J Autom Reasoning
DOI 10.1007/s10817-010-9174-1

Closed-Form Upper Bounds in Static Cost Analysis

**Elvira Albert · Puri Arenas · Samir Genaim ·
Germán Puebla**

Received: 11 November 2008 / Accepted: 12 April 2010

Abstract The classical approach to automatic cost analysis consists of two phases. Given a program and some measure of cost, the analysis first produces *cost relations* (*CRs*), i.e., recursive equations which capture the cost of the program in terms of the size of its input data. Second, *CRs* are converted 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 been comparatively less studied. This article presents, to our knowledge, the first practical framework for the generation of closed-form upper bounds for *CRs* which (1) is fully automatic, (2) can handle the distinctive features of *CRs*, originating from cost analysis of realistic programming languages, (3) is not restricted to simple complexity classes, and (4) produces reasonably accurate solutions. A key idea in our approach is to view *CRs* as programs, which allows applying semantic-based static analyses and transformations to bound them, namely our method is based on the inference of *ranking functions* and *loop invariants* and on the use of *partial evaluation*.

Keywords Cost analysis · Closed-form upper bounds · Resource analysis · Automatic complexity analysis · Static analysis · Abstract interpretation · Programming languages

E. Albert · P. Arenas (✉) · S. Genaim
DSIC, Complutense University of Madrid (UCM), 28040 Madrid, Spain
e-mail: puri@sip.ucm.es

E. Albert
e-mail: elvira@sip.ucm.es

S. Genaim
e-mail: samir.genaim@fdi.ucm.es

G. Puebla
DLSIIS, Technical University of Madrid (UPM), 28660 Boadilla del Monte, Madrid, Spain
e-mail: german.puebla@upm.es

Published online: 29 April 2010

1 Introduction

Having information about the execution cost of programs, i.e., the amount of resources that the execution will require, is quite useful for many different purposes. Also, reasoning about execution cost is difficult and error-prone. Therefore, it is widely recognized that *cost analysis*, sometimes also referred to as *resource analysis* or *automatic complexity analysis*, is quite important. In this work we are interested in *static cost analysis*, i.e., the analysis results for a program P should allow bounding the cost of executing P on any input data $\overline{x}$ without having to actually *run* $P(\overline{x})$.

The classical approach to static cost analysis consists of two phases. First, given a program and a *cost model*, the analysis produces *cost relations* (*CRs* for short), i.e., a system of recursive equations which capture the cost of the program in terms of the size of its input data. As a simple example, consider the following Java method m which traverses an array v and, depending whether the array elements are odd or even, invokes a different method m2 or m1:

```
public void m(int[ ] v) {
   int i=0;
   for (i=0; i<v.length; i++)
      if (v[i]%2==0) m1();
      else m2();
}
```

The following cost relations capture the cost of executing this program:

$$\begin{array}{llll} (a) & C_m(v) & = k_1 + C_{for}(v,0) & \{v{\geq}0\} \\ (b) & C_{for}(v,i) & = k_2 & \{i{\geq}v,\ v{\geq}0\} \\ (c) & C_{for}(v,i) & = k_3 + C_{m1}() + C_{for}(v,i{+}1) & \{i{<}v,\ v{\geq}0\} \\ (d) & C_{for}(v,i) & = k_4 + C_{m2}() + C_{for}(v,i{+}1) & \{i{<}v,\ v{\geq}0\} \end{array}$$

where v denotes the length of the array v, i stands for the counter of the loop and C_m, C_{m1} and C_{m2} approximate, respectively, the costs of executing the methods m, m1 and m2. The constraints attached to the equations contain their applicability conditions. For instance, equation (a) corresponds to the cost of executing the method m with an array of length greater that 0 (stated in the condition $\{v{\geq}0\}$), where a cost k_1 is accumulated to the cost of executing the loop, given by C_{for}. The constants $k_1, \ldots, k_4$ take different values depending on the cost model that one selects. For instance, if the cost model is the number of executed instructions, then k_1 is 1 which corresponds to the execution of the Java instruction "int i = 0;". If the cost model is the heap consumption, then k_1 is 0 since the previous instruction does not allocate any memory. Equations (c) and (d) capture, respectively, the costs of the then and the else branches. Note that, even if the program is deterministic, they are non-deterministic equations which contain the same applicability conditions. This is due to the fact that the array v is abstracted to its length and hence the values of its elements are unknown statically. Equation (b) captures the cost of exiting the loop.

Some interesting features of cost relations are that: (1) They are programming language independent: there are analyzers for many different languages which produce cost relations. (2) They can cover a wide range of complexity classes: the same techniques can be used to infer cost which is logarithmic, exponential, etc. (3) They can be used for capturing a variety of non-trivial notions of resources, such as heap consumption, number of calls to a specific method, etc.

Though cost relations are simpler than the programs they originate from, since all variables are of integer type, in several respects they are not as static as one would expect from the result of a static analysis. One reason is that they are recursive and thus we may need to iterate for computing their value for concrete input values. Another reason is that even for deterministic programs, it is well known that the loss of precision introduced by the size abstraction may result in cost relations which are non-deterministic. This happens in the above example: since the array v has been abstracted to its length v, the values of v[i] are unknown statically. Hence, the last two equations (c) and (d) become non-deterministic choices. In general, for finding the worst-case cost we may need to compute and compare (infinitely) many results. For both reasons, it is clear that it is interesting to compute *closed-form* upper bounds for the cost relation, whenever this is possible, i.e., upper bounds which are not in recursive form. For instance, for the above example, we aim at inferring the closed-form upper bound $k_1{+}k_2{+}v{*}max(\{k_3 + C_{m1}, k_4 + C_{m2}\})$ where C_{m1} and C_{m2} are in turn closed-form upper bounds for the corresponding methods.

Since cost relations are syntactically quite close to *Recurrence Relations* [15] (*RRs* for short), in most cost analysis frameworks, it has been assumed that cost relations can be easily converted into *RRs*. This has led to the belief that it is possible to use existing *Computer Algebra Systems* (CAS for short) for finding closed-forms in cost analysis. As we will show, cost relations are far from *RRs*. In this article, we present, to the best of our knowledge, the first practical framework for the fully automatic inference of reasonably accurate closed-form upper bounds for *CRs* originating from a wide range of programs. The main novelty of our approach is that, by providing a semantics for *CRs*, we can view *CRs* as programs and, thus, apply semantic-based static analyses and transformations to automatically infer upper bounds for them. In particular, our main contributions are summarized as follows:

– We identify the differences between *CRs* and *RRs*, in Section 2.
– We provide a formal definition of *CRs* and their semantics in terms of *evaluation trees*, in Section 3. These notions are independent of the language and cost model.
– We present a general approximation scheme to infer closed-form upper bounds in Section 4. Basically, it is based on the idea of bounding the cost of the corresponding evaluation trees. This requires computing upper bounds both on the *depth* of trees and also on the *cost* of nodes.
– In Section 5, we propose to use a specific form of ranking functions, which have been extensively studied in termination analysis (see e.g. [45]), to bound the depth of the evaluation tree.
– In Section 6, we present how to bound the cost of nodes by relying on loop invariants [23] and maximization operations.
– In Section 7, we develop an extension of our method to obtain more accurate upper bounds for divide and conquer programs which is based on counting *levels* in the evaluation tree rather than counting nodes.
– Our method can be used when *CRs* are directly recursive. We present in Section 8 an automatic program transformation, formalized in terms of *partial evaluation* (see e.g. [33]), which converts *CRs* into an equivalent directly recursive form.

– We report on a prototype implementation and apply it to obtain closed-form upper bounds for *CRs* automatically generated from Java bytecode programs.

A preliminary version of this work appeared in the Proceedings of SAS'08 [4]. We have pursued cost relations as a language-independent target language for cost analysis in [5]. Our remaining previous work on cost analysis [6, 8, 10, 11] is not related to this article but to the first phase in cost analysis which obtains, from a program and a cost model, a cost relation.

1.1 Applications of Upper Bounds of Cost Relations

Automatic cost analysis requires the inference of closed-form upper bounds in order to be used within its large application field, which includes the following applications:

Resource Bound Certification This research area deals with security properties involving resource usage requirements; i.e., the code must adhere to specific bounds on its resource consumption. The present work enables the automatic generation of non-trivial closed-form upper bounds on cost. Such upper bounds can be computed by a trusted server who signs the code using public key infrastructure. Alternatively, they can be computed from scratch on the client side or (hopefully) efficiently checked by using *certificates*, in the proof-carrying code [43] style, though the latter would require further research. Previous work in resource bound certification was restricted to *linear bounds* [12, 25, 31] and to *semi-automatic techniques* [21].

Performance Debugging and Validation This application is based on automating the process of checking whether certain *assertions* about the efficiency of the program, possibly written by the programmer, hold or not. This application was already mentioned as future work in [54] and is available in the CiaoPP system for Prolog programs [29]. Our closed-form upper bounds can be used to check whether the overall cost of an application meets the resource-consumption constraints specified in the assertions.

Program Synthesis and Optimization This application was already mentioned as one of the motivations for [54]. Both in program synthesis and in semantic-preserving optimizations, such as partial evaluation (see e.g. [24, 46]), there are multiple programs which may be produced in the process, with possibly different efficiency levels. Here, upper bounds on the cost can be used for guiding the selection process among a set of candidates.

2 Cost Relations vs. Recurrence Relations

The aim of this section is to identify the differences between cost relations and traditional recurrence relations. For this purpose, we take a close look at the *CRs* which appear in cost analysis of real programs. Figure 1 shows a Java program which we use as running example. We explain in detail, in Section 2.1 below, the *CRs* produced for this program by the automatic cost analysis of [6]. Then, in Section 2.2 we discuss the differences with *RRs*.

```
static void del(List l, int p, int a[], int la, int b[], int lb){
   while (l!=null){                            // cost equations (2),(3),(4)
      if (l.data < p)   la=rm_vec(l.data,a,la);
      else              lb=rm_vec(l.data,b,lb);
      l=l.next;
   }
}
static int rm_vec(int e, int a[], int la){
   int i=0;
   while (i < la && a[i]<e) {i++;};    // cost equations (5),(6),(7)
   for (int j=i; j<la-1; j++) a[j]=a[j+1]; // cost equations (8),(9)
   return la-1;
}
```

Fig. 1 Java code of running example

2.1 Cost Relations for the Running Example

Consider the Java code in Fig. 1. It uses a List class for (non sorted) linked lists of integers which is implemented in the usual way. The del method receives as input: l, a list without repetitions; p, an integer value (the *pivot*); a and b, two sorted arrays of integers; and la and lb, two integers which indicate, respectively, the number of positions occupied in a and b. The a (resp. b) array is expected to contain values which are smaller (resp. greater or equal) than p, the pivot. 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 rm_vec auxiliary method removes a given value e from an array a of length la and returns a's new length, la−1.

Example 1 The system Costa [7] is an abstract interpretation-based COSt and Termination Analyzer for Java bytecode. It receives as input a bytecode program and (a choice of) a resource of interest in the form of a cost model, and tries to obtain an upper bound of the resource consumption of the program. In Fig. 2, we show the control flow graphs (CFG) constructed by Costa in order to generate automatically the *CRs*. Such CFGs correspond to the graphs for the two methods (del and rm_vec) and separate CFGs for the loops, as in Costa loop extraction is performed mainly for efficiency issues (see [2]). Although [6] analyzes Java bytecode and not Java source, we show the source for clarity of the presentation.

Figure 3 shows the *CRs* automatically generated by the system for the del method in Fig. 1 using the CFGs in Fig. 2. The syntax and semantics of *CRs* is explained in detail in Section 3. Briefly, cost relations are defined by means of equations, each of which has an associated set of constraints which is shown to the right of the equation. Intuitively, the *CRs* are obtained from the program after performing the following three main steps:

1. In the first step, the recursive structure of the cost relation is determined by observing the iterative constructs in the program. In the case of imperative programs, both loops and recursion produce recursive calls in the cost relation. The *CR* matches the structure of the program such that when the program contains an iterative construct, its *CR* has a recursion. To carry out this step,

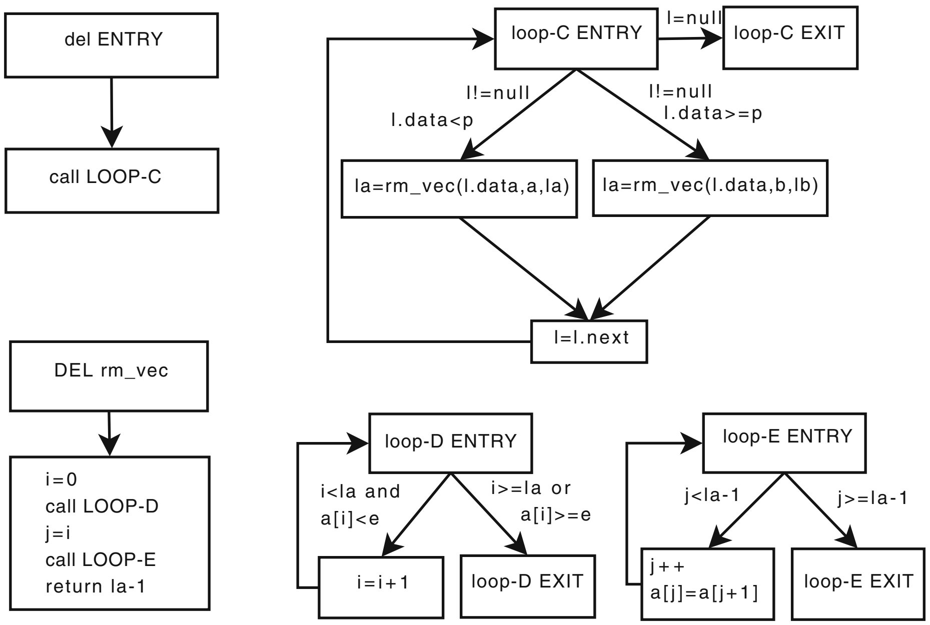


Fig. 2 Control flow graphs for running example

analyzers usually build CFGs. In our example, we have three recursive cost relations C, D and E which correspond to the three CFGs for the loops in Fig. 2:

- C : cost of the while loop in del,
- D : cost of the while loop in rm_vec,
- E : cost of the for loop in rm_vec.

For readability, the *CRs* in Fig. 3 are shown after performing *partial evaluation*, as we will explain in Section 8. This explains why there is no relation for the method rm_vec: the calls to rm_vec have been unfolded within its calling context, i.e., they have been replaced by the right hand side of the corresponding equation.

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

Fig. 3 Cost relations generated by cost analysis of running example

2. In the second step, static analysis techniques are used in order to approximate how the *sizes* of variables change from one call in the cost relation to another. Each program variable is abstracted using a *size measure* such that every non-integer value is represented as a natural number. Classical size measures used for non-integer types are: array length for arrays, list length for lists, the length of the longest reference path for linked data structures, etc. In the above example, l represents the *path-length* [51] of the corresponding dynamic structure, which in this case coincides with the length of the list; a and b are the lengths of the corresponding arrays. Since la and lb are numeric (integer) variables, the *CR* directly handles those values, i.e., no abstraction is required for them. Analysis is often done by obtaining an *abstract* version of the program by relying on abstract interpretation [22]. Essentially, the abstraction consists in inferring *size constraints*, sometimes also referred to as *size relations*, between the program variables at different program points. In Fig. 3, such size relations are shown to the right of the equations. They are usually expressed by means of linear constraints. We refer to such abstraction by *size abstraction* and to an analysis that infers such relations by *size analysis*.
3. In the last step, instructions in the original program are replaced by the cost they represent. In the running example, we count the number of bytecode instructions executed such that each Java instruction corresponds to several bytecodes. It is not a concern of this paper to understand how bytecode instructions are related to Java statements. Hence, we omit explanations about the inferred constants in the equations.

After applying the above steps, the analyzer can set up the *CRs* shown in Fig. 3 which we explain below. Equation (1) defines the cost of method del as 1 bytecode instruction plus the cost of the call to C. Observe also that the set of constraints contain applicability conditions (i.e., *guards*) for each equation, if any, by providing constraints which only affect a subset of the variables in the left hand side (lhs for short). For clarity, we have inlined equality constraints (e.g., inlining equality $lb' = lb - 1$ is done by replacing all occurrences of lb' by $lb - 1$). The constraints attached to (1) are the (abstract) preconditions of the program. Among them, we have $a \geq la$ (resp. $b \geq lb$), which requires that the number of elements occupied in each array is less or equal than its length. Such preconditions are propagated properly to the rest of the equations.

In addition to *Del*, we have three recursive relations. As regards E, (8) is its base case and it corresponds to the exit from the for loop, whereas (9) counts the cost of each iteration in the loop. As expected, the value of j is increased by one at the recursive call to E. As regards the cost relation D, we have two base cases, (5) and (6), which correspond to the exits from the loop because i $\geq$ la and because a[i] $\geq$ e, respectively. The important point here is that the second condition does not appear in the constraints of (6) because this condition is not observable after abstracting the array a to its length, i.e., the value in a[i] is *unknown*. For the selected cost model, we count three bytecode instructions in the first base case and eight in the second one. The cost of executing an iteration of the loop is captured by (7), where the condition $i < la$ must be satisfied and variable i is increased by one at each recursive call.

Finally, in relation C, (2) corresponds to the case of an empty list, indicated by the condition $l = 0$. Equations (3) and (4) correspond, respectively, to the then and else branches of the if-then-else construct within the while loop. Hence, both of them contain the relation $l > 0$. Note that, as before, the conditions l.data < p and l.data ≥ p in the Java program do not appear in the constraints attached to (3) and (4) as they are not preserved by the corresponding size abstraction. The calls to D and E in (3) capture the cost of executing the method rm_vec for a and la. In the constraints, la decreases by one upon exit from rm_vec. l' corresponds to the length of the list when we perform the recursive call. It is ensured that the size of l has decreased ($l > l'$), but due to the size abstraction, we do not know how much. This is because the size analysis for heap allocated data structures used in [6] is based on path-length analysis, where size relations are expressed using $>$ and $\geq$ only. Equation (4) is similar to (3) but for b and lb instead of a and la. Note that when calling E in (3) and (4), a fresh variable j is used since we do not know the value that j can take after executing the while loop. We only know that $j \geq 0$, as it appears in the attached constraint.

Importantly, if the program were written in a different programming language, the first phase in cost analysis would produce a similar cost relation which differs essentially only on intermediate equations and on the constants which are counted. This step is outside the scope of this article (see Section 11 for references to this phase in several programming languages). Our approach for computing closed-form upper bounds takes as input cost relations which originate from programs written in any programming language.

2.2 Why Cost Relations Are Not Recurrence Relations?

As can be seen in the *CRs* in the example, *CRs* differ from standard *RRs* [15] in the following ways:

(a) *Non-determinism* In contrast to *RRs*, *CRs* are highly 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 value. In Example 1, this happens between (3) and (4) and also between (6) and (7).

(b) *Inexact constraints* *CRs* may have constraints other than equalities, such as $l > l'$. 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 Example 1 in (3) and (4).

(c) *Multiple arguments* *CRs* usually depend on several arguments that may increase (variable i in (7)) or decrease (variable l in (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) is an obvious source of non-determinism and it was already detected in [54]. Point (b) is another source of non-determinism. Though it may not be so evident in small examples, it is almost unavoidable in programs handling trees or when numeric value analysis loses precision. As a result of (a) and (b), strictly speaking, *CRs* do not define functions, but rather relations: given a relation C and input values $\bar{v}$, there may exist multiple output values for $C(\bar{v})$.

As regards point (c), most existing solvers can only handle single-argument recurrences (Mathematica is an exception). Sometimes it is possible to automatically convert relations with several arguments into relations with only one. However, this approach is only applicable when the equations, in addition to the recursive calls themselves, only have constant value expression in the right hand side (rhs for short). This problem is illustrated in Fig. 4. There, relation C has two arguments, but it can be converted into relation C' which only has one argument by defining $z = y - x$, resulting in equations $1'$ and $2'$, with the adapter equation 5. Now, if we try to apply the same transformation to relation D, the situation is different. The reason for this is that (4) accumulates the non-constant expression x in each iteration. Now, the transformation results in ($4'$), where the value of y is unbounded and thus an upper bound cannot be found. Note that a fundamental difference between C and D is that while the former only depends on $y - x$ the latter takes different values depending on the initial value of x. E.g., $C(0, 10) = C(1000, 1010)$ but $D(0, 10) \neq D(1000, 1010)$.

The above differences make existing methods for solving *RRs* 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.

Given a (non-deterministic) cost relation, it is sometimes useful to define a cost *function*. A relatively straightforward way of obtaining a cost function from non-deterministic *CRs* would be to introduce a maximization operator. Unfortunately, the cost functions thus produced are not very useful since existing CAS do not support the maximization operator. 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. This combinatorial explosion also affects the use of such cost-bound function in dynamic approaches, i.e., those based on executing cost-bound functions, such as [28].

Another approach is to obtain a cost-bound function by eliminating non-determinism. For this, we need to remove equations from *CRs* as well as sometimes to replace inexact constraints by exact ones while preserving the worst-case solution. However, this is not possible in general. E.g., in Fig. 3, the maximum cost is obtained when the execution interleaves (3) and (4), and therefore the worst case cannot be achieved if we remove either equation. In other words, the upper bound obtained by removing either of (3) and (4) is not an upper bound of the original *CR*.

Finally, let us observe that the properties listed above are all evident properties of constraint programs whose arguments are integer values. This explains the fact

Fig. 4 Replacing multiple arguments with a single one

$$\begin{array}{lll}
(1) & C(x,y) = 2 & \{x \geq y\} \\
(2) & C(x,y) = 3 + C(x',y') & \{x < y',\ x' = x-1,\ y' = y\} \\
(3) & D(x,y) = 2 & \{x \geq y\} \\
(4) & D(x,y) = x + D(x',y') & \{x < y',\ x' = x-1,\ y' = y,\ x \geq 0\} \\
(5) & C(x,y) = C'(y-x) & \\
(1') & C'(z) = 2 & \{z \leq 0\} \\
(2') & C'(z) = 3 + C'(z') & \{z > 0,\ z' = z-1\} \\
(4') & D'(z) = (y-z) + D'(x',y') & \{x < y',\ x' = x-1,\ y' = y,\ y \geq z\}
\end{array}$$

that we treat *CR* as programs and apply analysis and transformations developed for programming languages on them.

3 Cost Relations: Syntax and Semantics

Let us introduce some notation and preliminary definitions. The sets of natural, integer and real values are denoted respectively by $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$. The sets of non-negative integer and real values are denoted respectively by $\mathbb{Z}_+$ and $\mathbb{R}_+$. We use v and w for values from $\mathbb{Z}$ and $\mathbb{Z}_+$, r for values from $\mathbb{R}$ and $\mathbb{R}_+$, and n for values from $\mathbb{N}$. We write x, y, and z, to denote variables which range over $\mathbb{Z}$. Given any entity t, $vars(t)$ refers to the set of variables occurring in t. The notation $\bar{t}$ stands for a sequence of entities $t_1, \ldots, t_n$, for some $n > 0$. For simplicity, we sometimes interpret these sequences as sets. We use $t[\bar{y}/\bar{x}]$ to denote the renaming of the variables $\bar{x}$ by $\bar{y}$.

A *linear expression* has the form $v_0 + v_1x_1 + \cdots + v_nx_n$. A *linear constraint c* (over $\mathbb{Z}$) has the form $l_1 \leq l_2$ where l_1 and l_2 are linear expressions. For simplicity, we write $l_1 = l_2$ instead of $l_1 \leq l_2 \wedge l_2 \leq l_1$, and $l_1 < l_2$ instead of $l_1 + 1 \leq l_2$. Note that constraints with rational coefficients can be always transformed to equivalent constraints with integer coefficients, e.g., $\frac{1}{2}x > y$ is equivalent to $x > 2y$. We write φ, ψ or ϕ, possibly subscripted, to denote sets of linear constraints, i.e., of the form $\{c_1, \ldots, c_n\}$, which should be interpreted as the conjunction $c_1 \wedge \cdots \wedge c_n$. We write $\bar{x} = \bar{y}$ to denote $x_1 = y_1 \wedge \cdots \wedge x_n = y_n$ and $\varphi_1 \models \varphi_2$ to indicate that the (set of) linear constraints φ_1 implies the (set of) linear constraints φ_2. An assignment σ over a tuple of variables $\bar{x}$ is a mapping from $\bar{x}$ to $\mathbb{Z}$. Sometimes we denote an assignment over $\bar{x}$ as $\bar{x} = \bar{v}$, therefore we might write $\sigma \models \varphi$ for $\bar{x} = \bar{v} \models \varphi$. We use $\sigma(x)$ to refer to the value of x in σ, and $\sigma(\bar{x})$ for $\langle \sigma(x_1), \ldots, \sigma(x_n) \rangle$. The projection operator $\exists \bar{x}.\varphi$ (resp. $\bar{\exists} \bar{x}.\varphi$) projects the polyhedron defined by φ on the space $vars(\varphi) \setminus \bar{x}$ (resp. $\bar{x}$).

The following definition presents our notion of *basic cost expression*, which characterizes syntactically the kind of expressions we deal with. Such expressions will be crucial to characterize the cost relation systems defined in the next section.

Definition 1 (basic cost expression) A symbolic expression exp is a *basic cost expression* if it can be generated using the grammar below:

$$\mathsf{exp} ::= r \mid \mathsf{nat}(l) \mid \mathsf{exp} + \mathsf{exp} \mid \mathsf{exp} * \mathsf{exp} \mid \mathsf{exp}^r \mid \log_n(\mathsf{exp}) \mid n^{\mathsf{exp}} \mid \max(S) \mid \mathsf{exp} - r$$

where $r \in \mathbb{R}_+$, l is a linear expression, S is a non empty set of basic cost expressions, $\mathsf{nat} : \mathbb{Z} \rightarrow \mathbb{Z}_+$ is defined as $\mathsf{nat}(v) = \max(\{v, 0\})$, and exp satisfies that for any assignment $\sigma : vars(\mathsf{exp}) \mapsto \mathbb{Z}$ we have that $[\![\mathsf{exp}]\!]_\sigma \in \mathbb{R}_+$, where $[\![\mathsf{exp}]\!]_\sigma$ is the result of evaluating exp w.r.t. σ.

Basic cost expressions are symbolic expressions which represent the resources we accumulate and are the non-recursive building blocks for defining cost relations and for the closed-form upper bounds that we infer for them. Cost expressions enjoy two crucial properties: (1) By definition, they are always evaluated to non-negative values, for instance, the expression $\mathsf{nat}(x) - 1$ is not a cost expression, since its evaluated to negative numbers for $x \leq 0$, however, $\mathsf{nat}(x - 1)$ is a valid cost expression. Note that the $-r$ expression has been introduced to the above grammar only for being able of constructing $n^{\mathsf{nat}(l)}-1$ (when counting the number of nodes of a tree), which is clearly evaluated to a non-negative value. (2) They are monotonic in their nat components, i.e., 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. This is essential for defining the maximization procedure *ub_exp*, which is defined in Section 6.2.

Proposition 1 *Let* $\mathtt{exp}$ *be a basic cost expression,* l *and* l' *be linear expressions and* φ *be a set of linear constraints such that* $\varphi \models l' \geq l$*. Let* $\mathtt{exp}'$ *be the result of replacing an occurrence of* $\mathsf{nat}(l)$ *in* $\mathtt{exp}$ *by* $\mathsf{nat}(l')$*. Then for any assignment* σ *for* $vars(\mathtt{exp}') \cup vars(\mathtt{exp})$*, if* $\sigma \models \varphi$ *then* $[\![\mathtt{exp}']\!]_\sigma \geq [\![\mathtt{exp}]\!]_\sigma$.

Proof By structural induction on basic cost expressions: (1) for expressions of the form $\mathsf{nat}(l)$ the result follows from $\sigma \models \varphi$ and $\varphi \models l' \geq l$, which implies $[\![l']\!]_\sigma \geq [\![l]\!]_\sigma$; and (2) for the induction step, composing expressions as described in Definition 1 preserves trivially the monotonicity property. □

Definition 2 (Cost Relation System) A *cost relation system* $\mathcal{S}$ is a finite set of equations of the form $\langle C(\bar{x}) = \mathtt{exp} + \sum_{i=1}^{k} D_i(\bar{y}_i), \varphi\rangle$ with $k \geq 0$, where C and all D_i are cost relation symbols, all variables $\bar{x} \cup \bar{y}_i$ are distinct variables; $\mathtt{exp}$ is a basic cost expression; and φ is a set of linear constraints over $\bar{x} \cup vars(\mathtt{exp}) \bigcup_{i=1}^{k} \bar{y}_i$.

In contrast to standard definitions of *RRs*, in *CRSs*, the variables which occur in the rhs of the equations do not need to be related to those in the left hand side (lhs for short) by equality constraints. Other constraints such as $\leq$ and $<$ can also be used. We denote by *rel*$(\mathcal{S})$ the set of cost relation symbols which are defined in $\mathcal{S}$, i.e., which appear in the lhs of some equation in $\mathcal{S}$. Given a *CRS* $\mathcal{S}$ and a cost relation symbol C, the definition of C in $\mathcal{S}$, denoted *def*$(\mathcal{S}, C)$, is the subset of the equations in $\mathcal{S}$ whose lhs is of the form $C(\bar{x})$. Without loss of generality, we assume that all equations in *def*$(\mathcal{S}, C)$ have the same variable names in the lhs, and that $\mathcal{S}$ is self-contained in the sense that all cost relation symbols which appear in the rhs of an equation in $\mathcal{S}$ must be in *rel*$(\mathcal{S})$.

A cost equation $\langle C(\bar{x}) = \mathtt{exp} + \sum_{i=1}^{k} D_i(\bar{y}_i), \varphi\rangle$ states that the cost of $C(\bar{x})$ is $\mathtt{exp}$ plus the sum of the cost of all $D_i(\bar{y}_i)$ where the linear constraints φ contain the applicability conditions for the equation as well as size relations for the equation variables. Intuitively, a cost relation is program, very similar to a constraint logic program [32] where the relation plays the role of a predicate and an equation plays the role of a clause. Evaluating a call $C(\bar{v})$ can be done as follows: (1) choose a matching equation $\mathcal{E} \equiv \langle C(\bar{x}) = \mathtt{exp} + \sum_{i=1}^{k} D_i(\bar{y}_i), \varphi\rangle$; (2) choose an assignment σ over *vars*$(\mathcal{E})$ s.t. $\sigma \models \bar{v} = \bar{x} \wedge \varphi$; (3) evaluate $\mathtt{exp}$ w.r.t σ and accumulate it to the result; and (4) evaluate each call $D_i(\bar{v}_i)$ where $\bar{v}_i = \sigma(\bar{y}_i)$. Note that the result (i.e., the cost of the execution) of the evaluation is the sum of all cost expressions accumulated in step (3). Such evaluation strategy can be described in terms of *evaluation trees*. Each node in the tree describes the cost accumulated at step (3), and the n sub-trees correspond to the evaluation of the calls in step (4). Then, the result of the evaluation corresponds to the sum of all nodes in the tree.

The next definition provides a formal (denotational) semantics for *CRSs* which maps a call $C(\bar{v})$ to the set of all possible evaluation trees, and therefore the set of all possible answers. 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 Given a cost relation system $\mathcal{S}$, the set of evaluation trees induced by an initial query $C(\bar{v})$ is defined as:

$$
\begin{aligned}
&\mathit{Trees}(C(\bar{v}), \mathcal{S})\\
&= \left\{ \mathit{node}(C(\bar{v}), r, \langle T_1, \ldots, T_k\rangle) \;\middle|\; \begin{array}{l} 1.\ \mathcal{E}{\equiv}\langle C(\bar{x}){=}\mathtt{exp}{+}\sum_{i=1}^{k} D_i(\bar{y}_i), \varphi\rangle{\in}\mathcal{S} \\ 2.\ \sigma \text{ is an assignment over } \mathit{vars}(\mathcal{E}) \text{ s.t.} \\ \quad \sigma{\models}\bar{x}{=}\bar{v}{\wedge}\varphi \\ 3.\ r = [\![\mathtt{exp}]\!]_\sigma \\ 4.\ T_i \in \mathit{Trees}(D_i(\bar{v}_i), S) \text{ s.t } \bar{v}_i = \sigma(\bar{y}_i) \end{array} \right\}
\end{aligned}
$$

Then, the set of all possible answers for $C(\bar{v})$ is defined as:

$$\mathit{Answers}(C(\bar{v}), \mathcal{S}){=}\{\mathsf{Sum}(T) \mid T \in \mathit{Trees}(C(\bar{v}), \mathcal{S})\}$$

where $\mathsf{Sum}(T) = \mathsf{Sum}(\mathit{node}(C(\bar{v}), r, \langle T_1, \ldots, T_k\rangle)) = r + \sum_{i=1}^{k} \mathsf{Sum}(T_i)$.

A cost-bound function $C_+(\bar{x})$ can be defined as $C_+(\bar{v}) = \max(\mathit{Answers}(C(\bar{v}), \mathcal{S}))$. Clearly, it is not always computable. Sometimes there is actually no upper bound because the tree is infinite. Also, it can happen that an upper bound exists but it is not computable. Note that the branching in each tree is conjunctive and corresponds to the different calls in the body, an that the disjunction comes in the form of multiple trees for the same query.

Example 2 Figure 5 shows two possible evaluation trees for *Del*(3, 10, 2, 20, 2) in $\mathcal{S}$, where $\mathcal{S}$ is the *CR* in Fig. 3. The tree on the left has maximal cost, whereas the one on the right has minimal cost. Nodes are represented using boxes split in two parts. The part on the left contains a call, e.g., *Del*(3, 10, 2, 20, 2) in the root nodes of both trees, annotated with a number in parenthesis, e.g., (1) in such nodes, which indicates the equation which was selected for evaluating such call. The part on the right contains the local cost associated to the call, 1 in both root nodes. Nodes are linked by arrows to their children, if any.

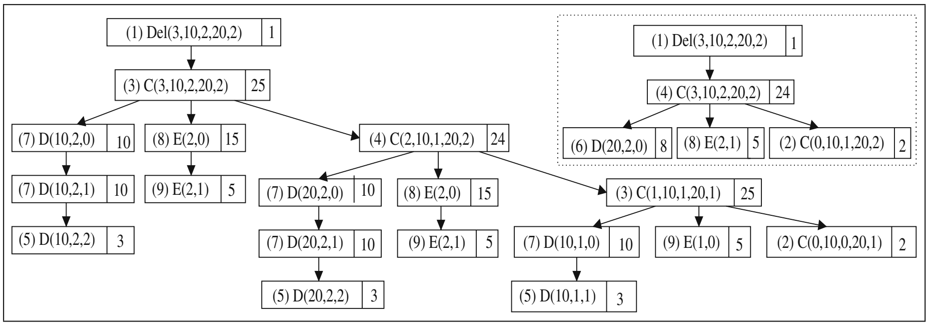


Fig. 5 Two evaluation trees for *Del*(3, 10, 2, 20, 2)

The two trees differ in that, for solving $C(3, 10, 2, 20, 2)$, in the one on the left we pick (3) and in the one on the right we pick (4). Furthermore, in the recursive call to C in (3) and (4) we always assign $l' = l - 1$ in the tree on the left and we assign $l' = l - 3$ in the tree on the right. Note that both possibilities are valid w.r.t. $\mathcal{S}$, since we are allowed to pick any value l' such that $l' < l$. The tree on the left corresponds to a possible execution of the program. However, the tree on the right does not correspond to any actual execution. This is a side effect of using safe approximations in static analysis for computing size abstractions: information is correct in the sense that given a concrete program execution, at least one of the evaluation trees must correspond to such execution, but there may be other trees which do not correspond to any valid execution. Therefore, *CRSs* provide information which is sound but possibly imprecise.

As this example shows, there may be multiple evaluation trees for a call. In fact, there may even be infinitely many of them. The latter happens in our example call, as step 1 in Definition 3 can provide an infinite number of assignments to variable j which are compatible with the constraint $j \geq 0$ in (3) and (4). This shows that approaches like [28] based on evaluation of *RRs* may not be of general applicability in *CRSs*, as size relations can be inexact and multiple, or even infinitely many evaluation trees may exist. Fortunately, since we are not interested in executing *CRSs* but rather on finding closed-form (i.e., static) upper bounds for them, whether there are infinitely many evaluation trees for a call is not directly an issue, as long as there are not infinitely many *different* answers. In our example, $Trees(Del(3, 10, 2, 20, 2), \mathcal{S})$ is an infinite set, but infinitely many of the trees in this set produce equivalent results and $Answers(Del(3, 10, 2, 20, 2)), \mathcal{S})$ is finite. Thus, it is in principle possible to find an upper bound for it.

4 Closed-Form Upper-Bounds for Cost Relations

After providing a suitable semantics for *CRs*, we now study how to obtain closed-form upper bounds for them. In what follows, we are only interested in upper-bound functions which are in closed-form. Therefore, for brevity, we often just write ‘upper bound’ instead of ‘closed-form upper bound’.

A function $f : \mathbb{Z}^n \mapsto \mathbb{R}_+$ is in *closed-form* if it is defined as $f(\bar{x}) = \texttt{exp}$, where $\texttt{exp}$ is a basic cost expression and $vars(\texttt{exp}) \subseteq \bar{x}$. Let C be a cost relation, a closed-form function $U : \mathbb{Z}^n \mapsto \mathbb{R}_+$ is an *upper bound* of C if $\forall \bar{v} \in \mathbb{Z}^n$ and $\forall r \in Answers(C(\bar{v}), \mathcal{S})$ it holds that $U(\bar{v}) \geq r$. Similarly, we say that a function $f : \mathbb{Z}^n \mapsto \mathbb{Z}$ is an upper bound for $g : \mathbb{Z}^n \mapsto \mathbb{Z}$, if $f(\bar{v}) \geq g(\bar{v})$ for any $\bar{v} \in \mathbb{Z}^n$. Given a relation C (resp. function f), we use C_+ (resp. f_+) to refer to an upper bound of C (resp. f).

4.1 Stand-alone Cost Relations

An important feature of *CRSs*, also present in *RRs*, is their *compositionality*. This allows computing upper bounds of *CRSs* composed of multiple relations by concentrating on one relation at a time. Let us consider an equation $\mathcal{E}$ for a cost relation $C(\bar{x})$ where a call of the form $D(\bar{y})$, with $D \neq C$ appears on the rhs of $\mathcal{E}$. In order to compute an upper bound of $C(\bar{x})$, we can replace $\mathcal{E}$ by another equation $\mathcal{E}'$ where

$$
\begin{array}{l}
(2)\ C(l,a,la,b,lb) = \boxed{2} \\
\quad \{a \geq la, b \geq lb, b \geq 0, a \geq 0, l = 0\} \\
(3)\ C(l,a,la,b,lb) = \\
\quad \boxed{\text{38+15*nat(la-j-1)+10*nat(la)}} + C(l',a,la-1,b,lb) \\
\quad \{a \geq 0, a \geq la, b \geq lb, j \geq 0, b \geq 0, l > l', l > 0\} \\
(4)\ C(l,a,la,b,lb) = \\
\quad \boxed{\text{37+15*nat(lb-j-1)+10*nat(lb)}} + C(l',a,la,b,lb-1) \\
\quad \{b \geq 0, b \geq lb, a \geq la, j \geq 0, a \geq 0, l > l', l > 0\}
\end{array}
$$

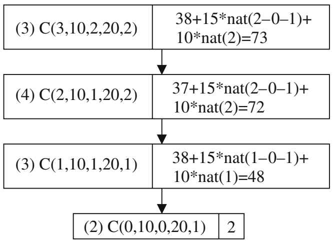

(3) C(3,10,2,20,2)
38+15*nat(2–0–1)+ 10*nat(2)=73
(4) C(2,10,1,20,2)
37+15*nat(2–0–1)+ 10*nat(2)=72
(3) C(1,10,1,20,1)
38+15*nat(1–0–1)+ 10*nat(1)=48
(2) C(0,10,0,20,1)
2

Fig. 6 Stand-alone *CR* for relation *C* and a corresponding evaluation tree

the call to $D(\bar{y})$ is replaced by a call to an upper bound $D_+(\bar{y})$, already in closed-form. The resulting cost relation is trivially an upper bound of the original one. E.g., suppose that we have the following upper bounds:

$$
\begin{array}{ll}
E_+(la, j) & = 5 + 15 * \mathsf{nat}(la - j - 1) \\
D_+(a, la, i) & = 8 + 10 * \mathsf{nat}(la - i)
\end{array}
$$

Replacing the calls to D and E in (3) and (4) by D_+ and E_+ results in the *CR* shown in Fig. 6.

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 *stand-alone 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 stand-alone cost relations and, in Section 8, we provide a mechanism for obtaining direct recursion automatically.

4.2 Approximating Evaluation Trees

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. 6 for the stand-alone 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, *leaf*(T) denotes the set of leaves of T (i.e., those without children) and $internal(T)$ denotes the set of internal nodes (all nodes but the leaves) of T.

Proposition 2 *(node-count upper bound) Let C be a cost relation. We define:*

$$C_+(\bar{x}) = internal_+(\bar{x}) * costr_+(\bar{x}) + leaf_+(\bar{x}) * costnr_+(\bar{x})$$

where $internal_+(\bar{x})$, $costr_+(\bar{x})$, $leaf_+(\bar{x})$ and $costnr_+(\bar{x})$ are closed-form functions defined on $\mathbb{Z}^n \mapsto \mathbb{R}_+$. Then, C_+ is an upper bound of C if for all $\bar{v} \in \mathbb{Z}^n$ and for all $T \in Trees(C(\bar{v}), \mathcal{S})$, the following properties hold:

1. *$internal_+(\bar{v}) \geq |internal(T)|$ and $leaf_+(\bar{v}) \geq |leaf(T)|$;*
2. *$costr_+(\bar{v})$ is an upper bound of $\{r \mid node(\_, r, \_) \in internal(T)\}$ and*
3. *$costnr_+(\bar{v})$ is an upper bound of $\{r \mid node(\_, r, \_) \in leaf(T)\}$.*

Proof Trivially correct by the definition of upper bound and *Answers*. □

This proposition presents the main approximation approach which we use for computing upper bounds. Our main contribution is to come up with mechanisms to infer the four functions appearing above.

5 Upper Bounds on the Number of Nodes

In this section, we present an automatic mechanism for obtaining correct $internal_+(\bar{x})$ and $leaf_+(\bar{x})$ functions which *statically* provides upper bounds of the number of internal nodes and leaves in evaluation trees. The basic idea is to first obtain upper bounds on the *branching factor* (denoted b) and *height* (the distance from the root to the deepest leaf) of all corresponding evaluation trees (denoted $h_+(\bar{x})$) and, then, use the number of internal nodes and leaves of a *complete* tree with such branching factor and height as an upper bound. Well-known formulas exist which, given the branching factor and the height of the tree, compute the number of nodes of the *complete* tree. As usual, a tree is complete when all internal nodes have as many children as indicated by the branching factor and leaves are at the same depth. Clearly, complete trees provide an upper bound of the number of nodes of any tree with such height and branching factor. Therefore, we define $internal_+(\bar{x})$ and $leaf_+(\bar{x})$ as follows:

$$leaf_+(\bar{x}) = b^{h_+(\bar{x})} \qquad internal_+(\bar{x}) = \begin{cases} h_+(\bar{x}) & b = 1 \\ \frac{b^{h_+(\bar{x})}\text{-}1}{b\text{-}1} & b \geq 2 \end{cases}$$

For a cost relation C, the branching factor b in any evaluation tree for a call $C(\bar{v})$ is limited by the maximum number of recursive calls which occur in a single equation for C, which obviously can be computed statically. Note that we mean the actual occurrences of recursive calls in the right hand side of the equations which determines the complexity scheme (exponential, polynomial, etc.) not how many calls will actually be performed in a concrete execution. This is not related to how the arguments increase or decrease.

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

Definition 4 (loops) Let $\mathcal{E} = \langle C(\bar{x}) = \mathtt{exp} + \sum_{i=1}^{k} C(\bar{y}_i), \varphi \rangle$ be an equation for a cost relation C. The set of *loops* induced by $\mathcal{E}$ is defined as:

$$\mathit{Loops}(\mathcal{E}) = \{\langle C(\bar{x}) \rightarrow C(\bar{y}_i), \varphi' \rangle \mid \varphi' = \bar{\exists} \bar{x} \cup \bar{y}_i.\varphi,\ 1 \leq i \leq k\}$$

Similarly, we define $\mathit{Loops}(C) = \cup_{\mathcal{E} \in \mathit{def}(\mathcal{S},C)} \mathit{Loops}(\mathcal{E})$.

Intuitively, a loop $\langle C(\bar{x}) \rightarrow C(\bar{y}), \varphi' \rangle$ over-approximates that evaluating $C(\bar{v}_1)$ such that $\bar{x} = \bar{v}_1 \models \varphi'$, may eventually be followed by an evaluation for $C(\bar{v}_2)$ such that $\bar{x} = \bar{v}_1 \wedge \bar{y} = \bar{v}_2 \models \varphi'$. In terms of evaluation trees, this means that the node corresponding to $C(\bar{v}_1)$ will have a child with $C(\bar{v}_2)$.

Example 3 The cost relation in Fig. 6 induces the following two loops which correspond to (3) and (4).

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

The problem of bounding the number of consecutive recursive calls has been extensively studied in the context of termination analysis. Automatic termination analyzers usually prove that an upper bound of the number of iterations of the loop exists by proving that there exists a function f from the loop's arguments to a *well-founded* partial order, such that f decreases in any two consecutive calls. This in turn guarantees the absence of infinite traces, and therefore termination. These functions are usually called *ranking functions* [27]. A difference w.r.t. termination analysis is that we aim at determining a concrete ranking function f, rather than just proving that it exists, which is usually enough for termination proofs. The following definition characterizes the kind of ranking functions we are interested in since, as we will see later, they are adequate for bounding the number of iterations of a loop.

Definition 5 (ranking function for a loop) 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 $\varphi \models f(\bar{x}) > f(\bar{y})$.

The above definition basically requires that f be decreasing in every iteration of the loop, and well-founded since the range of f is $\mathbb{Z}_+$. In order to satisfy these conditions it is required that: (1) the constraint φ captures information about the way in which the value of variables change from one iteration to another; and (2) φ captures sufficient information about the applicability conditions (guards) of the loop so as to identify cases where the loop does not apply.

In addition, since a cost relation may induce several loops (i.e., several possibilities for generating calls), we require the *ranking function* to decrease for all loops.

Definition 6 (ranking function for a cost relation) A function $f_C : \mathbb{Z}^n \mapsto \mathbb{Z}_+$ is a *ranking function* for C if it is a ranking function for all loops in $Loops(C)$.

Example 4 The function $f_C(l, a, la, b, lb) = \mathsf{nat}(l)$ is a ranking function for C in the cost relation in Fig. 6. Note that φ'_1 and φ'_2 in the loops of C in Example 3 contain the constraints $\{l > l', l > 0\}$ which is enough to guarantee that f_C is decreasing and well-founded.

The following example illustrates that sometimes the ranking function involves several arguments.

Example 5 Consider the loop which originates from (7) depicted in Fig. 3. $\langle D(a, la, i) \rightarrow D(a, la, i'), \{i' = i + 1, i < la, a \geq la, i \geq 0\}\rangle$. The function $f_D(a, la, i) = \mathsf{nat}(la - i)$ is a ranking function for the above loop. Any ranking function for D must involve both la and i.

We propose to use ranking functions for cost relations as an upper bound on the number of consecutive calls (and therefore on the height of the corresponding evaluation trees). This is justified by the following two facts: (1) the ranking function decreases at least by one unit in each iteration when applying it on two consecutive calls (since its range is $\mathbb{Z}_+$); and (2) it is always non-negative.

Lemma 1 *Let $f_C(\bar{x})$ be a ranking function for a cost relation C. Then, $\forall \bar{v} \in \mathbb{Z}^n$ and $\forall\, T \in Trees(C(\bar{v}), \mathcal{S})$ it holds $f_C(\bar{v}) \geq h(T)$.*

Proof For $h(T) = 0$, the proof is straightforward as $f_C(\bar{v})$ is non-negative. For $h(T) > 0$, assume the contrary, i.e., there exists an evaluation tree $T \in Trees(C(\bar{v}), \mathcal{S})$ such that $h(T) = n > f_C(\bar{v})$. This means there exists a path (starting from the root) which consists of $n + 1$ nodes. Let $C(\bar{v}_0), \ldots, C(\bar{v}_n)$ be the calls that correspond to the nodes in that path, where $\bar{v}_0 = \bar{v}$. By definition of ranking function for a cost relation, for all $i < n$, we have $f_C(\bar{v}_i) - f_C(\bar{v}_{i+1}) \geq 1$ and $f_C(\bar{v}_i) > 0$. Then, it holds that $f_C(\bar{v}) \geq n{+}1 > n = h(T)$, which contradicts the assumption that $f_C(\bar{v}) < h(T)$. □

As it can be observed, in the above examples, the ranking functions that we have used are linear cost expressions. However, in general, we are not restricted to linear, and any cost expression that satisfies the conditions of Definition 6 can be used as ranking functions. The following example demonstrates the need for non-linear ranking functions.

Example 6 Consider the following two loops:

$$\langle P(x, y, z) \rightarrow P(x', y', z'), \{x > 0, y > 0, z > 0, x' = x, z' = z, y > y', z \geq y'\}\rangle$$
$$\langle P(x, y, z) \rightarrow P(x', y', z'), \{x > 0, y > 0, z > 0, x > x', z' = z, z \geq y'\}\rangle$$

which correspond, for example, to the following while loop:

```
while (x>0 && y>0 && z>0) {
    if (*) {
        y=y-1;
    } else {
        x=x-1;
        y=random(1,z);
    }
}
```

No linear ranking function exists that decreases for both loops. However, the non-linear cost expression $f_P(x, y, z) = \mathsf{nat}(x) * \mathsf{nat}(z) + \mathsf{nat}(y)$ is a ranking function which can be used to bound the number of iterations.

In the current implementation, as we explain later, we have restricted ourselves to linear ranking functions. We infer them by using the algorithm described in [45] and, then, wrap them by nat in order to guarantee that they are always non-negative. This explains why cost expressions, as defined in Definition 1, include nat.

Even though ranking functions inferred using [45] provide an upper bound for the height of the corresponding trees, in some cases we can further refine them and obtain tighter upper bounds. For example, if the difference between the value of the ranking function in each two consecutive calls is guaranteed to be larger than a constant $\delta > 1$, then $\lceil \frac{f_C(\bar{x})}{\delta} \rceil$ is a tighter upper bound. A more interesting case, if each loop $\langle C(\bar{x}) \rightarrow C(\bar{y}), \varphi \rangle \in Loops(C)$ satisfies $\varphi \models f_C(\bar{x}) \geq k * f_C(\bar{y})$ where $k > 1$ is a constant, then the height of the tree is bounded by $\lceil \log_k(f_C(\bar{v}) + 1) \rceil$, as each time the value of the ranking function decreases by k. For instance, given a loop the form: $\langle C(l) \rightarrow C(l'), \{l' = l/3, l > 0\} \rangle$, we find the bound "$\lceil \log_3(\mathsf{nat}(l) + 1) \rceil$" for the height of the tree. These cases are handled in our system.

6 Bounding the Cost per Node

After studying how to obtain upper bounds of the number of internal and leaf nodes in evaluation trees, in this section, we present an automatic method to obtain functions $costr_+(\bar{x})$ and $costnr_+(\bar{x})$, which are upper bounds of the local cost associated to an internal node and of a leaf node, respectively. We first give an intuitive description of the technique on our running example. Consider the evaluation tree in Fig. 6. There is only one leaf node and its local cost is 2. Therefore, we can define $costnr_+(\bar{x}) = 2$. As regards the three internal nodes, observe that the corresponding expressions are instantiations of either:

$$\mathsf{exp}_3 = 38 + 15 * \mathsf{nat}(la - j - 1) + 10 * \mathsf{nat}(la)$$
$$\mathsf{exp}_4 = 37 + 15 * \mathsf{nat}(lb - j - 1) + 10 * \mathsf{nat}(lb)$$

Knowing the expressions which generate the possible values in nodes is important, since if we know (or have a safe approximation of) the values of the variables which appear in such expressions, then it is possible to obtain an upper bound of the cost of nodes. Therefore, we split the construction of $costr_{+}(\bar{x})$ and $costnr_{+}(\bar{x})$ in the following two parts.

Invariants First, it is necessary to know what are the possible values to which the different variables in $\mathtt{exp}_3$ and $\mathtt{exp}_4$ can be instantiated. Computing this information is usually undecidable or impractical, but it can be approximated (by means of a superset of the actual values) using static program analysis. One possible way to approximate it is to infer (linear) constraints between the values of the variable in each node and the initial values. For example, for the equations in Fig. 6, we are interested in obtaining constraints between the root call $C(l_0, a_0, la_0, b_0, lb_0)$ and the call in any node $C(l, a, la, b, lb)$. Note that for a variable x we use x_0 to refer to the value of x at the root call. The following linear constraints describe a (possible) relation:

$$\psi = \{0 \leq l \leq l_0, a = a_0, la \leq la_0, b = b_0, lb \leq lb_0\}$$

In other words, ψ is a loop *invariant* that holds between the initial values $\{l_0, a_0, la_0, b_0, lb_0\}$ and the variables in any recursive call $C(l, a, la, b, lb)$ during the evaluation.

Upper Bounds of Cost Expressions The invariant can then be used to infer upper bounds for $\mathtt{exp}_3$ and $\mathtt{exp}_4$. Since $\mathtt{exp}_3$ and $\mathtt{exp}_4$ are monotonic in their nat sub-expressions, as stated in Proposition 1, it is enough to obtain upper bounds for those sub-expressions in order to obtain upper bounds for $\mathtt{exp}_3$ and $\mathtt{exp}_4$. For maximizing $\mathtt{exp}_3$, we need to compute an upper bound for $la - j - 1$ in the context of the invariant ψ conjoined with the local constraints φ_3, associated to (3). By *maximizing* $la - j - 1$ w.r.t. $\{l_0, a_0, la_0, b_0, lb_0\}$, we infer that $la_0 - 1$ is an upper bound for $la - j - 1$ since $\psi \wedge \varphi_3 \models \{la \leq la_0, j \geq 0\}$. Similarly, we obtain the upper bounds la_0, $lb_0 - 1$ and lb_0 for la, $lb - j - 1$, and lb, respectively. By putting all pieces together we obtain that:

$$\begin{aligned}\mathtt{mexp}_3 &= 38 + 15 * \mathsf{nat}(la_0 - 1) + 10 * \mathsf{nat}(la_0)\\ \mathtt{mexp}_4 &= 37 + 15 * \mathsf{nat}(lb_0) + 10 * \mathsf{nat}(lb_0)\end{aligned}$$

are upper bounds for $\mathtt{exp}_3$ and $\mathtt{exp}_4$, respectively. Then, we use $\max(\{\mathtt{mexp}_3, \mathtt{mexp}_4\})$ as an upper bound for all possible expressions in the internal nodes of any possible evaluation tree for $C(l_0, a_0, la_0, b_0, lb_0)$. We now formalize the two steps that have been described above.

6.1 Invariants

Computing an *invariant*, in terms of linear constraints, that holds 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 $C(\bar{x}_0)$ and the arguments of a specific recursive call $C(\bar{x})$ during the evaluation, denoted $\langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \psi\rangle$, and we have a loop $\langle C(\bar{x}) \rightarrow C(\bar{y}), \varphi\rangle \in Loops(C)$, then we can apply the loop one more step and get the new *calling context* (or *context* for short) $\langle C(\bar{x}_0) \rightsquigarrow C(\bar{y}), \exists \bar{x}.(\psi \wedge \varphi)\rangle$. The following definition describes how from a set of contexts I we learn more contexts by applying all loops in a relation. We denote by $\mathcal{R}$ the set of all possible contexts for C, and by $\wp(\mathcal{R})$ all subsets of C that include $I_0 = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{\bar{x}_0 = \bar{x}\}\rangle$.

Definition 7 (loop invariants) For a relation C, let $\mathcal{T}_C : \wp(\mathcal{R}) \mapsto \wp(\mathcal{R})$ be an operator defined:

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

which derives a set of contexts, from a given context X, by applying all loops. The loop invariant I_C is defined as $\cup_{i \in \omega} \mathcal{T}_C^i(\{I_0\})$.

Example 7 Let us compute I_C for the loops that we have computed in Example 3. Let $\bar{x}_0 = \langle l_0, a_0, la_0, b_0, lb_0 \rangle$ and $\bar{x} = \langle l, a, la, b, lb \rangle$. The initial context is

$$I_0 = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{l = l_0, a = a_0, la = la_0, b = b_0, lb = lb_0\}\rangle$$

In the first iteration we compute $\mathcal{T}_C^0(\{I_0\}) = \{I_0\}$. In the second iteration we compute $\mathcal{T}_C^1(\{I_0\})$, which results in the contexts

$$\begin{array}{l} I_1 = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{\underline{l < l_0}, a = a_0, \underline{la = la_0 - 1}, b = b_0, lb = lb_0, \underline{l_0 > 0}\}\rangle \\ I_2 = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{\underline{l < l_0}, a = a_0, la = la_0, b = b_0, \underline{lb = lb_0 - 1}, \underline{l_0 > 0}\}\rangle \end{array}$$

where I_1 and I_2 correspond to applying respectively the first and second loops on I_0. The underlined constraints are the modifications due to the application of the loop. Note that in I_1 (resp. I_2) the variable la_0 (resp. lb_0) decreases by one. The third iteration $\mathcal{T}_C^2(\{I_0\})$, i.e., $\mathcal{T}_C(\{I_1, I_2\})$, results in

$$\begin{array}{l} I_3 = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{l < l_0, a = a_0, \underline{la = la_0 - 2}, b = b_0, lb = lb_0, l_0 > 0\}\rangle \\ I_4 = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{l < l_0, a = a_0, la = la_0 - 1, b = b_0, \underline{lb = lb_0 - 1}, l_0 > 0\}\rangle \\ I_5 = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{l < l_0, a = a_0, la = la_0, b = b_0, \underline{lb = lb_0 - 2}, l_0 > 0\}\rangle \\ I_6 = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{l < l_0, a = a_0, \underline{la = la_0 - 1}, b = b_0, lb = lb_0 - 1, l_0 > 0\}\rangle \end{array}$$

where I_3 and I_4 originate from applying the loops to I_1, and I_5 and I_6 from applying the loops to I_2. The modifications on the constraints reflect that, when applying a loop, either we decrease la or lb. After three iterations, the invariant I_C includes $\{I_0, \ldots, I_6\}$. More iterations will add more contexts that further modify the value of la or lb. Therefore, the invariant I_C grows indefinitely in this case.

The following lemma guarantees that I_C, as defined in Definition 7, is a loop invariant, i.e., it holds between the initial call and any call in the corresponding evaluation tree.

Lemma 2 *Let $C(\bar{v})$ be a call, then $\forall T \in Trees(C(\bar{v}), \mathcal{S})$ and $\forall node(C(\bar{w}), \_, \_) \in T$, there exists $\langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \psi \rangle \in I_C$ such that $\{\bar{x}_0 = \bar{v} \wedge \bar{x} = \bar{w}\} \models \psi$.*

Proof Given an initial call $C(\bar{v})$ and an evaluation tree $T \in Trees(C(\bar{v}), \mathcal{S})$, we show by induction that if $node(C(\bar{w}), \_, \_) \in T$ is at a level n (the level of the root is 0), then there exists $\langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \psi \rangle \in \cup_{0 \leq i \leq n} \mathcal{T}_C^i(\{I_0\})$ such that $\{\bar{x}_0 = \bar{v} \wedge \bar{x} = \bar{w}\} \models \psi$. Then, since $\mathcal{T}_C$ is continuous over the lattice $\langle \wp(\mathcal{R}), \{I_0\}, \mathcal{R}, \subseteq, \cup, \cap \rangle$, it holds for the least fixed point $I_C = \cup_{i \in \omega} \mathcal{T}_C^i(I_0)$ and any level.

Base case. If $n = 0$, it is obvious that the lemma holds using the initial context which is in $\mathcal{T}_C^0(\{I_0\})$.

Induction step. Assume the above lemma holds for any node at a level smaller than n. Consider a node $node(C(\bar{w}), \_, \_) \in T$ at level $n \geq 1$, and let its parent node be $node(C(\bar{w}'), \_, \_) \in T$. By the induction assumption, since the parent level is $n - 1$, there exists $I = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \psi\rangle \in \cup_{0 \leq i < n} \mathcal{T}_C^i(\{I_0\})$ such that $\bar{x}_0 = \bar{v} \wedge \bar{x} = \bar{w}' \models \psi$. By the definition of $Loops(C)$, there exists a loop $\ell = \langle C(\bar{x}) \rightarrow C(\bar{y}), \varphi\rangle \in Loops(C)$ such that $\bar{x} = \bar{w}' \wedge \bar{y} = \bar{w} \models \varphi$. Since the context I must have been introduced by $\mathcal{T}_C^k(\{I_0\})$ for some $k < n$, then at iteration $k + 1 \leq n$ the operator $\mathcal{T}_C$ will use I and ℓ to generate $\langle C(\bar{x}_0) \rightsquigarrow C(\bar{y}), \bar{\exists}\bar{x}_0 \cup \bar{y}.(\psi \wedge \varphi)\rangle \cup_{0 \leq i \leq n} \mathcal{T}_C^i(\{I_0\})$. Moreover, $\bar{x}_0 = \bar{v} \wedge \bar{y} = \bar{w} \models \bar{\exists}\bar{x}_0 \cup \bar{y}.(\psi \wedge \varphi)$. □

The problem with Definition 7 is that it is not computable in general since the invariant I_C possibly consists of an infinite number of calling contexts, as it happens in our example. In practice, we approximate I_C using abstract interpretation over, for instance, the domain of convex polyhedra [23]. For our example, as an approximation for I_C of Example 7 we obtain the invariant:

$$I_C^{\alpha} = \{\langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{l \leq l_0, a = a_0, la \leq la_0, b = b_0, lb \leq lb_0\}\rangle\}$$

In general, we approximate I_C by a single context $I_C^{\alpha} = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \psi'\rangle\}$ such that $\forall \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \psi\rangle \in I_C.\psi \models \psi'$. This is simply done by replacing $\cup$ in Definition 7 by a convex-hull operation, and applying a widening operator to guarantee termination [23]. It is clear that Lemma 2 also holds for such approximation of I_C.

6.2 Upper Bounds on Cost Expressions

At this point, we want to use the loop invariant in order to obtain upper bounds, in terms of the initial call values, for the values in all internal nodes and leaves in the corresponding evaluation trees. Since the values which appear in the nodes of evaluation trees correspond to different instantiations of the cost expressions in the cost equations, we concentrate first on finding upper bounds for those cost expressions and then combine them to build upper bounds for all internal nodes and all leaves.

Consider, for example, the expression $\mathsf{nat}(la - j - 1)$ which appears in (3) of Fig. 6. 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 that $\langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \psi\rangle$ where $\psi = \{l \leq l_0, a = a_0, \underline{la \leq la_0}, b = b_0, lb \leq lb_0\}$, is a safe approximation of the loop invariant I_C, from which we can observe that the maximum value that la can take is la_0. In addition, from the local constraints φ of (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 $la - j - 1$. In practice, this inference method can be done in a fully automatic way using linear constraints tools (e.g. [13]) as follow:

1. Compute $\phi = \bar{\exists}l_0, a_0, la_0, b_0, lb_0, r.(\psi \wedge \varphi \wedge y = la - j - 1)$, where y is a new variable;

2. *Syntactically* look in ϕ for an expression that can be rewritten to $y \leq f'$, where f' is a linear expression which (obviously) contains only variables from $\{l_0, a_0, la_0, b_0, lb_0\}$.

Given a cost equation $\langle C(\bar{x}) = \mathtt{exp} + \sum_{i=1}^{k} C(\bar{y}_i), \varphi \rangle$ and a safe approximation of its loop invariant $\langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \psi \rangle$, the function below computes an upper bound for exp by maximizing its nat components:

```
1: function $ub\_exp$($\mathtt{exp}$,$\bar{x}_0$,$\varphi$,$\psi$)
2:     $\mathtt{mexp} = \mathtt{exp}$
3:     for all $\mathsf{nat}(f) \in \mathtt{exp}$ do
4:         $\phi = \bar{\exists} \bar{x}_0, y.(\varphi \wedge \psi \wedge y = f)$    // $y$ is a fresh variable
5:         if $\exists f'$ such that $vars(f') \subseteq \bar{x}_0$ and $\phi \models y \leq f'$ then $\mathtt{mexp} =$
               $\mathtt{mexp}[\mathsf{nat}(f)/\mathsf{nat}(f')]$
6:         else return $\infty$
7:     return $\mathtt{mexp}$
```

This function computes an upper bound f' for each expression f which occurs inside a nat function 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).

Example 8 Applying *ub_exp* to the cost expressions $\mathtt{exp}_3$ and $\mathtt{exp}_4$, that appear in (3) and (4) in Fig. 6, w.r.t. the invariant that we have computed in Section 6.1, can be done by maximizing their nat sub-expressions. Similarly to what we have done above for $la - j - 1$, we can find upper bounds for $lb - j - 1$, la and lb as $lb_0 - 1$, la_0 and lb_0 respectively. Therefore, the expressions

$$\begin{aligned} \mathtt{mexp}_3 &= 38 + 15 * \mathsf{nat}(la_0 - 1) + 10 * \mathsf{nat}(la_0) \\ \mathtt{mexp}_4 &= 37 + 15 * \mathsf{nat}(lb_0 - 1) + 10 * \mathsf{nat}(lb_0) \end{aligned}$$

are upper bounds for $\mathtt{exp}_3$ and $\mathtt{exp}_4$.

The lemma below guarantees the soundness of the function *ub_exp*.

Lemma 3 *(soundness of ub_exp) Let* $\langle C(\bar{x}) = \mathtt{exp} + \sum_{i=1}^{k} C(\bar{y}_i), \varphi \rangle$ *be a cost equation for* C*,* $\langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \psi \rangle$ *be a safe approximation of the loop invariant* I_C*, and* $\mathtt{mexp} = ub\_exp(\mathtt{exp}, \bar{x}_0, \varphi, \psi)$*. Then, for any call* $C(\bar{v})$ *and for all* $T \in Trees(C(\bar{v}), \mathcal{S})$*, if* $node(C(\bar{w}), r, \_) \in T$ *such that* r *originates from* exp*, then* $[\![\mathtt{mexp}]\!]_\sigma \geq r$ *where* σ *is a substitution that maps* $\bar{x}_0$ *to* $\bar{v}$*.*

Proof The Lemma is trivially correct when $\mathtt{mexp} = \infty$. For $\mathtt{mexp} \neq \infty$, given $T \in Trees(C(\bar{v}), \mathcal{S})$ and $node(C(\bar{w}), r, \_) \in T$, by Lemma 2, there exists a substitution σ, over $\bar{x}_0$ and the variables of the equation, such that $\sigma \models \bar{x}_0 = \bar{v} \wedge \bar{x} = \bar{w} \wedge \psi \wedge \varphi$ and $r = [\![\mathtt{exp}]\!]_\sigma$. Let $\mathtt{exp}'$ be a cost expression obtained from exp by replacing only one $\mathsf{nat}(f)$ by $\mathsf{nat}(f')$ (lines 4 and 5 in function *ub_exp*). Proposition 1 and the fact that $\psi \wedge \varphi \models f \leq f'$ implies $[\![\mathtt{exp}']\!]_\sigma \geq [\![\mathtt{exp}]\!]_\sigma$. Since mexp is obtained by repeating such replacement for all nat components, at the end we will have $[\![\mathtt{mexp}]\!]_\sigma \geq [\![\mathtt{exp}]\!]_\sigma = r$. □

The following lemma is a completeness lemma for function *ub_exp*, in the sense that if ψ and φ imply that there is f' which is an upper bound for f, then by syntactically looking on ϕ (line 4 of *ub_exp*) we will be able to find one, without guarantees that it will be the tightest one.

Lemma 4 *(completeness of ub_exp) Consider line 5 of ub_exp, if there exists f' such that $\phi \models y \leq f'$ and $\phi = \{c_1, \ldots, c_n\}$, then there exists c_i which can be worked out to $y \leq f''$ (or $y = f''$) where $vars(f'') \subseteq \bar{x}_0$.*

Proof The lemma follows from: (1) if there exists f' such that $vars(f') \subseteq \bar{x}_0$ and $\psi \wedge \varphi \models f \leq f'$ then, $\phi \models y \leq f'$, since $y = f$ and $y \notin vars(\psi \wedge \varphi)$; (2) if $\phi \models y \leq f'$ and $vars(\phi) \subseteq \bar{x}_0 \cup \{y\}$, then y must appear in one of the c_i, which obviously can be worked out to $y \leq f'$; and (3) if there is more than one c_i where y appears, then taking one is safe as they appear in a conjunction. □

6.3 Concluding Remarks

Using Lemmata 2 and 3, the theorem below concludes by building the upper bound expression $costnr_+(\bar{x}_0)$ and $costr_+(\bar{x}_0)$.

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. Let*

- *$\langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \psi \rangle$ be a safe approximation of the loop invariant I_C;*
- *$E_i = \{ub\_exp(\texttt{exp}, \bar{x}_0, \varphi, \psi) \mid \langle C(\bar{x}) = \texttt{exp} + \sum_{j=1}^{k} C(\bar{y}_j), \varphi \rangle \in \mathcal{S}_i\}$, $1 \leq i \leq 2$; and*
- *$costnr_+(\bar{x}_0) = \max(E_1)$ and $costr_+(\bar{x}_0) = \max(E_2)$.*

Then, for any call $C(\bar{v})$ and for all $T \in Trees(C(\bar{v}), \mathcal{S})$, it holds that

- *$\forall node(\_, r, \_) \in internal(T).\ costr_+(\bar{v}) \geq r$; and*
- *$\forall node(\_, r, \_) \in leaf(T).\ costnr_+(\bar{v}) \geq r$.*

Proof Follows from Lemmata 2 and 3. □

Example 9 At this point we have all the pieces in order to compute an upper bound, as described in Proposition 2, for the *CR* depicted in Fig. 3. We start by computing upper bounds for E and D as they are stand-alone cost relations:

	h_+	$costnr_+$	$costr_+$	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 (3) and (4) which results in the cost relation for C depicted in Fig. 6. We have already computed a ranking function for C in Example 4, and $costnr_+$ and $costr_+$ in Example 8, which are then combined into:

$$C_+(l_0, a_0, la_0, b_0, lb_0) = 2 + \mathsf{nat}(l_0) * max(\{\texttt{mexp}_3, \texttt{mexp}_4\})$$

By reasoning similarly, we obtain the upper bound for *Delete* shown in Table 1.

Table 1 Upper bounds computed automatically

Benchmark	Properties	Upper bound
Polynomial*	a,b,c	216
DivByTwo	a,b	$8\mathsf{log}_2(\mathsf{nat}(2x{-}1){+}1){+}14$
ArrayReverse	a	$14\mathsf{nat}(x){+}12$
Concat	a,c	$11\mathsf{nat}(x){+}11\mathsf{nat}(y){+}25$
Incr	a,c	$19\mathsf{nat}(x{+}1){+}9$
ListReverse	a,b,c	$13\mathsf{nat}(x){+}8$
MergeList	a,b,c	$29\mathsf{nat}(x+y){+}26$
Power		$10\mathsf{nat}(x){+}4$
Cons*	a,b	$22\mathsf{nat}(x{-}1){+}24$
MergeSort^n	a,b,c	$2\mathsf{nat}(-x+y+1)(\mathsf{log}_2(\mathsf{nat}(-2x+2y-1)+1)+1)$
EvenDigits	a,b,c	$\mathsf{nat}(x)(8\mathsf{log}_2(\mathsf{nat}(2x{-}3)+1)+24)+9\mathsf{nat}(x)+9$
ListInter	a,b,c	$\mathsf{nat}(x)(10\mathsf{nat}(y)+43)+21$
SelectSort	a,c	$\mathsf{nat}(x{-}2)(17\mathsf{nat}(x{-}2)+34)+9$
FactSum	a	$\mathsf{nat}(x{+}1)(9\mathsf{nat}(x)+16)+6$
Delete	a,b,c	$3+\mathsf{nat}(l)\max(38{+}15\mathsf{nat}(la{-}1){+}10\mathsf{nat}(la),$ $37{+}15\mathsf{nat}(lb{-}1){+}10\mathsf{nat}(lb))$
MatMult	a,c	$\mathsf{nat}(y)(\mathsf{nat}(x)+10)(27\mathsf{nat}(x)+10)+17$
Hanoi		$20(2^{\mathsf{nat}(x)}){-}17$
Fibonacci		$18(2^{\mathsf{nat}(x-1)}){-}13$
BST*	a,b	$96(2^{\mathsf{nat}(x)}){-}49$

7 Improving Accuracy in Divide and Conquer Programs

We have presented in Section 4 an approximation approach, based on bounding both the number of nodes in evaluation trees and the cost per node, which is able to provide upper bounds for a large class of programs. However, there is an important class of programs known as *divide and conquer* for which the node-count upper bound does not compute sufficiently precise upper bounds. Intuitively, the reason for this is that divide and conquer programs have a branching factor greater than one. Therefore, the number of nodes grows exponentially with the height of the evaluation tree. However, the size of the input data decreases so quickly from one level of the tree to the next one that the *sum* of the local cost expressions in the nodes at each level does not increase from one level to another.

In this section we propose an approximation mechanism, which we refer to as *level-count upper bound* which is based on bounding both the number of levels in evaluation trees and the total cost per level. It allows obtaining accurate upper bounds for divide and conquer programs.

7.1 Level-count upper bound

Given an evaluation tree T, we denote by $\mathsf{Sum\_Level}(T,i)$ the sum of the local cost of all nodes in T which are at depth i, i.e., at distance i from the root. As before, we write $h(T)$ to denote the height of T.

Proposition 3 *(level-count upper bound) Let C be a cost relation. We define function C_+ as:*

$$C_+(\bar{x}) = l_+(\bar{x}) * costl_+(\bar{x})$$

where $l_+(\bar{x})$ and $costl_+(\bar{x})$ are closed-form functions defined on $\mathbb{Z}^n \mapsto \mathbb{R}_+$. Then, C_+ is an upper bound of C if for all $\bar{v} \in \mathbb{Z}^n$ and $T \in Trees(C(\bar{v}), \mathcal{S})$, it holds:

1. $l_+(\bar{v}) \geq h(T) + 1$*; and*
2. $\forall\, 0 \leq i \leq h(T)\ .\ costl_+(\bar{v}) \geq \mathsf{Sum\_Level}(T, i)$.

Proof The proposition is trivially correct by the definition of upper bound and *Answers*. □

Similarly to what we have done for $h_+(\bar{x})$ in Section 5, the function $l_+(\bar{x})$ can simply be defined as $l_+(\bar{x}) = \mathsf{nat}(f_C(\bar{x})) + 1$. Finding an accurate $costl_+$ function is not easy in general, which makes Proposition 3 not as widely applicable as Proposition 2.

7.2 Divide and Conquer Programs

We now provide a formal definition of *divide and conquer* programs and show that for all programs which fall into this class it is possible to apply the level-count upper bound approach. Intuitively, a program belongs to the divide and conquer class when the local cost of each node in the evaluation tree is guaranteed to be greater than or equal to the sum of the local costs of its children. As we will see, this guarantees that $\mathsf{Sum\_Level}(T, k) \geq \mathsf{Sum\_Level}(T, k+1)$. In that case, we can simply take the local cost of the root node as an upper bound of $costl_+(\bar{x})$.

Often we have multiple recursive and non-recursive equations for a cost relation. Checking that the local cost of a node is greater than the sum of those of its children needs to take into account all possible combinations of cost expressions produced by picking a recursive equation followed by picking any equation—be it recursive or not—for each recursive call in such equation. We now define the set of *child local-cost expressions* as a set of triplets composed by two cost expressions linked by a set of constraints which are all those achievable in the combinations explained.

Definition 8 (Child local-cost expressions) The set of child local-cost expressions of a stand-alone cost relation C, denoted $Child\_Exps(C)$, is defined as

$$Child\_Exps(C) = \left\{ \langle \mathsf{exp}, \mathsf{exp}', \psi \rangle \;\middle|\; \begin{array}{l} \langle C(\bar{x}) = \mathsf{exp} + \sum_{i=1}^{k} C(\bar{y}_i), \varphi \rangle \in \mathcal{S},\ \text{where } k \geq 1 \\ \forall\, 1 \leq i \leq k.\ \langle C(\bar{y}_i) = \mathsf{exp}_i + \sum_{j=1}^{k_i} C(\bar{z}_j), \varphi_i \rangle \in \mathcal{S} \\ \mathsf{exp}' = \mathsf{exp}_1 + \cdots + \mathsf{exp}_k \\ \psi = \bar{\exists} vars(\mathsf{exp}) \cup vars(\mathsf{exp}').\varphi \wedge \varphi_1 \wedge \cdots \wedge \varphi_k \end{array} \right\}$$

Example 10 Consider a *CR* in which C is defined by the two equations:

$$\begin{array}{l} \langle C(x) = 0, \{x \leq 0\} \rangle \\ \langle C(x) = \mathsf{nat}(x) + C(x_1) + C(x_2), \varphi \rangle \end{array}$$

where $\varphi = \{x > 0, x_1 + x_2 + 1 \leq x, x \geq 2 * x_1, x \geq 2 * x_2, x_1 \geq 0, x_2 \geq 0\}$. It corresponds to a divide and conquer problem such as merge-sort when the cost model used counts the number of comparison instructions executed, which is a usual criteria for comparing sorting programs and algorithms. The set $\mathit{Child\_Exps}(C)$ consists of:

$$\mathit{Child\_Exps}(C) = \left\{ \begin{array}{l} \langle \mathsf{nat}(x), 0, \varphi \wedge x_1 \leq 0 \wedge x_2 \leq 0 \rangle \\ \langle \mathsf{nat}(x), \mathsf{nat}(x_1), \varphi \wedge x_1 \leq 0 \wedge \varphi_2 \rangle \\ \langle \mathsf{nat}(x), \mathsf{nat}(x_2), \varphi \wedge \varphi_1 \wedge x_2 \leq 0 \rangle \\ \langle \mathsf{nat}(x), \mathsf{nat}(x_1) + \mathsf{nat}(x_2), \varphi \wedge \varphi_1 \wedge \varphi_2 \rangle \end{array} \right\}$$

where φ_1 (resp. φ_2) is a renaming apart of φ, except for the variable x_1 (resp. x_2).

The following lemma provides a sufficient condition for a cost relation falling into the divide and conquer class, i.e., for Proposition 3 to be applicable. It is based on checking that each cost expression contributed by an equation is greater than or equal to the sum of the cost expressions contributed by the corresponding immediate recursive calls.

Lemma 5 *(A sufficient condition for divide and conquer) Let C be a stand-alone cost relation. If for any* $\langle \mathsf{exp}, \mathsf{exp}', \psi \rangle \in \mathit{Child\_Exps}(C)$ *and any* $\sigma : \mathit{vars}(\mathsf{exp}) \cup \mathit{vars}(\mathsf{exp}') \mapsto \mathbb{Z}$ *such that* $\sigma \models \psi$ *it holds that* $[\![\mathsf{exp}]\!]_\sigma \geq [\![\mathsf{exp}']\!]_\sigma$*, then for any call* $C(\bar{v})$*, a corresponding evaluation tree* $T \in \mathit{Trees}(C(\bar{v}), \mathcal{S})$*, and a level k, it holds that* $\mathsf{Sum\_Level}(T, k) \geq \mathsf{Sum\_Level}(T, k+1)$.

Proof Assume the contrary, i.e., the condition holds but there exists a call $C(\bar{v})$, a corresponding evaluation tree $T \in \mathit{Trees}(C(\bar{v}), \mathcal{S})$, and a level k, such that $\mathsf{Sum\_Level}(T, k) < \mathsf{Sum\_Level}(T, k+1)$. This means that there exists a node $\mathit{node}(C(\bar{v}), r, \langle T_1, \ldots, T_n \rangle)$ at level k, such that for each subtree $T_i = \mathit{node}(C(\bar{v}_i), r_i, \_)$ it holds $r < r_1 + \cdots + r_n$. Assume this node was constructed using an equation $\mathcal{E} = \langle C(\bar{x}) = \mathsf{exp} + \sum_{i=1}^{m} C(\bar{y}_i), \varphi \rangle \in \mathcal{S}$ and that $\langle C(\bar{y}_i) = \mathsf{exp}_i + \sum_{j=1}^{m_i} C(\bar{z}_j), \varphi_i \rangle \in \mathcal{S}$ was used to match each call $C_i(\bar{y}_i)$ in $\mathcal{E}$. Then, there exists σ verifying $\sigma \models \varphi \wedge \varphi_1 \wedge \cdots \wedge \varphi_m \models \bar{x} = \bar{v} \wedge \bar{y}_1 = \bar{v}_1 \wedge \cdots \wedge \bar{y}_m = \bar{v}_m$, such that $[\![\mathsf{exp}]\!]_\sigma < [\![\mathsf{exp}_1 + \cdots + \mathsf{exp}_m]\!]_\sigma$, which contradicts the assumption that the condition holds. □

The intuition of the above lemma is that for each node in any evaluation tree, there exists a tuple $\langle \mathsf{exp}, \mathsf{exp}', \psi \rangle \in \mathit{Child\_Exps}(C)$ and a substitution $\sigma : \mathit{vars}(\mathsf{exp}) \cup \mathit{vars}(\mathsf{exp}') \mapsto \mathbb{Z}$ such that $\sigma \models \psi$, $[\![\mathsf{exp}]\!]_\sigma$ is equal to its local cost, and $[\![\mathsf{exp}']\!]_\sigma$ is equal to the sum of its children local costs.

Theorem 2 *Let C be a stand-alone cost relation which satisfies the divide and conquer condition of Lemma 5,* $E = \{\mathit{ub\_exp}(\mathsf{exp}, \bar{x}_0, \varphi, \{\bar{x}_0 = \bar{x}\}) \mid \langle C(\bar{x}) = \mathsf{exp} + \sum_{i=1}^{k} C(\bar{y}_i), \varphi \rangle \in \mathcal{S}\}$*, and* $\mathit{costl}_+(\bar{x}) = \max(E)$*. Then, for any call* $C(\bar{v})$*, a corresponding evaluation tree* $T \in \mathit{Trees}(C(\bar{v}), \mathcal{S})$*, and a level k, it holds that* $\mathit{costl}_+(\bar{v}) \geq \mathsf{Sum\_Level}(T, k)$.

Proof It follows from Lemmata 3 and 5. □

Example 11 Consider again the cost relation C defined in Example 10. Computing the set E of Theorem 2 results in $\{\mathsf{nat}(x), 0\}$, and therefore $costl_+(x) = \mathsf{nat}(x)$. Using the techniques described in Section 5 we can automatically compute

$$l_+(x) = \lceil \log_2(\mathsf{nat}(x)+1)\rceil + 1$$

Thus, we obtain the upper bound $C_+(x) = \mathsf{nat}(x) * (\lceil \log_2(\mathsf{nat}(x) + 1)\rceil + 1)$. Note that this upper bound is inferred in a fully automatic way by our prototype which is described in Section 10. By using the node-count approach, we would obtain $C_+(x) = \mathsf{nat}(x) * (2^{\lceil \log_2(\mathsf{nat}(x)+1)\rceil} - 1) = \mathsf{nat}(x)^2$ as upper bound.

8 Direct Recursion using Partial Evaluation

Our approach requires that all recursions be direct. However, automatically generated *CRSs* often contain recursions which are not direct, i.e., cycles involve more than one function.

Example 12 The cost analyzer of [6, 7], in order to define the cost of the "for" loop in the program in Fig. 1, instead of (8) and (9) (relation E) in Fig. 3, produces the following equations:

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

The new E relation captures the cost of evaluating the loop condition "$j < la - 1$" (5 cost units) plus the cost of its continuation, captured by F. In (9') the relation F corresponds to the exit of the loop (it calls the auxiliary relation H, which represents the cost of exiting the loop, i.e., 0 units). Equation (10) captures the cost of one iteration, which accumulates 10 cost units and calls E recursively.

In this section, we present an automatic transformation of *CRSs* into *directly recursive* form. The transformation is done by replacing calls to intermediate relations by their definitions using *unfolding*. For instance, given the *CRS* in Example 12, if we keep E and unfold the remaining relations in the example (F, G, and H), we obtain the equations for E shown in Fig. 3.

8.1 Binding Time Classification

We now recall some standard terminology on graphs. A *directed graph* G is a pair $\langle N, A\rangle$ where N is the set of *nodes* and $A \subseteq N \times N$ is the set of *arcs*. Given a graph $G = \langle N, A\rangle$, a set of nodes $S \subseteq N$ is *strongly connected* if $\forall n, n' \in S$ we have that n' is reachable from n. The *strongly connected components* of $G = \langle N, A\rangle$ is a partition of N into the largest possible strongly connected sets. Given a graph G we write $SCC(G)$ to denote its strongly connected components. Given a graph $G = \langle N, A\rangle$ and a set $S \subseteq N$, the *subgraph* of G w.r.t. S, denoted $G|_S$, is defined as $G|_S = \langle S, A \cap (S \times S)\rangle$. Also, given a strongly connected component S, a node $n \in S$ is a *covering point* for $G|_S$ if $G|_{S\setminus\{n\}}$ is an acyclic graph, i.e., n is a covering point of $G|_S$ if n is part of all cycles in $G|_S$. The problem of finding a minimal set of nodes to delete from a cyclic graph in order to convert it into an acyclic graph is also known as the *feedback vertex set* problem in computational complexity theory. The *feedback vertex set* decision problem is NP-complete in general, but for reducible graphs it is linear [50]. As explained in [50], control flow graphs originating from structured programming languages are often reducible, since usually there are no jumps to the middle of a loop. Moreover, since our interest is only in checking if there exists a feedback set of size 1, when the graph is not reducible, we can solve it in quadratic time simply by removing a node n from $G|_S$ and checking if $G|_{S\setminus\{n\}}$ is acyclic.

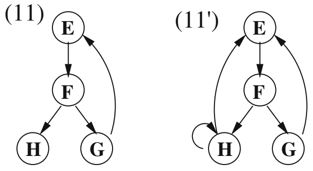
Note that, when the *CRS* originates from a structured program (i.e., without jumps), it is not common to have SCCs without covering points. This is due to: (1) As done in [6], each structured loop (e.g., while, for, etc.) can be transformed to a separated method in tail recursive form, and the loop itself is replaced by a call to this method. Therefore, the program becomes even more structured since nested loops are not anymore in the same SCC. (2) SCCs of a *CRS* coincide with those of the original program (after extracting the loops) and, in structured programs, it is common that each SCC has a point were all cycles go through (e.g, the entry of loop, the entry of a recursive method, etc). However, a covering point might not exist in programs with complex mutual recursion, as we explain in Section 9.

The notion of *unfolding* corresponds to the intuition of replacing a call to a relation by the definition of the corresponding relation. Naturally, this process in the presence of recursive relations might be non-terminating. Intuitively, the transformation proposed removes intermediate relations from the *CRS* and we achieve direct recursion if at most one relation remains per strongly connected component in the call graph of the original *CRS*. In this section, we find a *Binding Time Classification* (or BTC for short) which ensures the termination of the unfolding process by declaring which relations are *residual*, i.e., they have to remain in the *CRS*. The remaining relations are considered *unfoldable*, i.e., they are eliminated. To define such BTC, we associate a *call graph* to each *CRS* $\mathcal{S}$ as follows. Given a *CRS* $\mathcal{S}$ with $C, D \in rel(\mathcal{S})$, we say that C *calls* D in $\mathcal{S}$, denoted $C \mapsto_{\mathcal{S}} D$, iff there is an equation $\langle C(\bar{x}) = \mathtt{exp} + \sum_{i=1}^{k} D_i(\bar{y}_i), \varphi\rangle \in \mathcal{S}$ such that $D_i = D$ for some $i \in \{1, \ldots, k\}$. The call graph associated to $\mathcal{S}$, denoted $\mathcal{G}(\mathcal{S})$, is the directed graph obtained from $\mathcal{S}$ by taking $N = rel(\mathcal{S})$ and where $(C, D) \in A$ iff $C \mapsto_{\mathcal{S}} D$. We now present sufficient conditions under which *CRSs* can be put into directly recursive form. In particular, we require that the graph associated to the *CRS* be of *minimal coverage*.

Definition 9 (minimal coverage) A graph $G = \langle N, A\rangle$ is *of minimal coverage* iff $\forall S \in SCC(G)$, there exists $n \in S$ such that n is a covering point for $G|_S$.

Intuitively, a graph is of minimal coverage if each SCC has a *covering point*. Let us see some examples.

Example 13 Given the *CRS* $\mathcal{S}$ of Example 12, its call graph $\mathcal{G}(\mathcal{S})$ is shown on the left hand side of the figure below. Also, we have that $SCC(\mathcal{G}(\mathcal{S})) = \{\{E, F, G\}, \{H\}\}$.



The strongly connected component which could be problematic as regards minimal coverage (more than one element) is $\{E, F, G\}$. Since there is just one cycle, any of the nodes is a covering point and therefore G is of minimal coverage. However, if we replace (11) in Example 12 with (11') below:

$$(11')\ \langle H(j', la') \leftarrow 1 + H(j'', la') + E(j'', la'), \{j'' = j' - 1\}\rangle$$

we obtain the graph to the right of the figure. Now, $SCC(\mathcal{G}(\mathcal{S})) = \{\{E, F, G, H\}\}$, i.e., all nodes are in the same strongly connected component, and we have three cycles ($\langle E, F, G\rangle$, $\langle E, F, H\rangle$, and $\langle H\rangle$) which belong to such strongly connected component. Unfortunately, this time there is no node which belongs simultaneously to the three cycles.

As shown in the example above, there are graphs which are not of minimal coverage. Therefore, there are *CRSs* which cannot be put into canonical form. However, structured loops (built using for, while, etc.) and the recursive patterns found in most programs naturally result in *CRSs* whose reachability graphs are of minimal coverage.

We can now define the notion of *directly recursive* BTC which ensures both the termination of our partial evaluation process and the effectiveness of the transformation (i.e., we indeed obtain direct recursion form). Formally, a relation D is *reachable* from a relation C in $\mathcal{S}$ iff there is a path from C to D in $\mathcal{G}(\mathcal{S})$. A relation C is *recursive* iff C is reachable from itself. It is *directly recursive* if $(C \mapsto_{\mathcal{S}} D \wedge D \neq C) \Rightarrow C$ is not reachable from D in $\mathcal{S}$, i.e., there cannot be cycles in the reachability relation (recursion) of length greater than one.

Definition 10 (directly recursive BTC) Given a *CRS* $\mathcal{S}$ with graph G, a BTC btc for $\mathcal{S}$ is *directly recursive* if for all $S \in SCC(G)$ the following two conditions hold:

(DR) if $s_1, s_2 \in S$ and $s_1, s_2 \in$ btc, then $s_1 = s_2$.
(TR) if S has a cycle, then there exists $s \in S$ such that $s \in$ btc.

Condition (**DR**) ensures that all recursions in the transformed *CRS* are direct, as there is only one residual relation per SCC. Condition (**TR**) guarantees that the unfolding process terminates, as there is a residual relation per cycle.

A directly recursive BTC for Example 12 is btc $= \{E\}$. In our implementation we include in BTCs only the covering point of SCCs which contain cycles, but not that of components without cycles. This way of computing BTCs, in addition to ensuring direct recursion, also eliminates all intermediate cost relations which are not part of cycles. Coming back to Example 12, our implementation computes btc $= \{E\}$. This is why the *CRS* shown in Fig. 3 does not include equations for H.

8.2 Partial Evaluation of Cost Relations

We now present a *Partial Evaluation* [33] (PE for short) algorithm for transforming *CRSs*. Unfolding, in this context, in addition to taking care of combining arithmetic expressions, also has to combine the linear constraints and to consider a BTC btc to control the transformation process. The next definition of unfolding, given a call to a relation, produces a specialization for such call by unfolding all calls to relations which are marked as *unfoldable* in btc.

Definition 11 (unfolding) Given a *CRS* $\mathcal{S}$, a call $C(\bar{x}_0)$ such that $C \in rel(\mathcal{S})$, a set of linear constraints $\varphi_{\bar{x}_0}$ over the variables $\bar{x}_0$, and a BTC btc for $\mathcal{S}$, a *specialization* $\langle E, \varphi\rangle$ is obtained by unfolding $C(\bar{x}_0)$ and $\varphi_{\bar{x}_0}$ in $\mathcal{S}$ w.r.t. btc, denoted $\mathsf{Unfold}(\langle C(\bar{x}_0), \varphi_{\bar{x}_0}\rangle, \mathcal{S}, \mathsf{btc}) \rightsquigarrow \langle E, \varphi\rangle$, if one of the following conditions hold:

(res) $(C \in \mathsf{btc} \wedge \varphi \neq \mathsf{true}) \wedge \langle E, \varphi\rangle = \langle C(\bar{x}_0), \varphi_{\bar{x}_0}\rangle$.
(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 we have that:

1. $\langle C(\bar{x}) = \mathsf{exp} + \sum_{i=1}^{k} D_i(\bar{y}_i), \varphi_C\rangle$ is a renamed apart equation in $\mathcal{S}$ such that φ' is satisfiable in $\mathbb{Z}$, where $\varphi' = \varphi_{\bar{x}_0} \wedge \varphi_C[\bar{x}_0/\bar{x}]$.
2. $\mathsf{Unfold}(\langle D_i(\bar{y}_i), \varphi'\rangle, \mathcal{S}, \mathsf{btc}) \rightsquigarrow \langle e_i, \varphi_i\rangle$ for all $i \in \{1, \ldots, k\}$.

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 $C(\bar{x}_0)$ and constraints $\varphi_{\bar{x}_0}$, as long as $\varphi_{\bar{x}_0}$ is not the initial one (true). The latter condition is added in order to enforce the initial unfolding step for relations marked as residual. In all subsequent calls to Unfold different from the initial one, the constraints are different from true. The second case **(unf)** corresponds to continuing the unfolding process. Step 1 is non deterministic in general, since cost relations are often defined by means of several equations. Furthermore, since expressions are transitively unfolded, step 2 may also provide multiple solutions. As a result, unfolding may produce multiple outputs. Also, note that the final constraint φ can be unsatisfiable. In such case, we simply do not regard $\langle E, \varphi\rangle$ as a valid unfolding. In the following, we denote by $\overset{unf}{=}_e$ an "unfolding step" performed by **unf** where an equation e is selected to replace a function call by its right hand side.

Example 14 Given the initial call $\langle E(la, j), true\rangle$, we obtain an unfolding by performing the following steps.

$$\begin{array}{ll}
\langle E(la, j), true\rangle & \overset{unf}{=}_{(8')} \\
\langle 5 + F(la, j, j', la'), \{j' = j, la' = la - 1, j' \geq 0\}\rangle & \overset{unf}{=}_{(10)} \\
\langle 5 + G(la, j, j', la'), \{j' = j, la' = la - 1, j' \geq 0, j' < la'\}\rangle & \overset{unf}{=}_{(12)} \\
\langle 15 + E(la, j''), \{j < la - 1, j \geq 0\}\rangle &
\end{array}$$

The last call $E(la, j'')$ cannot be further unfolded because the relation belongs to btc and $\varphi \neq true$.

In the above definition, from each result of unfolding, we can build a *residual* equation. Given $\mathsf{Unfold}(\langle C(\bar{x}_0),\ \varphi_{\bar{x}_0}\rangle, \mathcal{S}, \mathsf{btc}) \rightsquigarrow \langle E, \varphi\rangle$, its corresponding residual equation is $\langle C(\bar{x}_0) = E, \varphi\rangle$. We use $\mathsf{Residuals}(\langle C(\bar{x}_0), \varphi_{\bar{x}_0}\rangle, \mathcal{S}, \mathsf{btc})$ to denote the set of residual equations for $\langle C(\bar{x}_0), \varphi_{\bar{x}_0}\rangle$ in $\mathcal{S}$ w.r.t. φ. Now, we obtain a *partial evaluation* of C by collecting all residual equations for the call $\langle C(\bar{x}_0), \mathsf{true}\rangle$ where $\bar{x}_0$ are distinct variables.

Definition 12 (partial evaluation) Given a *CRS* $\mathcal{S}$, a relation C, and a BTC btc for $\mathcal{S}$, the *partial evaluation* for C in $\mathcal{S}$ w.r.t. btc is defined as:

$$\bigcup_{D \in \mathsf{btc} \cup \{C\}} \mathsf{Residuals}(\langle D(\bar{x}_0), \mathsf{true}\rangle, \mathcal{S}, \mathsf{btc})$$

The above definition provides an algorithm for partial evaluation of *CRSs*. In terms of PE [33], the algorithm we propose is an *off-line* PE which at the *global control* level is monovariant, since the initial constraint is true for all residual relations, and at the *local-control* it unfolds all calls to unfoldable relations and residualizes all calls to residual relations. Note that, in addition to the relations in btc, we also generate equations for the initial relation C.

Example 15 The partial evaluation of the equations of Example 12 w.r.t. the call of Example 14 are (8) and (9) of Fig. 3. Equation (9) is obtained from the unfolding steps depicted in Example 14 and (8) from an unfolding derivation where the selected equations are (8'), then (9') and finally (11). As expected, the resulting *CRS* is directly recursive.

The lemma below shows that partial evaluation is an effective way of obtaining direct recursion. It easily follows by the definition of BTC.

Lemma 6 *Let $\mathcal{S}$ be a CRS of minimal coverage and let C be a relation. Let* btc *be a directly recursive BTC for $\mathcal{S}$. Then,*

1. *Partial evaluation for C in $\mathcal{S}$ w.r.t.* btc *produces a CRS $\mathcal{S}'$ which is directly recursive and,*
2. *$\mathcal{S}'$ is obtained in finite time.*

Proof The proof is by contradiction. Let us first prove claim 1. Assume that we have a relation in $\mathcal{S}'$ which is not directly recursive. This means that we can have equations of the form: $\langle C(\bar{x}) = \mathsf{exp} + D(\bar{y}), \varphi_C\rangle$ and $\langle D(\bar{x}) = \mathsf{exp} + C(\bar{y}), \varphi_D\rangle$ with $D \neq C$. As D has not been unfolded, then it must happen that $D \in \mathsf{btc}$. We have that C is in the same *SCC* as D. Then, by condition (**DR**) of Definition 10, it must happen that $C = D$. This contradicts the initial assumption. Claim 2 follows from the condition (**TR**) of Definition 10 by reasoning by contradiction. Let us assume that $\mathcal{S}$' is not obtained in finite time. This can only happen because Unfold does not terminate. Hence, there exists an infinite derivation $\langle E_1, \varphi_1\rangle \overset{unf}{=} \langle E_2, \varphi_2\rangle \overset{unf}{=} \ldots \overset{unf}{=} \langle E_n, \varphi_n\rangle \overset{unf}{=} \langle E_{n+1}, \varphi_{n+1}\rangle \overset{unf}{=} \ldots$. Since the number of cost relations in $rel(\mathcal{S})$ is finite and the sequence is infinite, there is a cycle from some E_i to an E_n for $i < n$. By condition (**TR**), this cannot happen because there must exist an E_j in the cycle with $i \leq j \leq n$ that belongs to btc. □

The following lemma guarantees that PE preserves the solutions of *CRSs*. The proof basically consists in ensuring the correctness of the basic operators in the partial evaluation algorithm of Definition 12 to, then, rely on the classical correctness results of PE proven in the context of logic programming (see e.g. [33, 34, 39] and more recent formulations like [37, 38]).

Lemma 7 *(correctness of PE) Let $\mathcal{S}$ be a CRS, C be a relation, and let* btc *be a BTC for $\mathcal{S}$. Let $\mathcal{S}'$ be the partial evaluation of C in $\mathcal{S}$ w.r.t.* btc*. Then, $\forall \bar{v} \in \mathbb{Z}^n$, $\forall r \in \mathbb{R}_+$ we have that $r \in Answers(C(\bar{v}), \mathcal{S})$ iff $r \in Answers(C(\bar{v}), \mathcal{S}')$.*

Proof (sketch) The proof can be done by demonstrating that Definition 12 is a *correct* partial evaluation as defined in logic programming. Correctness results were already stated in Theorem 1 of [34] and more recent formulations appear in [37, 38]. In all cases, correctness requires proving:

1. *Soundness.* The soundness condition ensures that the all answers in the partially evaluated program are also answers in the original program. It is proven by demonstrating that each unfolding step in the partially evaluated program corresponds to a sequence of equivalent steps in the original one. In our context, it amounts to ensuring that the operator Unfold of Definition 11 preserves the answers.
2. *Completeness.* Completeness guarantees that all answers in the original program are also found in the partially evaluated one. It can be ensured when the set of terms to be partially evaluated meets the so-called *closedness* condition [39]. The role of this condition is to ensure that all possible calls that raise during the execution of a *CRS* will find a matching relation. In our context, we need to ensure that the set btc enforces the closedness condition, i.e., answers are not lost.

Point 1 requires to prove the correctness of operator Unfold of Definition 11. It indeed trivially holds as Unfold simply replaces in rule **(unf)** a function call by its right hand side, with the corresponding propagation of constraints. In terms of evaluation trees, this step basically merges a node with (some of) its successors.

The closedness of the terms to be partially evaluated, i.e., the elements in the set btc, follows from the fact that only terms in btc remain in the relation and the remaining ones are unfolded. This trivially ensures that all possible calls during execution will be covered by btc, as required by point 2 above. In standard PE, correctness requires that the partial evaluation process terminates. This is ensured by Lemma 6. □

9 Incompleteness in Cost Analysis

When we consider the whole cost analysis which comprises the two phases mentioned in Section 1, i.e., obtaining a closed-form upper bound from a program—instead of from a *CRS*—the problem is strictly more difficult than proving termination. This is explained by the fact that obtaining a closed-form upper bound of a program which has a non-zero cost expression associated to each recursive equation implies the termination of the program from which the *CR* has been generated. Therefore, the approach is necessarily incomplete and might fail to produce an upper bound. Clearly, this may occur because the resource usage of the program is actually infinite w.r.t. the cost model used. For instance, a non-terminating program that can perform an infinite number of steps. When the resource consumption is finite, we can still fail to produce an upper bound because of loss of precision in one of the two phases in the cost analysis. This can occur in the first phase, i.e., when the program is transformed into the *CRS* since it applies abstract interpretation based analyses in order to approximate undecidable problems such as aliasing and size relations. However, the incompleteness in the first part of the analysis is completely outside the scope of this paper and we refer to [6] for further details.

Certainly, the second part of cost analysis is undecidable as well, i.e., if a given cost relation admits a closed-form upper bound, so we must accept certain restrictions. In [16], it is proven that a simpler problem, namely the termination of a special case of *CRS* where all equations have at most one call in the body and constraints are of the form $x - y \leq c$, is undecidable. A detailed discussion about decidability of simple loops with integer constraints can be found in [20]. There are three sources of incompleteness in our approach, i.e., in the process of obtaining an upper bound from a *CRS* by using our techniques.

1. The first one is obtaining directly recursive *CRs*. For instance, the following *CRS* does not have a cover point:

$$\langle C(n) = C(n') + D(n'), \{n > 0, n'{=}n{-}1\}\rangle$$
$$\langle D(n) = D(n') + C(n'), \{n > 0, n'{=}n{-}1\}\rangle$$

   Importantly, this phase is complete for *CRs* extracted from structured loops and from the recursive patterns found in most programs. The use of features like `break` and `continue` in languages like Java or C have do not pose any problem, since the control flow graph of the program can be constructed and the program can thus be turned into recursive form. As it can be seen in the example, incompleteness might occur in certain types of mutually recursive relations.
2. The second source of incompleteness in our method is in finding ranking functions. Currently, we use a complete procedure for inferring linear ranking functions [45]. However, there are *CRSs* which do not have a linear ranking function as explained in Example 6. Integrating other more sophisticated ranking functions is possible, but it is probably not required in practice.
3. The third one is finding useful invariants. Sometimes this is not possible by using linear constraints. This happens for example in this example:

$$\langle C(n, m) = m, \{n{=}0\}\rangle$$
$$\langle C(n, m) = C(n', m'), \{n'{=}n{-}1, m'{=}2{*}m, n{>}0\}\rangle$$

   The value of m in the base case will be $(2^n) * m_0$. In principle, we could use methods for inferring polynomial invariants, although we would need a different maximization procedure.

10 Experimental Evaluation

In order to evaluate the practicality of our approach, we have developed a system that we call PUBS (*Practical Upper Bounds Solver*), which implements the ideas presented in this paper. PUBS is implemented in Prolog and uses the Parma Polyhedra Library [13] for manipulating linear constraints. We have conducted a number of experiments which aim at evaluating the applicability of our approach, the quality of the upper bounds obtained, and the efficiency and scalability of the system.

In order to test our system on realistic *CRs* produced by automatic cost analysis, we have used as benchmarks in our experiments a set of *CRs* automatically generated by the cost analyzer of Java bytecode described in [6], using several cost models. The Java bytecode programs taken as input cover a wide range of complexity classes and are the result of compiling the corresponding Java source programs. Both the Java source code and the produced *CRs* for such programs are available at the PUBS web interface at http://costa.ls.fi.upm.es/pubs, from where PUBS can be run on such *CRs* and also on *CRs* provided by the user.

Now we briefly describe the programs considered, which are listed in increasing complexity order and range from constant to exponential complexity, going through polynomial and divide and conquer. Polynomial is a method for copying polynomials and has a constant upper bound (on memory consumption). DivByTwo is a loop which iterates a logarithmic number of times, as its counter is decremented by half in each iteration. ArrayReverse produces a reversed copy of an array of integers. Concat concatenates two arrays of integers into a new array. Incr has a loop which iterates a linear number of times that depends on the run-time type of an input argument. ListReverse is an in-place reversal of a list represented as a linked list. MergeList merges two sorted lists implemented as linked lists. Power recursively computes the power operation. Cons copies a linked list. MergeSort sorts an array using the Merge Sort algorithm. EvenDigits is a simple `for` loop with a call to the DivbyTwo method inside the loop body. ListInter computes the intersection of two unsorted linked lists. SelectSort sorts an array by Selection Sort. FactSum adds up the factorial of all naturals from 0 to the input value n. Delete is the running example in Fig. 1. MatMult multiplies two matrices. Hanoi has a doubly recursive structure, as the well-known Towers of Hanoi problem. Fibonacci is a naive doubly recursive implementation of Fibonacci numbers. Finally, BST is a method for recursively copying a binary search tree. In addition, in the experiments, we have used three different cost models:

– The heap consumption (in bytes), in those benchmarks marked with “$*$”,
– The number of executed comparison instructions, in the benchmark marked with “n”, and
– The number of executed bytecode instructions, in the rest of benchmarks.

10.1 Accuracy of the Upper-Bounds Obtained

The first set of experiments performed aims at evaluating the applicability of PUBS and the accuracy of the closed-form upper bounds thus obtained. Table 1 shows the upper bounds generated by PUBS for the benchmarks described above. The column **Properties** shows the properties of the corresponding *CR*, in such a way that *a*, *b* and *c* indicate, respectively, that the *CR* is non-deterministic, that it has inexact size constraints, and multiple arguments (Section 2.2). As can be seen, most of the benchmarks have one or more of such properties. If we handle the complete semantics of programs, including exceptions, even simple programs such as

ArrayReverse are non-deterministic since accesses to arrays may in principle throw array-out-of-bounds exceptions. As a result, only the purely numerical programs, i.e., Power, FactSum, Hanoi, and Fibonacci are in a format syntactically acceptable by Mathematica® or other CAS. In contrast, PUBS has been able to automatically find upper bounds for all benchmarks considered. This clearly shows that CAS have rather restricted applicability in *CRs* obtained from real programs. Column **Upper Bound** shows the closed-form upper bound obtained by PUBS. As can be seen, they are relatively syntactically simple. This is important since, as already mentioned in [54], one of the problems of cost analysis is that the cost functions produced can grow considerably large. This can hinder the success of cost analysis since large cost functions are hard to understand by humans and also difficult to automatically handle in applications such as resource certification [9], where it is required to compare cost functions [3].

In order to evaluate the accuracy of the upper bounds obtained using our approach, Table 2 compares the values obtained by evaluating the upper bounds generated by PUBS on some concrete input data with the maximum value which can be obtained by evaluating the input *CRs*. Column **Input** indicates the input data considered for each **Benchmark**, i.e., given the entry C for a cost relation $\mathcal{S}$, it provides the particular $C(\bar{v})$ used for evaluating both the upper bound and the associated cost relation.

Then, column **Estimated** provides the value obtained by evaluating the upper bound computed by PUBS on the given input data. Column **Actual** provides the actual value obtained by evaluating the cost-bound function discussed in Section 3, which is defined as $C_+(\bar{v}) = \max(\mathit{Answers}(C(\bar{v}), \mathcal{S}))$. For this we have implemented an evaluator for *CRSs* which given a *CRS* $\mathcal{S}$ and an initial call $C(\bar{v})$ produces all answers corresponding to all evaluation trees for $C(\bar{v})$ in $\mathcal{S}$ and then obtains the maximum of them. The evaluator has been implemented in *Constraint Logic*

Table 2 Estimated versus actual maximal value

Benchmark	Input	Estimated	Actual	Accuracy
Polynomial*	copy_pol(10)	216	216	100
DivByTwo	divByTwo(10)	49	38	76
ArrayReverse	arrayReverse(10)	152	152	100
Concat	concat(10,10)	245	245	100
Incr	add(10,10)	218	218	100
ListReverse	listReverse(10)	138	138	100
MergeList	merge(5,5)	316	279	88
Power	power(10)	104	104	100
Cons*	cons(10)	222	222	100
MergeSort^n	ms_sort(_,_,0,5)	52	32	62
EvenDigits	evenDigits(10)	462	345	75
ListInter	listInter(5,5)	486	486	100
SelectSort	selectSort(6)	417	315	76
FactSum	doSum(10)	1,172	677	58
Delete	delete(3,_,_,3,_,3)	297	256	86
MatMult	multiply(3,3)	866	866	100
Hanoi	hanoi(10)	20,463	20,463	100
Fibonacci	fibonacciMethod(10)	9,203	1,589	17
BST*	copy(4)	180	132	73

Programming [32] in order to efficiently handle the size constraints which are accumulated when obtaining the evaluation trees. It is important to note that due to the highly non-deterministic nature of many of the *CRs*, this evaluation often results in a combinatorial explosion which makes evaluation of most *CRs* unfeasible except for very small input values. This is why in some cases the input values are smaller than 10, which was the originally attempted input value for all arguments. We also use underscore to indicate arguments which do not affect the evaluation of the *CR*.

Finally, the column **Accuracy** tries to provide an indication of the accuracy obtained by showing the value **Actual**/**Estimated** $\times$ 100. Correctness of the upper bounds computed requires that **Actual** $\leq$ **Estimated**, which occurs in all cases. Also, this implies that **Accuracy** is a number between 0 and 100, with a 100 indicating that the upper bound computed by PUBS is exact. As can be seen, PUBS obtains the exact upper bound in a good number of cases. Then there is a group of programs for which the accuracy obtained ranges from 58% to 88% which we argue is quite good for many applications. The main reason for loss of precision in these benchmarks is the occurrence of loops (or recursion) whose body contains computations with cost which is different in different iterations, since our approach will take the worst case cost for such computation and multiply it by the number of iterations. Though this precision loss accumulates with the depth of nesting, it is important to note that it does not accumulate with the length of programs. Also, this precision loss does not occur if the cost of inner computations is the same in all iterations. This is why we obtain full accuracy for MatMult, even though it has three nested loops.

There are, however, some cases where accuracy is low, such as Fibonacci, where our approach is able to find an upper bound, but its accuracy is 17%. In contrast, this *CR* can be solved in Mathematica® and obtain an exact upper bound. However, such upper bound is syntactically rather complex: $-(2^{3-x}(15^{1+x} - 19(1-\sqrt{5})^x + 5\sqrt{5}(1-\sqrt{5})^x - 19(1+\sqrt{5})^x - 5\sqrt{5}(1+\sqrt{5})^x))/((-1+\sqrt{5})^2(1+\sqrt{5})^2)$. The fact that it is more complex makes it more difficult to use it for the applications discussed in Section 1.1 and in some cases it is preferable to use a simpler, though less accurate, upper bound, such as the one obtained by PUBS. Note also that the benchmark MergeSort falls into the class of divide-and-conquer programs explained in Section 7 where, by using the level-count approach, we obtain the accurate closed-form shown in the Table 1.

Also, we argue that using CAS for obtaining upper bounds of realistic *CRs* is not an option. In fact, it was our own previous experience in trying to obtain upper bounds with Mathematica®, in the work reported in [8], which motivated this work. There, we obtained upper bounds for a subset of the benchmarks considered in this paper, but only after significant human intervention in order to convert the *CRs* into a format solvable in Mathematica®, since it has several restrictions that *CRs* do not satisfy, namely, (1) we cannot include guards, (2) variables cannot be repeated in the equation head, (3) all equations must have at least one variable argument and (4) variables in the equation head must appear in the body.

10.2 Efficiency and Scalability of the Approach

Table 3 aims at studying the efficiency of our system by showing the results of two different experiments. In the first experiment, we analyze each of the benchmarks in isolation. Column **#**$_{eq}$ shows the number of equations before PE (in brackets after

Table 3 Scalability of upper bounds inference

Benchmark	$\#_{eq}$	T	$\#^c_{eq}$	T_{pe}	T_{ub}	Rat.
Polynomial*	23 (3)	10	385 (97)	388	1,190	4.1
DivByTwo	9 (3)	2	362 (94)	402	1,173	4.3
ArrayReverse	9 (3)	2	344 (88)	387	1,122	4.4
Concat	14 (5)	10	335 (85)	386	1,102	4.4
Incr	28 (5)	23	321 (80)	384	1,046	4.5
ListReverse	9 (3)	4	293 (75)	374	943	4.5
MergeList	21 (4)	17	284 (72)	374	925	4.6
Power	8 (2)	2	262 (67)	366	898	4.8
Cons*	22 (2)	6	253 (64)	376	912	5.1
MergeSort^n	39 (12)	499	230 (61)	354	805	5.0
EvenDigits	18 (5)	7	191 (49)	130	290	2.2
ListInter	37 (9)	48	173 (44)	126	246	2.2
SelectSort	19 (6)	22	136 (35)	115	169	2.1
FactSum	17 (5)	8	117 (29)	109	143	2.2
Delete	33 (9)	106	100 (24)	102	130	2.3
MatMult	19 (7)	17	67 (15)	69	34	1.5
Hanoi	9 (2)	5	48 (8)	67	16	1.7
Fibonacci	8 (2)	4	39 (6)	63	11	1.9
BST*	31 (4)	36	31 (4)	64	8	2.3

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 Quad Q9300 at 2.50GHz with 1.95GB of RAM, running Linux 2.6.24-21. We argue that analysis times are acceptable. In the case of MergeSort analysis time is higher because its equations contain a large number of variables when compared to those of the other examples. This affects the efficiency when computing the ranking function and also when maximizing expressions.

The second experiment aims at studying how analysis time increases when larger *CRs* are used as benchmarks, i.e., the scalability of our approach. In order to do so, we have connected together the *CRs* for the different benchmarks by introducing a call from each *CR* to the one appearing immediately below it in the table. Such call is always introduced in a recursive equation. The results of this second experiment are shown in the last four columns of the table. 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 when we analyze BST in the second experiment we have the same number of equations as in the first experiment. Then, for Fibonacci we have its eight equations plus 31 which have been previously accumulated. 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 a closed-form upper bound of a *CRS* with at least 20 nested loops and 385 equations. In this experiment, the analysis time is split into $\mathbf{T}_{pe}$ and $\mathbf{T}_{ub}$, where $\mathbf{T}_{pe}$ is the time of PE and $\mathbf{T}_{ub}$ is the time of all other phases. The results show that even though PE is a *global* transformation, its time efficiency is linear with the number of equations, since PE operates on strongly connected components. Our system solves 385 equations in $388 + 1{,}190ms$.

Finally, column **Rat**. shows the total time per equation. The ratio is quite small from BST to EvenDigits, which are the simplest benchmarks and also have few equations. It increases notably when we analyze the benchmark MergeSort because, as discussed above, its equations have a large number of variables. The important point is that for larger *CRs* (from MergeSort upwards) this ratio decreases more and more as we connect new benchmarks. It should be observed that it decreases even if the size of the *CRs* increases and also the equations have to count more complex expressions. This happens because the new benchmarks which are connected are simpler than MergeSort in terms of the number of variables. We believe that this demonstrates that our approach is 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.

Pubs is already integrated within the COst and Termination Analyzer for Java bytecode, Costa [7]. If one wants to obtain closed-form upper bounds from Java (bytecode) programs rather than from the cost relations, the Costa system can be used online at: http://costa.ls.fi.upm.es/costa.

In summary, we argue that our experimental results show that, for many common programs, our approach provides reasonably accurate results which are syntactically simple and in an acceptable amount of analysis time.

11 Related Work

As already mentioned in Section 1, the classical approach to automatic cost analysis, which dates back to the seminal work of [54] consists of two phases. In the first phase, given a program and a cost model, static analysis produces what we call a *cost relation* (*CR*), which is a set of recursive equations which capture the cost of our program in terms of the size of its input data. The fact that *CRs* are recursive make them not very useful for most applications of cost analysis. Therefore, a second phase is required to obtain a non-recursive representation of such *CRs*, 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.

There are a number of cost analyses available which are based on building *CRs* and which can handle a range of programming languages, including functional [18, 36, 40, 47, 49, 53, 54], logic [26, 42], and imperative [6]. Such *CRs* must ensure that, for any valid input integer tuple, a value which is guaranteed to be an upper bound of the execution cost of the program for any input data in the (usually infinite) set of values which are consistent with the input sizes. There is no unified terminology in this area and such cost relations are referred to as *worst-case complexity functions* in [1], as *time-bound functions* in [47], and *recursive time-complexity functions* in [36]. Apart from syntactic differences, the main differences between such forms of functions and our cost relations are twofold: (1) our equations contain associated size constraints and (2) we consider (possibly) non-deterministic relations. Both features are necessary to perform cost analysis of realistic languages (see Section 2.2). While in all such analyses the first phase, i.e., producing *CRs* is studied in detail, the second phase, i.e., obtaining closed-form upper bounds for them, has received comparatively less attention.

There are two main ways of viewing *CRs* which lead to different mechanisms for finding closed-form upper bounds. We call the first view *algebraic* and the second view *transformational*. The algebraic one is based on regarding *CRs* as *recurrence relations*. This view was the first one to be proposed and it is the one which is advocated for in a larger number of works. It allows reusing the large existing body of work in solving recurrence relations. Within this view, two alternatives have been used in previous analyzers. One alternative consists in implementing restricted recurrence solvers within the analyzer based on standard mathematical techniques, as done in [26, 54]. The other alternative, motivated by the availability of powerful *computer algebra systems* (CASs for short) such as Mathematica®, MAXIMA, MAPLE, etc., consists in connecting the analyzer with an external solver, as proposed in [6, 18, 40, 49, 53].

The transformational view consists in regarding *CRs* as (functional) programs. In this view, closed-form upper bounds are produced by applying (general-purpose) program transformation techniques on the *time-bound program* [47] until a non-recursive program is obtained. Note that, as discussed in Section 2, it is straightforward to obtain time-bound programs from *CRs* by introducing a maximization operator (or disjunctive execution). The transformational view was first proposed in the ACE system [36], which contained a large number of program transformation rules aimed at obtaining non-recursive representations. It was also advocated by Rosendahl in [47], who later in [48] provided a series of program transformation techniques based on super-compilation [52] which were able to obtain closed-forms for some classes of programs.

The problem with all the approaches mentioned above is that, though they can be successfully applied for obtaining closed-forms for *CRs* generated from simple programs, they do not fulfill the initial expectations in that they are not of general applicability to *CRs* generated from real programs. The essential features which neither the algebraic nor the transformational approaches can handle are discussed in Section 2.2. The main motivation for this work was our own experience in trying to apply the algebraic approach on the *CRs* generated by [6]. 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. In contrast, our approach can produce correct and comparatively simple results even in the presence of non-determinism.

The need for improved mechanisms for automatically obtaining closed-form upper bounds was already pointed out in Hickey and Cohen [30]. A significant work in this direction is PURRS [14], which has been the first system to provide, in a fully automatic way, non-asymptotic closed-form upper and lower bounds for a wide class of recurrences. Unfortunately, and unlike our proposal, it also requires *CRs* to be deterministic. Another relevant work is that of Marion et. al. [19, 41], who propose an analysis for stack frame size in first order functional programming. They use quasi-interpretations, which are different from ranking functions and the whole approach is limited to polynomial bounds.

An altogether different approach to cost analysis is based on type systems with resource annotations, which does not use *CRs* as an intermediate step. Thus, this approach does not require computing closed-form upper bounds for *CRs*, but it is often restricted to linear bounds [31], with some notable exception like [25].

A program analysis based approach for inferring *polynomial* boundedness of computed values (as a function of the input) has been recently proposed in [17]. It infers the complexity of a given program by first obtaining a step-counting program. This work builds on similar previous works along the lines of [35, 44], and the main novelty here is that it provides completeness for a simple (Turing incomplete) language. Compared to this line of research, our approach is more powerful in that it is not limited to polynomial complexity but, on the other hand, the techniques we use are inherently incomplete.

12 Conclusions

We have proposed an approach to the automatic inference of non-asymptotic closed-form upper bounds of *CRs* produced by automatic cost analysis. For this, we have formally defined *CRs* as a target language for cost analysis. Hence, our method for closed-form upper bound inference can be used in static cost analysis of any programming language. In spite of the inherent incompleteness, we have experimentally shown that our approach is able to obtain useful upper bounds for a large class of common programs. In summary, the use of ranking functions and our practical method to compute upper bounds for a very general notion of cost expression (including exponential, logarithmic, etc.) allows obtaining closed-form upper bounds for realistic *CRs* with possibly non-deterministic equations, multiple arguments, and inexact size constraints.

In recent work [11], we have applied our method to obtain closed-form upper bounds from non-standard *CRs*, namely from *CRs* which capture the *heap space usage* of programs by taking into account the deallocations performed by garbage collection, without requiring any change to the techniques presented in this paper. The way in which cost relations are generated is different from the standard approach because the live heap space is not an accumulative resource of a program's execution but, instead, it requires to reason on all possible states to obtain their maximum. As a result, cost relations include non-deterministic equations which capture the different peak heap usages reached along the execution. Importantly, the additional non-determinism does not pose any problem to the applicability of our method.

Acknowledgements We gratefully thank the anonymous referees for many useful comments and suggestions that greatly helped to improve this article. This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, by the Spanish Ministry of Science and Innovation (MICINN) under the TIN-2008-05624 *DOVES* project, the TIN2008-04473-E (Acción Especial) project, the HI2008-0153 (Acción Integrada) project, the UCM-BSCH-GR58/08-910502 Research Group and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS* project.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading (1974)
2. Albert, E., Arenas, P., Codish, M., Genaim, S., Puebla, G., Zanardini, D.: Termination analysis of Java bytecode. In: 10th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'08). Lecture Notes in Computer Science, vol. 5051, pp. 2–18. Springer, Heidelberg (2008)

3. Albert, E., Arenas, P., Genaim, S., Herraiz, I., Puebla, G.: Comparing cost functions in resource analysis. In: 1st International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA'09). Lecture Notes in Computer Science. Springer, Heidelberg (2009)
4. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Automatic inference of upper bounds for recurrence relations in cost analysis. In: 15th International Symposium on Static Analysis (SAS'08). Lecture Notes in Computer Science, vol. 5079, pp. 221–237 (2008)
5. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Cost relation systems: a language–independent target language for cost analysis. In: 8th Spanish Conference on Programming and Computer Languages (PROLE'08), vol. 17615 of Electronic Notes in Theoretical Computer Science. Elsevier (2008)
6. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of Java bytecode. In: 16th European Symposium on Programming, (ESOP'07). Lecture Notes in Computer Science, vol. 4421, pp. 157–172. Springer (2007)
7. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: COSTA: design and implementation of a cost and termination analyzer for Java bytecode. In: 6th International Symposioum on Formal Methods for Components and Objects (FMCO'08). Lecture Notes in Computer Science, no. 5382, pp. 113–133. Springer (2007)
8. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Experiments in cost analysis of Java bytecode. In 2nd Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'07), vol. 190, Issue 1 of Electronic Notes in Theoretical Computer Science. Elsevier (2007)
9. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Resource usage analysis and its application to resource certification. In: 9th International School on Foundations of Security Analysis and Design (FOSAD'09). Lecture Notes in Computer Science, no. 5705, pp. 258–288. Springer (2009)
10. Albert, E., Genaim, S., Gómez-Zamalloa, M.: Heap space analysis of Java bytecode. In: 6th International Symposium on Memory Management (ISMM'07), pp. 105–116. ACM Press (2007)
11. Albert, E., Genaim, S., Gómez-Zamalloa, M.: Live Heap space analysis for languages with garbage collection. In: 8th International Symposium on Memory management (ISMM'09). ACM Press (2009)
12. Aspinall, D., Gilmore, S., Hofmann, M., Sannella, D., Stark, I.: Mobile resource guarantees for smart devices. In: Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS'04). Lecture Notes in Computer Science, vol. 3362, pp. 1–27. Springer (2005)
13. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library: toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Sci. Comput. Program. **72**(1–2), 3–21 (2008)
14. Bagnara, R., Pescetti, A., Zaccagnini, A., Zaffanella, E.: PURRS: towards computer algebra support for fully automatic worst-case complexity analysis. Technical report (2005). arXiv:cs/0512056
15. Batchelder, P.M.: An introduction to linear difference equations. Dover Publications (1967)
16. Ben-Amram, A.M.: Size-change termination with difference constraints. ACM Trans. Program. Lang. Syst. **30**(3), 31 (2008) (Article 16)
17. Ben-Amram, A.M., Jones, N.D., Kristiansen L.: Linear, polynomial or exponential? Complexity inference in polynomial time. In: Logic and Theory of Algorithms, 4th Conference on Computability in Europe, (CiE'08). Lecture Notes in Computer Science, vol. 5028, pp. 67–76. Springer (2008)
18. Benzinger, R.: Automated higher-order complexity analysis. Theor. Comp. Sci. **318**(1–2), 79–103 (2004)
19. Bonfante, G., Marion, J.-Y., Moyen, J.-Y.: Quasi-interpretations and small space bounds. In: 16th International Conference on Rewriting Techniques and Applications (RTA'05). Lecture Notes in Computer Science, vol. 3467, pp. 150–164 (2005)
20. Braverman, M.: Termination of integer linear programs. In: 18th Computer Aided Verification (CAV'06). Lecture Notes in Computer Science, vol. 4144, pp. 372–385. Springer (2006)
21. Chander, A., Espinosa, D., Islam, N., Lee, P., Necula, G.: Enforcing resource bounds via static verification of dynamic checks. In: 14th European Symposium on Programming (ESOP'05). Lecture Notes in Computer Science, vol. 3444, pp. 311–325. Springer (2005)
22. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: ACM Symposium on Principles of Programming Languages (POPL'77), pp. 238–252. ACM Press (1977)

23. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: ACM Symposium on Principles of Programming Languages (POPL'78), pp. 84–97. ACM Press (1978)
24. Craig, S.-J., Leuschel, M.: Self-tuning resource aware specialisation for prolog. In: 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'05), pp. 23–34. ACM Press (2005)
25. Crary, K., Weirich, S.: Resource bound certification. In: 27th ACM Symposium on Principles of Programming Languages (POPL'05), pp. 184–198. ACM Press (2000)
26. Debray, S.K., Lin, N.W.: Cost Analysis of Logic Programs. ACM Trans. Program. Lang. Syst. **15**(5), 826–875 (1993)
27. Floyd, R.W.: Assigning meanings to programs. In: Proceedings of Symposium in Applied Mathematics, vol. 19, Mathematical Aspects of Computer Science, pp. 19–32. American Mathematical Society, Providence, RI (1967)
28. Gómez, G., Liu, Y.A.: Automatic time-bound analysis for a higher-order language. In: Proceedings of the ACM SIGPLAN 2002 Workshop on Partial Evaluation and Semantics-Based Program Manipulation, pp. 75–88. ACM Press (2002)
29. Hermenegildo, M., Puebla, G., Bueno, F., López-García, P.: Integrated program debugging, verification, and optimization using abstract interpretation (and The Ciao System Preprocessor). Sci. Comput. Program. **58**(1–2), 115–140 (2005)
30. Hickey, T., Cohen, J.: Automating program analysis. J. ACM **35**(1), 185–220 (1988)
31. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. In: 30th Symposium on Principles of Programming Languages (POPL'03), pp. 185–197. ACM Press, New York (2003)
32. Jaffar, J., Maher, M.J.: Constraint logic programming: a survey. J. Log. Program. **19**/**20**, 503–581 (1994)
33. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial evaluation and automatic program generation. Prentice Hall, New York (1993)
34. Komorovski, J.: An introduction to partial deduction. In: Meta Programming in Logic (META'92). Lecture Notes in Computer Science, vol. 649, pp. 49–69. Springer, Heidelberg (1992)
35. Kristiansen, L., Jones, N.D.: The flow of data and the complexity of algorithms. In: 1st Conference on Computability in Europe (CiE'05). Lecture Notes in Computer Science, vol. 3526, pp. 263–274 (2005)
36. Le Metayer, D.: ACE: an automatic complexity evaluator. ACM Trans. Program. Lang. Syst. **10**(2), 248–266 (1988)
37. Leuschel, M.: A framework for the integration of partial evaluation and abstract interpretation of logic programs. ACM Trans. Program. Lang. Syst. **26**(3), 413–463 (2004)
38. Leuschel, M., Bruynooghe, M.: Logic program specialisation through partial deduction: control issues. Theory Pract. Log. Program. **2**(4 & 5), 461–515 (2002)
39. Lloyd, J.W., Shepherdson, J.C.: Partial evaluation in logic programming. J. Log. Program. **11**(3–4), 217–242 (1991)
40. Luca, B., Andrei, S., Anderson, H., Khoo, S.-C.: Program transformation by solving recurrences. In: ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '06), pp. 121–129. ACM (2006)
41. Marion, J.-Y., Péchoux, R.: Sup-interpretations, a semantic method for static analysis of program resources. ACM Trans. Comput. Log. **10**(4), 31 (2009) (Article 27)
42. Navas, J., Mera, E., López-García, P., Hermenegildo, M.: User-definable resource bounds analysis for logic programs. In: 23rd International Conference on Logic Programming (ICLP'07), vol. 4670 of LNCS, pp. 348–363. Springer, Heidelberg (2007)
43. Necula, G.: Proof-carrying code. In: ACM Symposium on Principles of Programming Languages (POPL 1997), pp. 106–119. ACM Press, New York (1997)
44. Niggl, K.-H., Wunderlich, H.: Certifying polynomial time and linear/polynomial space for imperative programs. SIAM J. Comput. **35**(5), 1122–1147 (2006)
45. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04). Lecture Notes in Computer Science, pp. 239–251. Springer, Heidelberg (2004)
46. Puebla, G., Ochoa, C.: Poly-controlled partial evaluation. In: 8th ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'06), pp. 261–271. ACM Press, New York (2006)

47. Rosendahl, M.: Automatic complexity analysis. In: 4th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'89), pp. 144–156. ACM Press (1989)
48. Rosendahl, M.: Simple driving techniques. In: Mogensen, T., Schmidt, D., Hal Sudborough, I. (eds.) The Essence of Computation. Lecture Notes in Computer Science, vol. 2566, pp. 404–419. Springer, Heidelberg (2002)
49. Sands, D.: A naïve time analysis and its theory of cost equivalence. J. Log. Comput. **5**(4), 495–541 (1995)
50. Shamir, A.: A linear time algorithm for finding minimum cutsets in reducible graphs. SIAM J. Comput. **8**(4), 645–655 (1979)
51. Spoto, F., Hill, P.M., Payet, E.: Path-length analysis of object-oriented programs. In: 1st International Workshop on Emerging Applications of Abstract Interpretation (EAAI'06), Electronic Notes in Theoretical Computer Science. Elsevier, Amsterdam (2006)
52. Turchin, V.F.: The concept of a supercompiler. ACM Trans. Program. Lang. Syst. **8**(3), 292–325 (1986)
53. Wadler, P.: Strictness analysis aids time analysis. In: ACM Symposium on Principles of Programming Languages (POPL'88), pp. 119–132. ACM Press (1988)
54. Wegbreit, B.: Mechanical program analysis. Commun. ACM **18**(9), 69–73 (1975)