{Trees}(C(\overline{v}), \mathcal{S})$ for a cost relation C, consecutive nodes in any branch of T represent consecutive recursive calls which occur during the evaluation of $C(\overline{v})$. Therefore, bounding the height of a tree may be reduced to bounding consecutive recursive calls. The notion of *loop* in a cost relation, which we introduce below, is used to model consecutive calls.

Definition 4. *Let* $\mathcal{E}=\langle C(\overline{x})=\mathsf{exp}+\sum_{i=1}^{k} C(\overline{y_i}), \varphi\rangle$ *be an equation for a cost relation* C*. Then,* $\mathit{Loops}(\mathcal{E})=\{\langle C(\overline{x})\rightarrow C(\bar{y}_i), \varphi'\rangle \mid \varphi'=\bar{\exists}\bar{x}\cup\bar{y}_i.\varphi, i=1\cdots k\}$ *is the set of* loops *induced by* $\mathcal{E}$*. Similarly,* $\mathit{Loops}(C) = \cup_{\mathcal{E}\in def(\mathcal{S},C)}\mathit{Loops}(\mathcal{E})$*.*

Example 3. Eqs. (3) and (4) in Fig. 3 induce the following two loops:

$(3)\langle C(l, a, la, b, lb)\rightarrow C(l', a, la', b, lb), \varphi'_1=\{a\geq 0, a\geq la, b\geq lb, b\geq 0, l>l', l>0, la'=la-1\}\rangle$
$(4)\langle C(l, a, la, b, lb)\rightarrow C(l', a, la, b, lb'), \varphi'_2=\{b\geq 0, b\geq lb, a\geq la, a\geq 0, l>l', l>0, lb'=lb-1\}\rangle$

Bounding the number of consecutive recursive calls is extensively used in the context of termination analysis. It is usually done by proving that there is a function f from the loop's arguments to a *well-founded* partial order which decreases in any two consecutive calls and which guarantees the absence of infinite traces, and thus termination. These functions are usually called *ranking functions*. We propose to use the ranking function to generate a h^+ function. In practice, we use [21] to generate functions which are defined as follows: a function $f{:}\mathbb{Z}^n\mapsto\mathbb{Z}$ is a *ranking function* for a loop $\langle C(\bar{x})\rightarrow C(\bar{y}), \varphi\rangle$ if $\varphi\models f(\bar{x})>f(\bar{y})$ and $\varphi\models f(\bar{x})\geq 0$.

Example 4. The function $f_C(l, a, la, b, lb)=l$ is a ranking function for C in the cost relation in Fig. 3. Note that φ'_1 and φ'_2 in the above loops of C contain the constraints $\{l>l', l>0\}$ which is enough to guarantee that f_C is decreasing and well-founded. The height of the evaluation tree for $C(3, 10, 2, 20, 2)$ is precisely predicted by $f_C(3, 10, 2, 20, 2)=3$. Ranking functions may involve several arguments, e.g., $f_D(a, la, i)=la-i$ is a ranking function for $\langle D(a, la, i) \rightarrow D(a, la, i'), \{i'=i+1, i<la, a\geq la, i\geq 0\}\rangle$ which comes from Eq. (7). □

Observe that the use of global ranking functions allows bounding the number of iterations of possibly non-deterministic CRSs with multiple arguments (see Sec. 1.2). In order to be able to define h^+ in terms of the ranking function, one thing to fix is that the ranking function might return a negative value when is applied to values which correspond to base cases (leaves of the tree). Therefore, we define $h^+(\overline{x})=\mathsf{nat}(f_C(\overline{x}))$. Function nat guarantees that negative values are lifted to 0 and, therefore, they provide a correct approximation for the height of evaluation trees with a single node. Even though the ranking function provides an upper bound for the height of the corresponding trees, in some cases we can further refine it and obtain a tighter upper bound. For example, if the difference between the value of the ranking function in each two consecutive calls is larger than a constant $\delta>1$, then $\lceil \mathsf{nat}(\frac{f_C(\bar{x})}{\delta})\rceil$ is a tighter upper bound. A more interesting case, if each loop $\langle C(\overline{x})\rightarrow C(\overline{y}), \varphi\rangle \in \mathit{Loops}(C)$ satisfies $\varphi\models f_C(\overline{x})\geq k*f_C(\overline{y})$ where $k>1$, then the height of the tree is bounded by $\lceil \log_k(\mathsf{nat}(f_C(\overline{v})+1))\rceil$.

4 Estimating the Cost Per Node

Consider the evaluation tree in Fig. 3. Note that all expressions in the nodes are instances of the expressions which appear in the corresponding equations. Thus, computing $costr^+(\overline{x})$ and $costnr^+(\overline{x})$ can be done by first finding an upper bound of such expressions and then combining them through a *max* operator. We first compute *invariants* for the values that the expression variables can take w.r.t. the initial values, and use them to derive upper bounds for such expressions.

4.1 Invariants

Computing an *invariant* (in terms of linear constraints) that holds in all *calling contexts* (*contexts* for short) to a relation C between the arguments at the initial call and at each call during the evaluation can be done by using $Loops(C)$. Intuitively, if we know that a linear constraint ψ holds between the arguments of the initial call $C(\overline{x}_0)$ and those of a recursive call $C(\overline{x})$, denoted $\langle C(\overline{x}_0){\rightsquigarrow}C(\overline{x}),\psi\rangle$, and we have a loop $\langle C(\overline{x}){\rightarrow}C(\overline{y}),\varphi\rangle{\in}Loops(C)$, then we can apply the loop one more step and get the new *calling context* $\langle C(\overline{x}_0){\rightsquigarrow}C(\overline{y}),\overline{\exists\overline{x}_0\cup\overline{y}}.\psi{\wedge}\varphi\rangle$.

Definition 5 (loop invariants). *For a relation C, let $\mathcal{T}$ be an operator defined:*

$$\mathcal{T}(X)=\left\{\langle C(\overline{x}_0){\rightsquigarrow}C(\overline{y}),\psi'\rangle \;\middle|\; \begin{array}{l}\langle C(\overline{x}_0){\rightsquigarrow}C(\overline{x}),\psi\rangle{\in}X,\langle C(\overline{x}){\rightarrow}C(\overline{y}),\varphi\rangle{\in}Loops(C),\\ \psi'=\overline{\exists\overline{x}_0\cup\overline{y}}.\psi{\wedge}\varphi\end{array}\right\}$$

which derives a set of contexts, from a given context X, by applying all loops, then the loop invariants I is $lfp{\cup_{i\geq 0}}\mathcal{T}^i(I_0)$ where $I_0=\{\langle C(\overline{x}_0){\rightsquigarrow}C(\overline{x}),\{\overline{x}_0{=}\overline{x}\}\rangle\}$.

Example 5. Let us compute I for the loops in Sec. 3. The initial context is $I_1{=}\langle C(\bar{x}_0){\rightsquigarrow}C(\bar{x}),\{l{=}l_0,a{=}a_0,la{=}la_0,b{=}b_0,lb{=}lb_0\}\rangle$ where $\bar{x}_0{=}\langle l_0,a_0,la_0,b_0,lb_0\rangle$ and $\bar{x}{=}\langle l,a,la,b,lb\rangle$. In the first iteration we compute $\mathcal{T}^0(\{I_1\})$ which by definition is $\{I_1\}$. In the second iteration we compute $\mathcal{T}^1(\{I_1\})$ which results in

$$I_2{=}\langle C(\bar{x}_0){\rightsquigarrow}C(\bar{x}),\{\underline{l{<}l_0},a{=}a_0,\underline{la{=}la_0{-}1},b{=}b_0,lb{=}lb_0,\underline{l_0{>}0}\}\rangle$$
$$I_3{=}\langle C(\bar{x}_0){\rightsquigarrow}C(\bar{x}),\{\underline{l{<}l_0},a{=}a_0,la{=}la_0,b{=}b_0,\underline{lb{=}lb_0{-}1},\underline{l_0{>}0}\}\rangle$$

where I_2 and I_3 correspond to applying respectively the first loop and second loops on I_1. The underlined constraints are the modifications due to the application of the loop. Note that in I_2 the variable la_0 decreases by one, and in I_3 lb_0 decreases by one. The third iteration $\mathcal{T}^2(\{I_1\})$, i.e. $\mathcal{T}(\{I_2,I_3\})$, results in

$$I_4{=}\langle C(\bar{x}_0){\rightsquigarrow}C(\bar{x}),\{l{<}l_0,a{=}a_0,\underline{la{=}la_0{-}2},b{=}b_0,lb{=}lb_0,l_0{>}0\}\rangle$$
$$I_5{=}\langle C(\bar{x}_0){\rightsquigarrow}C(\bar{x}),\{l{<}l_0,a{=}a_0,\underline{la{=}la_0{-}1},b{=}b_0,\underline{lb{=}lb_0{-}1},l_0{>}0\}\rangle$$
$$I_6{=}\langle C(\bar{x}_0){\rightsquigarrow}C(\bar{x}),\{l{<}l_0,a{=}a_0,la{=}la_0,b{=}b_0,\underline{lb{=}lb_0{-}2},l_0{>}0\}\rangle$$
$$I_7{=}\langle C(\bar{x}_0){\rightsquigarrow}C(\bar{x}),\{l{<}l_0,a{=}a_0,\underline{la{=}la_0{-}1},b{=}b_0,lb{=}lb_0{-}1,l_0{>}0\}\rangle$$

where I_4 and I_5 originate from applying the loops to I_2, and I_6 and I_7 from applying the loops to I_3. The modifications on the constraints reflect that, when applying a loop, either we decrease *la* or *lb*. After three iterations, the invariant I includes $I_1\cdots I_7$. More iterations will add more contexts that further modify the value of *la* or *lb*. Therefore, the invariant I grows indefinitely in this case. □

In practice, we approximate I using abstract interpretation over, for instance, the domain of convex polyhedra [10], whereby we obtain the invariant $\Psi = \langle C(\overline{x}_0) \rightsquigarrow C(\overline{x}), \{l \leq l_0, a = a_0, la \leq la_0, b = b_0, lb \leq lb_0\}\rangle$.

4.2 Upper Bounds on Cost Expressions

Once invariants are available, finding upper bounds of cost expressions can be done by maximizing their nat parts independently. This is possible due to the monotonicity property of cost expressions. Consider, for example, the expression $\mathsf{nat}(la-j-1)$ which appears in equation (3) of Fig. 3. We want to infer an upper bound of the values that it can be evaluated to in terms of the input values $\langle l_0, a_0, la_0, b_0, lb_0\rangle$. We have inferred, in Sec. 4.1, that whenever we call C the invariant Ψ holds, from which we can see that the maximum value that la can take is la_0. In addition, from the local size relations φ of equation (3) we know that $j \geq 0$. Since $la-j-1$ takes its maximal value when la is maximal and j is minimal, the expression la_0-1 is an upper bound for $la-j-1$. This can be done automatically using linear constraints tools [6]. Given a cost equation $\langle C(\overline{x}) = \mathtt{exp} + \sum_{i=0}^{k} C(\overline{y}_i),\ \varphi\rangle$ and an invariant $\langle C(\overline{x}_0) \rightsquigarrow C(\overline{x}), \Psi\rangle$, the function below computes an upper bound for exp by maximizing its nat components.

```
1: function ub_exp(exp,x̄0,φ,Ψ)
2:     mexp=exp
3:     for all nat(f)∈exp do
4:         Ψ'=∃x̄0, r.(φ∧Ψ∧(r=f))        // r is a fresh variable
5:         if ∃f' s.t. vars(f')⊆x̄0 and Ψ'⊨r≤f' then mexp=mexp[nat(f)/nat(f')]
6:         else return ∞
7:     return mexp
```

This function computes an upper bound f' for each expression f which occurs inside a nat operator and then replaces in exp all such f expressions with their corresponding upper bounds (line 5). If it cannot find an upper bound, the method returns ∞ (line 6). The *ub_exp* function is complete in the sense that if Ψ and φ imply that there is an upper bound for a given $\mathsf{nat}(f)$, then we can find one by *syntactically* looking on Ψ' (line 4).

Example 6. Applying *ub_exp* to $\mathtt{exp}_3$ and $\mathtt{exp}_4$ of Eqs. (3) and (4) in Fig. 3 w.r.t. the invariant we have computed in Sec. 4.1 results in $\mathtt{mexp}_3 = 38 + 15 * \mathsf{nat}(la_0 - 1) + 10 * \mathsf{nat}(la_0)$ and $\mathtt{mexp}_4 = 37 + 15 * \mathsf{nat}(lb_0 - 1) + 10 * \mathsf{nat}(lb_0)$. □

Theorem 1. *Let $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$ be a cost relation where $\mathcal{S}_1$ and $\mathcal{S}_2$ are respectively the sets of non-recursive and recursive equations for C, and let $\mathcal{I} = \langle C(\overline{x}_0) \rightsquigarrow C(\overline{x}), \Psi\rangle$ be a loop invariant for C; $E_i = \{ub\_exp(\mathtt{exp}, \overline{x}_0, \varphi, \Psi) \mid \langle C(\overline{x}) = \mathtt{exp} + \sum_{j=0}^{k} C(\overline{y}_j), \varphi\rangle \in \mathcal{S}_i\}$; $costnr^+(\overline{x}_0) = \max(E_1)$ and $costr^+(\overline{x}_0) = \max(E_2)$. Then for any call $C(\overline{v})$ and for all $T \in Trees(C(\overline{v}), \mathcal{S})$: (1) $\forall node(\_, e, \_) \in internal(T)$ we have $costr^+(\overline{v}) \geq e$; and (2) $\forall node(\_, e, \_) \in leaf(T)$ we have $costnr^+(\overline{v}) \geq e$.*

Example 7. At this point we have all the pieces in order to compute an upper bound for the CRS depicted in Fig. 1 as described in Prop. 1. We start by computing upper bounds for E and D as they are cost relations:

	Ranking Function	$costnr^+$	$costr^+$	*Upper Bound*
$E(la_0, j_0)$	$\mathsf{nat}(la_0 - j_0 - 1)$	5	15	$5+15*\mathsf{nat}(la_0 - j_0 - 1)$
$D(a_0, la_0, i_0)$	$\mathsf{nat}(la_0 - i_0)$	8	10	$8+10*\mathsf{nat}(la_0 - i_0)$

These upper bounds can then be substituted in the equations (3) and (4) which results in the cost relation for C depicted in Fig. 3. We have already computed a ranking function for C in Ex. 4 and $costnr^+$ and $costr^+$ in Ex. 6, which are then combined into $C^+(l_0, a_0, la_0, b_0, lb_0) = 2+\mathsf{nat}(l_0)*max(\{\mathtt{mexp}_3, \mathtt{mexp}_4\})$. Reasoning similarly, for *Del* we get the upper bound shown in Table 1. □

5 Improving Accuracy in Divide and Conquer Programs

For some CRSs, we can obtain a more accurate upper bound by approximating the cost of *levels* instead of approximating the cost of nodes, as indicated by Prop. 1. Given an evaluation tree T, we denote by $\mathsf{Sum\_Level}(T, i)$ the sum of the values of all nodes in T which are at depth i, i.e., at distance i from the root.

Proposition 2 (level-count upper bound). *Let C be a cost relation and let C^+ be a function defined as: $C^+(\overline{x}) = l^+(\overline{x}) * costl^+(\overline{x})$, where $l^+(\overline{x})$ and $costl^+(\overline{x})$ are closed form functions defined on $\mathbb{Z}^n \to \mathbb{R}^+$. Then, C^+ is an upper bound of C if for all $\overline{v} \in \mathbb{Z}^n$ and $T \in Trees(C(\overline{v}), \mathcal{S})$, it holds: (1) $l^+(\overline{v}) \geq depth(T) + 1$; and (2) $\forall i \in \{0, \ldots, depth(T)\}$ we have that $costl^+(\overline{v}) \geq \mathsf{Sum\_Level}(T, i)$.*

The function l^+ can simply be defined as $l^+(\overline{x}) = \mathsf{nat}(f_C(\overline{x}))+1$ (see Sec. 3). Finding an accurate $costl^+$ function is not easy in general, which makes Prop. 2 not as widely applicable as Prop. 1. However, evaluation trees for *divide and conquer* programs satisfy that $\mathsf{Sum\_Level}(T, k) \geq \mathsf{Sum\_Level}(T, k+1)$, i.e., the cost per level does not increase from one level to another. In that case, we can take the cost of the root node as an upper bound of $costl^+(\overline{x})$. A sufficient condition for a cost relation falling into the divide and conquer class is that each cost expression that is contributed by an equation is greater than or equal to the sum of the cost expressions contributed by the corresponding immediate recursive calls. This check is implemented in our prototype using [6].

Consider a CRS with the two equations $\langle C(n) = 0, \{n \leq 0\}\rangle$ and $\langle C(n) = \mathsf{nat}(n)+C(n_1)+C(n_2), \varphi\rangle$ where $\varphi = \{n>0, n_1+n_2+1 \leq n, n \geq 2*n_1, n \geq 2*n_2, n_1 \geq 0, n_2 \geq 0\}$. It corresponds to a divide and conquer problem such as merge-sort. In order to prove that $\mathsf{Sum\_Level}$ does not increase, it is enough to check that, in the second equation, n is greater than or equal to the sum of the expressions that immediately result from the calls $C(n_1)$ and $C(n_2)$, which are n_1 and n_2 respectively. This can be done by simply checking that $\varphi \models n \geq n_1+n_2$. Then, $costl^+(\overline{x}) = max\{0, \mathsf{nat}(x)\} = \mathsf{nat}(x)$. Thus, given that $l^+(x) = \lceil \log_2(\mathsf{nat}(x)+1) \rceil + 1$, we obtain the upper bound $\mathsf{nat}(x) * (\lceil \log_2(\mathsf{nat}(x)+1) \rceil + 1)$. Note that by using the node-count approach we would obtain $\mathsf{nat}(x) * (2^{\mathsf{nat}(x)} - 1)$ as upper bound.

6 Direct Recursion Using Partial Evaluation

Automatically generated CRSs often contain recursions which are not direct, i.e., cycles involve more than one function. E.g., the actual CRS obtained for

the program in Fig. 1 by the analysis in [3] differs from that shown in the right hand side of Fig. 1 in that, instead of Eqs. (8) and (9), the "for" loop results in:

(8') $E(la,j)=5+F(la,j,j',la')$ $\{j'=j, la'=la-1, j'\geq 0\}$
(9') $F(la,j,j',la')=H(j',la')$ $\{j'\geq la'\}$
(10) $F(la,j,j',la')=G(la,j,j',la')$ $\{j'<la'\}$
(11) $H(j',la')=0$ $\{\}$
(12) $G(la,j,j',la')=10+E(la,j+1)$ $\{j<la-1, j\geq 0, la-la'=1, j'=j\}$

Now, E captures the cost of the loop condition "$j<la-1$" (5 cost units) plus the cost of its continuation, captured by F. Eq. (9') corresponds to the exit of the loop (it calls H, Eq. (11), which has 0 cost). Eq. (10) captures the cost of one iteration by calling G, Eq. (12), which accumulates 10 units and returns to E.

In this section we present an automatic transformation of CRSs into *directly recursive* form. The transformation is based on *partial evaluation* (PE) [17] and it is performed by replacing calls to intermediate relations by their definitions using *unfolding*. The first step in the transformation is to find a *binding time classification* (or BTC for short) which declares which relations are *residual*, i.e., they have to remain in the CRS. The remaining relations are considered *unfoldable*, i.e., they are eliminated. For computing BTCs, we associate to each CRS $\mathcal{S}$ a *call graph*, denoted $\mathcal{G}(\mathcal{S})$, which is the directed graph obtained from $\mathcal{S}$ by taking $rel(\mathcal{S})$ as the set of nodes and by including an arc (C, D) iff D appears in the rhs of an equation for C. The following definition provides sufficient conditions on a BTC which guarantee that we obtain a directly recursive CRS.

Definition 6. *Let $\mathcal{G}(\mathcal{S})$ be the call graph of $\mathcal{S}$ and let SCC be its strongly connected components. A BTC* btc *for $\mathcal{S}$ is* directly recursive *if for all $S\in SCC$ the following conditions hold: (1) if $s_1, s_2\in S$ and $s_1, s_2\in$ btc, then $s_1=s_2$; and (2) if S has a cycle, then there exists $s\in S$ such that $s\in$ btc.*

Condition 1 ensures that all recursions in the transformed CRS are direct, as there is only one residual relation per SCC. Condition 2 guarantees that the unfolding process terminates, as there is a residual relation per cycle. A directly recursive BTC for the above example is btc$=\{E\}$. In our implementation we only include in the BTC the *covering point* (i.e., a node which is part of all cycles) of SCCs which contain cycles, but no node is included for SCCs without cycles. This way of computing BTCs, in addition to ensuring direct recursion, also eliminates all relations which are not part of cycles (such as H in our example).

We now define unfolding in the context of CRSs. Such unfolding is guided by a BTC and at each step it combines both cost expressions and size relations.

Definition 7 (unfolding). *Given a CRS $\mathcal{S}$, a call $C(\overline{x}_0)$ s.t. $C\in rel(\mathcal{S})$, a size relation $\varphi_{\overline{x}_0}$ over $\overline{x}_0$, and a BTC* btc *for $\mathcal{S}$, a pair $\langle E,\varphi\rangle$ is an* unfolding *for $C(\overline{x}_0)$ and $\varphi_{\overline{x}_0}$ in $\mathcal{S}$ w.r.t.* btc*, denoted* $\mathsf{Unfold}(\langle C(\overline{x}_0),\varphi_{\overline{x}_0}\rangle,\mathcal{S},\mathsf{btc})\rightsquigarrow\langle E,\varphi\rangle$*, if either of the following conditions hold:*

(res) $C\in\mathsf{btc}\wedge\varphi\neq\mathsf{true}\wedge\langle E,\varphi\rangle=\langle C(\overline{x}_0),\varphi_{\overline{x}_0}\rangle$;
(unf) $(C\notin\mathsf{btc}\vee\varphi=\mathsf{true})\wedge\langle E,\varphi\rangle=\langle(\mathsf{exp}+e_1+\ldots+e_k),\varphi'\bigwedge_{i=1..k}\varphi_i\rangle$

where $\langle C(\overline{x})=\mathsf{exp}+\sum_{i=1}^{k}D_i(\overline{y}_i),\varphi_C\rangle$ is a renamed apart equation in $\mathcal{S}$ s.t. $\varphi'=\varphi_{\overline{x}_0}\wedge\varphi_C[\overline{x}/\overline{x}_0]$ is satisfiable in $\mathbb{Z}$ and $\forall 1\leq i\leq k$ $\mathsf{Unfold}(\langle D_i(\overline{y_i}),\varphi'\rangle,\mathcal{S},\mathsf{btc})\rightsquigarrow\langle e_i,\varphi_i\rangle$.

The first case, (**res**), is required for termination. When we call a relation C which is marked as residual, we simply return the initial call $C(\overline{x}_0)$ and size relation $\varphi_{\overline{x}_0}$, as long as the current size relation $\varphi_{\overline{x}_0}$ is not the initial one (true). The latter condition is added in order to force the initial unfolding step for relations marked as residual. In all subsequent calls to Unfold different from the initial one, the size relation is different from true. The second case (**unf**) corresponds to continuing the unfolding process. Note that step 1 is non-deterministic, since often cost relations contain several equations. Since expressions are transitively unfolded, step 2 may also provide multiple solutions. Also, if the final size relation φ is unsatisfiable, we simply do not regard $\langle E, \varphi\rangle$ as a valid unfolding.

Example 8. Given the initial call $\langle E(la, j), true\rangle$, we obtain an unfolding by performing the following steps, denoted by $\overset{e}{\leadsto}$ where e is the selected equation:
$\langle E(la,j), true\rangle \overset{(8')}{\leadsto} \langle 5{+}F(la,j,j',la'), \{j'{=}j, la'{=}la{-}1, j'{\geq}0\}\rangle \overset{(10)}{\leadsto}$
$\langle 5{+}G(la,j,j',la'), \{j'{=}j, la'{=}la{-}1, j'{\geq}0, j'{<}la'\}\rangle \overset{(12)}{\leadsto} \langle 15{+}E(la,j''), \{j{<}la{-}1, j{\geq}0\}\rangle$
The call $E(la, j'')$ is not further unfolded as E belongs to btc and $\varphi{\neq}true$. □

From each result of unfolding we can build a *residual equation*. Given the unfolding $\mathsf{Unfold}(\langle C(\overline{x}_0), \varphi_{\overline{x}_0}\rangle, \mathcal{S}, \mathsf{btc}) \rightsquigarrow \langle E, \varphi\rangle$ its corresponding residual equation is $\langle C(\overline{x}_0){=}E, \varphi\rangle$. As customary in PE, a *partial evaluation* of C is obtained by collecting all residual equations for the call $\langle C(\overline{x}_0), \mathsf{true}\rangle$. The PE of $\langle E(la, j), true\rangle$ results in Eqs. (8) and (9) of Fig. 1. Eq. (9) is obtained from the unfolding steps depicted in Ex. 8 and Eq. (8) from unfolding w.r.t. Eqs. (8'), (9'), and (11).

Correctness of PE ensures that the solutions of CRSs are preserved. Regarding completeness, we can obtain direct recursion if all SCCs in the call graph have covering point(s). Importantly, structured loops (for, while, etc.) and recursive patterns found in most programs result in CRSs that satisfy this property. In addition, before applying PE, we check that the CRS terminates [2] with respect to the initial query, otherwise we might compromise non-termination and thus lead to incorrect upper bounds. We believe this check is not required when CRSs are generated from imperative programs.

7 Experiments in Cost Analysis of Java Bytecode

A prototype implementation in Ciao Prolog, which uses PPL [6] for manipulating linear constraints, is available at `http://www.cliplab.org/Systems/PUBS`. We have performed a series of experiments which are shown in Table 1. We have used CRSs automatically generated by the cost analyzer of Java bytecode described in [3] using two cost measures: heap consumption for those marked with "*", and the number of executed bytecode instructions for the rest. The benchmarks are presented in increasing complexity order and grouped by asymptotic class. Those marked with M were solved using Mathematica® by [3] but after significant human intervention. The marks a, b and c after the name indicate, respectively, if the CRS is non-deterministic, has inexact size relations and multiple arguments (Sec. 1.2). Column $\#_{eq}$ shows the number of equations before PE (in brackets

Table 1. Experiments on Cost Analysis of Java Bytecode

Benchmark		$\#_{eq}$	**T**	$\#^c_{eq}$	$\mathbf{T}_{pe}$	$\mathbf{T}_{ub}$	**Rat.**	**Upper Bound**
Polynomial*	abc	23 (3)	13	346 (70)	174	649	2.4	216
DivByTwo	ab	9 (3)	3	323 (68)	166	596	2.4	$8\log_2(\mathsf{nat}(2x-1)+1)+14$
Factorial$^{\mathsf{M}}$		8 (2)	4	314 (66)	165	590	2.4	$9\mathsf{nat}(x)+4$
ArrayRev$^{\mathsf{M}}$	a	9 (3)	4	305 (64)	165	579	2.4	$14\mathsf{nat}(x)+12$
Concat$^{\mathsf{M}}$	ac	14 (5)	13	296 (62)	158	538	2.4	$11\mathsf{nat}(x)+11\mathsf{nat}(y)+25$
Incr$^{\mathsf{M}}$	ac	28 (5)	29	282 (58)	155	490	2.3	$19\mathsf{nat}(x+1)+9$
ListRev$^{\mathsf{M}}$	abc	9 (3)	4	254 (54)	144	415	2.2	$13\mathsf{nat}(x)+8$
MergeList	abc	21 (4)	18	245 (52)	138	406	2.2	$29\mathsf{nat}(x+y)+26$
Power		8 (2)	3	223 (48)	125	371	2.2	$10\mathsf{nat}(x)+4$
Cons*	ab	22 (2)	6	214 (46)	123	359	2.3	$22\mathsf{nat}(x-1)+24$
EvenDigits	abc	18 (5)	9	191 (44)	115	322	2.3	$\mathsf{nat}(x)(8\log_2(\mathsf{nat}(2x-3)+1)+24)+9\mathsf{nat}(x)+9$
ListInter	abc	37 (9)	59	173 (40)	110	298	2.4	$\mathsf{nat}(x)(10\mathsf{nat}(y)+43)+21$
SelectOrd	ac	19 (6)	27	136 (32)	86	198	2.1	$\mathsf{nat}(x-2)(17\mathsf{nat}(x-2)+34)+9$
FactSum	a	17 (5)	8	117 (27)	76	173	2.1	$\mathsf{nat}(x+1)(9\mathsf{nat}(x)+16)+6$
Delete	abc	33 (9)	125	100 (23)	71	165	2.4	$3+\mathsf{nat}(l)\max(38+15\mathsf{nat}(la-1)+10\mathsf{nat}(la),$ $37+15\mathsf{nat}(lb-1)+10\mathsf{nat}(lb))$
MatMult$^{\mathsf{M}}$	ac	19 (7)	23	67 (15)	27	40	1.0	$\mathsf{nat}(y)(\mathsf{nat}(x)(27\mathsf{nat}(x))+10)+17$
Hanoi		9 (2)	4	48 (8)	23	17	0.8	$20(2^{\mathsf{nat}(x)})$-17
Fibonacci$^{\mathsf{M}}$		8 (2)	5	39 (6)	20	13	0.8	$18(2^{\mathsf{nat}(x-1)})$-13
BST*	ab	31 (4)	26	31 (4)	19	7	0.9	$96(2^{\mathsf{nat}(x)})$-49

after PE). Note that PE greatly reduces $\#_{eq}$ in all benchmarks. Column **T** shows the total runtime in milliseconds. The experiments have been performed on an Intel Core 2 Duo 1.86GHz with 2GB of RAM, running Linux.

The next four columns aim at demonstrating the scalability of our approach. To do so, we connect the CRSs for the different benchmarks by introducing a call from each CRS to the one appearing immediately below it in the table. Such call is always introduced in a recursive equation. Column $\#^c_{eq}$ shows the number of equations we want to solve in each case (in brackets after PE). Reading this column bottom-up, we can see that BST has the same number of equations as the original one and that, progressively, each benchmark adds its own number of equations to $\#^c_{eq}$. Thus, in the first row we have a CRS with all the equations connected, i.e., we compute an upper bound of CRS with at least 19 nested loops and 346 equations. The total runtime is split into $\mathbf{T}_{pe}$ and $\mathbf{T}_{ub}$, where $\mathbf{T}_{pe}$ is the time of PE and it shows that even though PE is a *global* transformation, its time efficiency is linear with the number of equations. Our system solves 346 equations in $823ms$. Column **Rat**. shows the total time per equation. The ratio is small for benchmarks with few equations, and for reasonably large CRSs (from Delete upwards) it almost has no variation (2.1–2.4 ms/eq). The small increase is due to the fact that the equations count more complex expressions as we connect more benchmarks. This demonstrates that our approach is totally scalable, even if the implementation is preliminary. The upper bound expressions get considerably large when the benchmarks are composed together. We are currently implementing standard techniques for simplification of arithmetic expressions.

Acknowledgments. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT* project, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project. S. Genaim was supported by a *Juan de la Cierva* Fellowship awarded by MEC.

References

1. Adachi, A., Kasai, T., Moriya, E.: A theoretical study of the time analysis of programs. In: Becvar, J. (ed.) MFCS 1979. LNCS, vol. 74, pp. 201–207. Springer, Heidelberg (1979)
2. Albert, E., Arenas, P., Codish, M., Genaim, S., Puebla, G., Zanardini, D.: Termination Analysis of Java Bytecode. In: Proc. of FMOODS. LNCS, Springer, Heidelberg (to appear, 2008)
3. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of java bytecode. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, Springer, Heidelberg (2007)
4. Aspinall, D., Gilmore, S., Hofmann, M., Sannella, D., Stark, I.: Mobile Resource Guarantees for Smart Devices. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, Springer, Heidelberg (2005)
5. Bagnara, R., Pescetti, A., Zaccagnini, A., Zaffanella, E.: PURRS: Towards computer algebra support for fully automatic worst-case complexity analysis. Technical report, arXiv:cs/0512056 (2005), `http://arxiv.org/`
6. Bagnara, R., Ricci, E., Zaffanella, E., Hill, P.M.: Possibly not closed convex polyhedra and the Parma Polyhedra Library. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, Springer, Heidelberg (2002)
7. Benzinger, R.: Automated higher-order complexity analysis. In: TCS, vol. 318(1-2) (2004)
8. Bonfante, G., Marion, J.-Y., Moyen, J.-Y.: Quasi-interpretations and small space bounds. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, Springer, Heidelberg (2005)
9. Chander, A., Espinosa, D., Islam, N., Lee, P., Necula, G.: Enforcing resource bounds via static verification of dynamic checks. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, Springer, Heidelberg (2005)
10. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL (1978)
11. Crary, K., Weirich, S.: Resource bound certification. In: POPL (2000)
12. Debray, S.K., Lin, N.W.: Cost analysis of logic programs. TOPLAS 15(5) (1993)
13. Gómez, G., Liu, Y.A.: Automatic time-bound analysis for a higher-order language. In: PEPM, ACM Press, New York (2002)
14. Hickey, T., Cohen, J.: Automating program analysis. J. ACM 35(1) (1988)
15. Hill, P.M., Payet, E., Spoto, F.: Path-length analysis of object-oriented programs. In: Proc. EAAI, Elsevier, Amsterdam (2006)
16. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. In: POPL (2003)
17. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall, New York (1993)
18. Le Metayer, D.: ACE: An Automatic Complexity Evaluator. TOPLAS 10(2) (1988)

19. Marion, J.-Y., Péchoux, R.: Resource analysis by sup-interpretation. In: Hagiya, M., Wadler, P. (eds.) FLOPS 2006. LNCS, vol. 3945, Springer, Heidelberg (2006)
20. Navas, J., Mera, E., López-García, P., Hermenegildo, M.: User-definable resource bounds analysis for logic programs. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, Springer, Heidelberg (2007)
21. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, Springer, Heidelberg (2004)
22. Rosendahl, M.: Automatic Complexity Analysis. In: FPCA, ACM Press, New York (1989)
23. Sands, D.: A naïve time analysis and its theory of cost equivalence. Journal of Logic and Computation 5(4) (1995)
24. Wadler, P.: Strictness analysis aids time analysis. In: POPL (1988)
25. Wegbreit, B.: Mechanical Program Analysis. Comm. of the ACM 18(9) (1975)