Comparing Cost Functions in Resource Analysis

E. Albert[1], P. Arenas[1], S. Genaim[1], I. Herraiz[1] and G. Puebla[2]

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

Abstract. *Cost functions* provide information about the amount of resources required to execute a program in terms of the sizes of input arguments. They can provide an upper-bound, a lower-bound, or the average-case cost. Motivated by the existence of a number of automatic cost analyzers which produce cost functions, we propose an approach for automatically proving that a cost function is smaller than another one. In all applications of resource analysis, such as resource-usage verification, program synthesis and optimization, etc., it is essential to compare cost functions. This allows choosing an implementation with smaller cost or guaranteeing that the given resource-usage bounds are preserved. Unfortunately, automatically generated cost functions for realistic programs tend to be rather intricate, defined by multiple cases, involving non-linear subexpressions (e.g., exponential, polynomial and logarithmic) and they can contain multiple variables, possibly related by means of constraints. Thus, comparing cost functions is far from trivial. Our approach first syntactically transforms functions into simpler forms and then applies a number of sufficient conditions which guarantee that a set of expressions is smaller than another expression. Our preliminary implementation in the COSTA system indicates that the approach can be useful in practice.

1 Introduction

Cost analysis [12,6] aims at statically predicting the resource consumption of programs. Given a program, cost analysis produces a *cost function* which approximates the resource consumption of the program in terms of the input data sizes. This approximation can be in the form of an upper-bound, a lower-bound, or the average-case resource consumption, depending on the particular analysis and the target application. For instance, upper bounds are required to ensure that a program can run within the resources available; lower bounds are useful for scheduling distributed computations. The seminal cost analysis framework by Wegbreit [12] was already generic on the notion of *cost model*, e.g., it can be used to measure different resources, such as the number of instructions executed, the memory allocated, the number of calls to a certain method, etc. Thus, cost functions can be used to predict any of such resources.

In all applications of resource analysis, such as resource-usage verification, program synthesis and optimization, etc., it is necessary to compare cost functions. This allows choosing an implementation with smaller cost or to guarantee that the given resource-usage bounds are preserved. Essentially, given a method

m, a cost function f_m and a set of linear constraints ϕ_m which impose size restrictions (e.g., that a variable in m is larger than a certain value or that the size of an array is non zero, etc.), we aim at comparing it with another cost function bound $\mathtt{b}$ and corresponding size constraints $\phi_{\mathtt{b}}$. Depending on the application, such functions can be automatically inferred by a resource analyzer (e.g., if we want to choose between two implementations), one of them can be user-defined (e.g., in resource usage verification one tries to verify, i.e., prove or disprove, *assertions* written by the user about the efficiency of the program).

From a mathematical perspective, the problem of cost function comparison is analogous to the problem of proving that the difference of both functions is a decreasing or increasing function, e.g., $\mathtt{b} - f_m \geq 0$ in the context $\phi_{\mathtt{b}} \wedge \phi_m$. This is undecidable and also non-trivial, as cost functions involve non-linear subexpressions (e.g., exponential, polynomial and logarithmic subexpressions) and they can contain multiple variables possibly related by means of constraints in $\phi_{\mathtt{b}}$ and ϕ_m. In order to develop a practical approach to the comparison of cost functions, we take advantage of the form that cost functions originating from the analysis of programs have and of the fact that they evaluate to non-negative values. Essentially, our technique consists in the following steps:

1. Normalizing cost functions to a form which make them amenable to be syntactically compared, e.g., this step includes transforming them to sums of products of basic cost expressions.
2. Defining a series of comparison rules for basic cost expressions and their (approximated) differences, which then allow us to compare two products.
3. Providing sufficient conditions for comparing two sums of products by relying on the product comparison, and enhancing it with a *composite* comparison schema which establishes when a product is larger than a sum of products.

We have implemented our technique in the COSTA system [3], a COSt and Termination Analyzer for Java bytecode. Our experimental results demonstrate that our approach works well in practice, it can deal with cost functions obtained from realistic programs and verifies user-provided upper bounds efficiently.

The rest of the paper is organized as follows. The next section introduces the notion of cost bound function in a generic way. Sect. 3 presents the problem of comparing cost functions and relates it to the problem of checking the inclusion of functions. In Sect. 4, we introduce our approach to prove the inclusion of one cost function into another. Section 5 describes our implementation and how it can be used online. In Sect. 6, we conclude by overviewing other approaches and related work.

2 Cost Functions

Let us introduce some notation. The sets of natural, integer, real, non-zero natural and non-negative real values are denoted by $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, $\mathbb{N}^+$, and $\mathbb{R}^+$, respectively. We write x, y, and z, to denote variables which range over $\mathbb{Z}$. A *linear* *expression* has the form $v_0 + v_1x_1 + \ldots + v_nx_n$, where $v_i \in \mathbb{Z}$, $0 \leq i \leq n$. Similarly, a *linear constraint* (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 into equivalent constraints with integer coefficients, e.g., $\frac{1}{2}x > y$ is equivalent to $x > 2y$. The notation $\bar{t}$ stands for a sequence of entities $t_1, \ldots, t_n$, for some $n{>}0$. We write φ, ϕ or ψ, to denote sets of linear constraints which should be interpreted as the conjunction of each element in the set. An assignment σ over a tuple of variables $\bar{x}$ is a mapping from $\bar{x}$ to $\mathbb{Z}$. We write $\sigma \models \varphi$ to denote that $\sigma(\varphi)$ is satisfiable.

The following definition presents our notion of *cost expression*, which characterizes syntactically the kind of expressions we deal with.

Definition 1 (cost expression). Cost expressions *are symbolic expressions which can be generated using this grammar:*

$e ::= n \mid \mathsf{nat}(l) \mid e + e \mid e * e \mid \log_a(\mathsf{nat}(l) + 1) \mid \mathsf{nat}(l)^n \mid a^{\mathsf{nat}(l)} \mid \mathsf{max}(S)$

where $n, a \in \mathbb{N}^+$ and $a \geq 2$, l is a linear expression, S is a non empty set of cost expressions, $\mathsf{nat} : \mathbb{Z} \rightarrow \mathbb{N}$ *is defined as* $\mathsf{nat}(v)$=$\mathsf{max}(\{v, 0\})$. *Given an assignment σ and a basic cost expression e, $\sigma(e)$ is the result of evaluating e w.r.t. σ.*

Observe that linear expressions are always wrapped by nat, as we will explain below in the example. Logarithmic expressions contain a linear subexpression plus "1" which ensures that they cannot be evaluated to $\log_a(0)$. By ignoring syntactic differences, cost analyzers produce cost expressions in the above form.

It is customary to analyze programs (or methods) w.r.t. some initial *context constraints*. Essentially, given a method $m(\bar{x})$, the considered context constraints φ describe conditions on the (sizes of) initial values of $\bar{x}$. With such information, a cost analyzer outputs a *cost function* $f_m(\bar{x}_s) = \langle e, \varphi \rangle$ where e is a cost expression and $\bar{x}_s$ denotes the data sizes of $\bar{x}$. Thus, f_m is a function of the input data sizes that provides bounds on the resource consumption of executing m for any concrete value of the input data $\bar{x}$ such that their sizes satisfy φ. Note that φ is basically a set of linear constraints over $\bar{x}_s$. We use $\mathcal{CF}$ to denote the set of all possible cost functions. Let us see an example.

Example 1. Figure 1 shows a Java program which we use as running example. It is interesting because it shows the different complexity orders that can be obtained by a cost analyzer. We analyze this program using the COSTA system, and selecting the number of executed *bytecode* instructions as cost model. Each Java instruction is compiled to possibly several corresponding bytecode instructions but, since this is not a concern of this paper, we will skip explanations about the constants in the upper bound function and refer to [2] for details.

Given the context constraint $\{n > 0\}$, the COSTA system outputs the *upper bound* cost function for method m which is shown at the bottom of the figure. Since m contains two recursive calls, the complexity is exponential on n, namely we have a factor $2^{\mathsf{nat}(n)}$. At each recursive call, the method f is invoked and its cost (plus a constant value) is multiplied by $2^{\mathsf{nat}(n)}$. In the code of f, we can observe that the while loop has a logarithmic complexity because the loop

```
void m(int n, int a, int b) {
   if (n > 0) {
      m(n - 1, a, b);
      m(n - 2, a, b);
      f(a, b, n);
   }
   }
```

```
void f(int a, int b, int n) {
   int acc = 0;
   while (n > 0) {
      n = n/2;  acc++;
   }
   for (int i = 0; i < a; i++)
      for (int j = 0; j < b; j++) acc++;
}
```

Upper Bound Cost Function

$$m(n,a,b) = 2^{\mathsf{nat}(n)}*\underbrace{(31+\underbrace{\underbrace{(8*\log(1+\mathsf{nat}(2*n-1))}_{\text{while loop}}+\underbrace{\mathsf{nat}(a)*(10+6*\mathsf{nat}(b))}_{\text{nested loop}}}_{\text{cost of f}}))}_{\text{cost of recursive calls}} + \underbrace{3*2^{\mathsf{nat}(n)}}_{\text{base cases}}$$

Fig. 1. Running example and upper bound obtained by COSTA on the number of executed bytecode instructions.

counter is divided by 2 at each iteration. This cost is accumulated with the cost of the second nested loop, which has a quadratic complexity Finally, the cost introduced by the base cases of m is exponential since, due to the double recursion, there is an exponential number of computations which correspond to base cases. Each such computation requires a maximum of 3 instructions.

The most relevant point in the upper bound is that all variables are wrapped by nat in order to capture that the corresponding cost becomes zero when the expression inside the nat takes a negative value. In the case of $\mathsf{nat}(n)$, the nat is redundant since thanks to the context constraint we know that $n > 0$. However, it is required for variables a and b since, when they take a negative value, the corresponding loops are not executed and thus their costs have to become zero in the formula. Essentially, the use of nat allows having a compact cost function instead of one defined by multiple cases. Some cost analyzers generate cost functions which contain expressions of the form $max(\{Exp, 0\})$, which as mentioned above is equivalent to $\mathsf{nat}(Exp)$. We prefer to keep the max operator separate from the nat operator since that will simplify their handling later. □

3 Comparison of Cost Functions

In this section, we state the problem of comparing two cost functions represented as cost expressions. As we have seen in Ex. 1, a cost function $\langle e, \varphi\rangle$ for a method m is a single cost expression which approximates the cost of any possible execution of m which is consistent with the context constraints φ. This can be done by means of nat subexpressions which encapsulate conditions on the input data sizes in a single cost expression. Besides, cost functions often contain max subexpressions, e.g., $\langle \mathsf{max}(\{\mathsf{nat}(x) * \mathsf{nat}(z), \mathsf{nat}(y) * \mathsf{nat}(z)\}), \mathit{true}\rangle$ which represent the cost of disjunctive branches in the program (e.g., the first sub-expression might correspond to the cost of a then-branch and the second one the cost of the else-branch of a conditional statement).

Though nat and max expressions allow building cost expressions in a compact format, when comparing cost functions it is useful to *expand* cost expressions into sets of simpler expressions which altogether have the same semantics. This, on one hand, allows handling simpler syntactic expressions and, on the other hand, allows exploiting stronger context constraints. This expansion is performed in two steps. In the first one we eliminate all max expressions. In the second one we eliminate all nat expressions. The following definition transforms a cost function into a set of max-free cost functions which cover all possible costs comprised in the original function. We write $e[a \mapsto b]$ to denote the expression obtained from e by replacing all occurrences of subexpression a with b.

Definition 2 (max-free operator). *Let $\langle e, \varphi\rangle$ be a cost function. We define the* max*-free operator $\tau_{\mathsf{max}} : 2^{\mathcal{CF}} \mapsto 2^{\mathcal{CF}}$ as follows: $\tau_{\mathsf{max}}(M) = (M - \{\langle e, \varphi\rangle\}) \cup \{\langle e[\mathsf{max}(S) \mapsto e'], \varphi\rangle, \langle e[\mathsf{max}(S) \mapsto \mathsf{max}(S'), \varphi\rangle\}$, where $\langle e, \varphi\rangle \in M$ contains a subexpression of the form $\mathsf{max}(S)$, $e' \in S$ and $S' = S - \{e'\}$.*

In the above definition, each application of τ_{max} takes care of taking out one element e' inside a max subexpression by creating two non-deterministic cost functions, one with the cost of such element e' and another one with the remaining ones. This process is iteratively repeated until the fixed point is reached and there are no more max subexpressions to be transformed. The result of this operation is a max-free cost function, denoted by $\mathit{fp}_{\mathsf{max}}(M)$. An important observation is that the constraints φ are not modified in this transformation.

Once we have removed all max-subexpressions, the following step consists in removing the nat-subexpressions to make two cases explicit. One case in which the subexpression is positive, hence the nat can be safely removed, and another one in which it is negative or zero, hence the subexpression becomes zero. As notation, we use capital letters to denote fresh variables which replace the nat subexpressions.

Definition 3 (nat-free operator). *Let $\langle e, \varphi\rangle$ be a* max*-free cost function. We define the* nat*-free operator $\tau_{\mathsf{nat}} : 2^{\mathcal{CF}} \mapsto 2^{\mathcal{CF}}$ as follows: $\tau_{\mathsf{nat}}(M) = (M - \{\langle e, \varphi\rangle\}) \cup \{\langle e_i, \varphi_i\rangle \mid \varphi \wedge \varphi_i$ is satisfiable $, 1 \leq i \leq 2\}$, where $\langle e, \varphi\rangle \in M$ contains a subexpression $\mathsf{nat}(l)$, $\varphi_1 = \varphi \cup \{A = l, A > 0\}$, $\varphi_2 = \varphi \cup \{l \leq 0\}$, with A a fresh variable, and $e_1 = e[\mathsf{nat}(l) \mapsto A]$, $e_2 = e[\mathsf{nat}(l) \mapsto 0]$.*

In contrast to the max elimination transformation, the elimination of nat subexpressions modifies the set of linear constraints by adding the new assignments of fresh variables to linear expressions and the fact that the subexpression is greater than zero or when it becomes zero. The above operator τ_{nat} is applied iteratively until there are new terms to transform. The result of this operation is a nat-free cost function, denoted by $\mathit{fp}_{\mathsf{nat}}(M)$. For instance, for the cost function $\langle \mathsf{nat}(x) * \mathsf{nat}(z{-}1), \{x > 0\}\rangle$, $\mathit{fp}_{\mathsf{nat}}$ returns the set composed of the following nat-free cost functions:

$$\langle A*B, \{A=x, A>0, B=z{-}1, B>0\}\rangle \text{ and } \langle A*0, \{A=x, A>0, z{-}1\leq 0\}\rangle$$

In the following, given a cost function f, we denote by $\tau(f)$ the set $fp_{\mathsf{nat}}(fp_{\mathsf{max}}(\{f\}))$ and we say that each element in $fp_{\mathsf{nat}}(fp_{\mathsf{max}}(\{f\}))$ is a *flat* cost function.

Example 2. Let us consider the cost function in Ex. 1. Since such cost function contains the context constraint $n{>}0$, then the subexpressions $\mathsf{nat}(n)$ and $\mathsf{nat}(2{*}n{-}1)$ are always positive. By assuming that fp_{nat} replaces $\mathsf{nat}(n)$ by A and $\mathsf{nat}(2{*}n{-}1)$ by B, only those linear constraints containing $\varphi=\{n>0, A=n, A>0, B=2{*}n{-}1, B>0\}$ are satisfiable (the remaining cases are hence not considered). We obtain the following set of flat functions:

(1) $\langle 2^A{*}(31{+}8{*}\log(1{+}B){+}C{*}(10{+}6{*}D)){+}3{*}2^A, \varphi_1=\varphi\cup\{C{=}a, C>0, D{=}b, D{>}0\}\rangle$
(2) $\langle 2^A{*}(31{+}8{*}\log(1{+}B)){+}3{*}2^A, \varphi_2=\varphi\cup\{a{\leq}0, D{=}b, D{>}0\}\rangle$
(3) $\langle 2^A{*}(31{+}8{*}\log(1{+}B){+}C{*}10{+}3{*}2^A, \varphi_3=\varphi\cup\{C{=}a, C>0, b{\leq}0\}\rangle$
(4) $\langle 2^A{*}(31{+}8{*}\log(1{+}B)){+}3{*}2^A, \varphi_4=\varphi\cup\{a{\leq}0, b{\leq}0\}\rangle$ □

In order to compare cost functions, we start by comparing two flat cost functions in Def. 4 below. Then, in Def. 5 we compare a flat function against a general, i.e., non-flat, one. Finally, Def. 6 allows comparing two general functions.

Definition 4 (smaller flat cost function in context). *Given two flat cost functions $\langle e_1,\varphi_1\rangle$ and $\langle e_2,\varphi_2\rangle$, we say that $\langle e_1,\varphi_1\rangle$ is smaller than or equal to $\langle e_2,\varphi_2\rangle$ in the context of φ_2, written $\langle e_1,\varphi_1\rangle\trianglelefteq\langle e_2,\varphi_2\rangle$, if for all assignments σ such that $\sigma\models\varphi_1\cup\varphi_2$ it holds that $\sigma(e_1)\leq\sigma(e_2)$.*

Observe that the assignments in the above definition must satisfy the conjunction of the constraints in φ_1 and in φ_2. Hence, it discards the values for which the constraints become incompatible. An important point is that Def. 4 allows comparing pairs of flat functions. However, the result of such comparison is weak in the sense that the comparison is only valid in the context of φ_2. In order to determine that a flat function is smaller than a general function for any context we need to introduce Def. 5 below.

Definition 5 (smaller flat cost function). *Given a flat cost function $\langle e_1,\varphi_1\rangle$ and a (possibly non-flat) cost function $\langle e_2,\varphi_2\rangle$, we say that $\langle e_1,\varphi_1\rangle$ is smaller than or equal to $\langle e_2,\varphi_2\rangle$, written $\langle e_1,\varphi_1\rangle\preceq\langle e_2,\varphi_2\rangle$, if $\varphi_1\models\varphi_2$ and for all $\langle e_i,\varphi_i\rangle\in\tau(\langle e_2,\varphi_2\rangle)$ it holds that $\langle e_1,\varphi_1\rangle\trianglelefteq\langle e_i,\varphi_i\rangle$.*

Note that Def. 5 above is only valid when the context constraint φ_2 is more general, i.e., less restrictive than φ_1. This is required because in order to prove that a function is smaller than another one it must be so for all assignments which are satisfiable according to φ_1. If the context constraint φ_2 is more restrictive than φ_1 then there are valid input values for $\langle e_1,\varphi_1\rangle$ which are undefined for $\langle e_2,\varphi_2\rangle$. For example, if we want to check whether the flat cost function (1) in Ex. 2 is smaller than another one f which has the context constraint $\{n>4\}$, the comparison will fail. This is because function f is undefined for the input values $0<n\leq 4$. This condition is also required in Def. 6 below, which can be used on two general cost functions.

Definition 6 (smaller cost function). *Consider two cost functions $\langle e_1, \varphi_1\rangle$ and $\langle e_2, \varphi_2\rangle$ such that $\varphi_1 \models \varphi_2$. We say that $\langle e_1, \varphi_1\rangle$ is smaller than or equal to $\langle e_2, \varphi_2\rangle$ iff for all $\langle e'_1, \varphi'_1\rangle \in \tau(\langle e_1, \varphi_1\rangle)$ it holds that $\langle e'_1, \varphi'_1\rangle \preceq \langle e_2, \varphi_2\rangle$.*

In several applications of resource usage analysis, we are not only interested in knowing that a function is smaller than or equal than another. Also, if the comparison fails, it is useful to know which are the pairs of flat functions for which we have not been able to prove them being smaller, together with their context constraints. This can be useful in order to strengthen the context constraint of the left hand side function or to weaken that of the right hand side function.

4 Inclusion of Cost Functions

It is clearly not possible to try all assignments of input variables in order to prove that the comparison holds as required by Def. 4 (and transitively by Defs. 5 and 6). In this section, we aim at defining a practical technique to syntactically check that one flat function is smaller or equal than another one for all valid assignments, i.e., the relation $\trianglelefteq$ of Def. 4. The whole approach is defined over flat cost functions since from it one can use Defs. 5 and 6 to apply our techniques on two general functions.

The idea is to first *normalize* cost functions so that they become easier to compare by removing parenthesis, grouping identical terms together, etc. Then, we define a series of *inclusion schemas* which provide sufficient conditions to syntactically detect that a given expression is smaller or equal than another one. An important feature of our approach is that when expressions are syntactically compared we compute an approximated difference (denoted adiff) of the comparison, which is the subexpression that has not been required in order to prove the comparison and, thus, can still be used for subsequent comparisons. The whole comparison is presented as a fixed point transformation in which we remove from cost functions those subexpressions for which the comparison has already been proven until the left hand side expression becomes zero, in which case we succeed to prove that it is smaller or equal than the other, or no more transformations can be applied, in which case we fail to prove that it is smaller. Our approach is safe in the sense that whenever we determine that a function is smaller than another one this is actually the case. However, since the approach is obviously approximate, as the problem is undecidable, there are cases where one function is actually smaller than another one, but we fail to prove so.

4.1 Normalization Step

In the sequel, we use the term *basic cost expression* to refer to expressions of the form n, $\log_a(A{+}1)$, A^n, a^l. Furthermore, we use the letter b, possibly subscripted, to refer to such cost expressions.

Definition 7 (normalized cost expression). *A* normalized *cost expression is of the form $\Sigma_{i=1}^{n} e_i$ such that each e_i is a product of basic cost expressions.*

Note that each cost expression as defined above can be normalized by repeatedly applying the distributive property of multiplication over addition in order to get rid of all parentheses in the expression. We also assume that products which are composed of the same basic expressions (modulo constants) are grouped together in a single expression which adds all constants.

Example 3. Let us consider the cost functions in Ex. 2. Normalization results in the following cost functions:

$(1)_n$ $\langle 34{*}2^A{+}8{*}\log_2(1{+}B){*}2^A{+}10{*}C{*}2^A{+}6{*}C{*}D{*}2^A,$
$\varphi_1 = \{A{=}n, A{>}0, B{=}2{*}n{-}1, B{>}0, C{=}a, C > 0, D{=}b, D{>}0\}\rangle$
$(2)_n$ $\langle 34{*}2^A{+}8{*}\log_2(1{+}B){*}2^A,$
$\varphi_2 = \{A{=}n, A{>}0, B{=}2{*}n{-}1, B{>}0, a{\leq}0, D{=}b, D{>}0\}\rangle$
$(3)_n$ $\langle 34{*}2^A{+}8{*}\log_2(1{+}B){*}2^A{+}10{*}C{*}2^A,$
$\varphi_3 = \{A{=}n, A{>}0, B{=}2{*}n{-}1, B{>}0, C{=}a, C > 0, b{\leq}0\}\rangle$
$(4)_n$ $\langle 34{*}2^A{+}8{*}\log_2(1{+}B){*}2^A,$
$\varphi_4 = \{A{=}n, A{>}0, B{=}2{*}n{-}1, B{>}0, a{\leq}0, b{\leq}0\}\rangle$

□

Since $e_1 * e_2$ and $e_2 * e_1$ are equal, it is convenient to view a *product* as the set of its elements (i.e., basic cost expressions). We use $\mathcal{P}_b$ to denote the set of all products (i.e., sets of basic cost expressions) and $\mathcal{M}$ to refer to one product of $\mathcal{P}_b$. Also, since $\mathcal{M}_1 + \mathcal{M}_2$ and $\mathcal{M}_2 + \mathcal{M}_1$ are equal, it is convenient to view the *sum of products* as the set of its elements (its products). We use $\mathcal{P}_\mathcal{M}$ to denote the set of all sums of products and $\mathcal{S}$ to refer to one sum of products of $\mathcal{P}_\mathcal{M}$. Therefore, *a normalized cost expression* is a set of sets of basic cost expressions.

Example 4. For the normalized cost expressions in Ex. 3, we obtain the following set representation:

$(1)_s$ $\langle\{\{34, 2^A\}, \{8, \log_2(1{+}B), 2^A\}, \{10, C, 2^A\}, \{6, C, D, 2^A\}\},$
$\varphi_1 = \{A{=}n, A{>}0, B{=}2{*}n{-}1, B{>}0, C{=}a, C > 0, D{=}b, D{>}0\}\rangle$
$(2)_s$ $\langle\{\{34, 2^A\}, \{8, \log_2(1{+}B), 2^A\}\},$
$\varphi_2 = \{A{=}n, A{>}0, B{=}2{*}n{-}1, B{>}0, a{\leq}0, D{=}b, D{>}0\}\rangle$
$(3)_s$ $\langle\{\{34, 2^A\}, \{8, \log_2(1{+}B), 2^A\}, \{10, C, 2^A\}\},$
$\varphi_3 = \{A{=}n, A{>}0, B{=}2{*}n{-}1, B{>}0, C{=}a, C > 0, b{\leq}0\}\rangle$
$(4)_s$ $\langle\{\{34, 2^A\}, \{8, \log_2(1{+}B), 2^A\}\},$
$\varphi_4 = \{A{=}n, A{>}0, B{=}2{*}n{-}1, B{>}0, a{\leq}0, b{\leq}0\}\rangle$

□

4.2 Product Comparison

We start by providing sufficient conditions which allow proving the $\trianglelefteq$ relation on the basic cost expressions that will be used later to compare products of basic cost expressions. Given two basic cost expressions e_1 and e_2, the third column in Table 1 specifies sufficient, linear conditions under which e_1 is smaller or equal than e_2 in the context of φ (denoted as $e_1 \leq_\varphi e_2$). Since the conditions under which $\leq_\varphi$ holds are over linear expressions, we can rely on existing linear constraint solving techniques to automatically prove them. Let us explain some of entries in the table. E.g., verifying that $A^n \leq m^l$ is equivalent to verifying

e_1	e_2	$e_1 \leq_{\varphi} e_2$	adiff
n	n'	$n \leq n'$	1
n	$\log_a(A+1)$	$\varphi \models \{a^n \leq A+1\}$	1
n	A^m	$m > 1 \wedge \varphi \models \{n \leq A\}$	A^{m-1}
n	m^l	$m > 1 \wedge \varphi \models \{n \leq l\}$	m^{l-n}
l_1	l_2	$l_2 \notin \mathbb{N}^+$, $\varphi \models \{l_1 {\leq} l_2\}$	1
l	A^n	$n > 1 \wedge \varphi \models \{l \leq A\}$	A^{n-1}
l	$n^{l'}$	$n > 1 \wedge \varphi \models \{l \leq l'\}$	$n^{l'-l}$
$\log_a(A{+}1)$	l	$l \notin \mathbb{N}^+$, $\varphi \models \{A+1 \leq l\}$	1
$\log_a(A{+}1)$	$\log_b(B{+}1)$	$a \geq b \wedge \varphi \models \{A \leq B\}$	1
$\log_a(A{+}1)$	B^n	$n > 1 \wedge \varphi \models \{A+1 \leq B\}$	B^{n-1}
$\log_a(A{+}1)$	n^l	$n > 1 \wedge \varphi \models \{l > 0, A+1 \leq l\}$	$n^{l-(A+1)}$
A^n	B^m	$n > 1 \wedge m > 1 \wedge n \leq m \wedge \varphi \models \{A \leq B\}$	B^{m-n}
A^n	m^l	$m > 1 \wedge \varphi \models \{n * A \leq l\}$	m^{l-n*A}
n^l	$m^{l'}$	$n \leq m \wedge \varphi \models \{l \leq l'\}$	$m^{l'-l}$

Table 1. Comparison of basic expressions $e_1 \leq_{\varphi} e_2$

$\log_m(A^n) \leq \log_m(m^l)$, which in turn is equivalent to verifying that $n{*}\log_m(A) \leq l$ when $m > 1$ (i.e., $m \geq 2$ since m is an integer value). Therefore we can verify a stronger condition $n * A \leq l$ which implies $n * \log_m(A) \leq l$, since $\log_m(A) \leq A$ when $m \geq 2$. As another example, in order to verify that $l \leq n^{l'}$, it is enough to verify that $\log_n(l) \leq l'$ when $n > 1$, which can be guaranteed if $l \leq l'$.

The "part" of e_2 which is not required in order to prove the above relation becomes the *approximated difference* of the comparison operation, denoted $\mathsf{adiff}(e_1, e_2)$. An essential idea in our approach is that adiff is a cost expression in our language and hence we can transitively apply our techniques to it. This requires having an approximated difference instead of the exact one. For instance, when we compare $A \leq 2^B$ in the context $\{A \leq B\}$, the approximated difference is 2^{B-A} instead of the exact one $2^B - A$. The advantage is that we do not introduce the subtraction of expressions, since that would prevent us from transitively applying the same techniques.

When we compare two products $\mathcal{M}_1$, $\mathcal{M}_2$ of basic cost expressions in a context constraint φ, the basic idea is to prove the inclusion relation $\leq_{\varphi}$ for every basic cost expression in $\mathcal{M}_1$ w.r.t. a different element in $\mathcal{M}_2$ and at each step accumulate the difference in $\mathcal{M}_2$ and use it for future comparisons if needed.

Definition 8 (product comparison operator). *Given* $\langle \mathcal{M}_1, \varphi_1 \rangle, \langle \mathcal{M}_2, \varphi_2 \rangle$ *in* $\mathcal{P}_b$ *we define the product comparison operator* $\tau_* : (\mathcal{P}_b, \mathcal{P}_b) \mapsto (\mathcal{P}_b, \mathcal{P}_b)$ *as follows:* $\tau_*(\mathcal{M}_1, \mathcal{M}_2) = (\mathcal{M}_1 - \{e_1\}, \mathcal{M}_2 - \{e_2\} \cup \{\mathsf{adiff}(e_1, e_2)\})$ *where* $e_1 \in \mathcal{M}_1$, $e_2 \in \mathcal{M}_2$, *and* $e_1 \leq_{\varphi_1 \wedge \varphi_2} e_2$.

In order to compare two products, first we apply the above operator τ_* iteratively until there are no more terms to transform. In each iteration we pick e_1 and e_2 and modify $\mathcal{M}_1$ and $\mathcal{M}_2$ accordingly, and then repeat the process on the new sets. The result of this operation is denoted $fp_*(\mathcal{M}_1, \mathcal{M}_2)$. This process is finite because the size of $\mathcal{M}_1$ strictly decreases at each iteration.

Example 5. Let us consider the product $\{8, \log_2(1{+}B), 2^A\}$ which is part of $(1)_s$ in Ex. 4. We want to prove that this product is smaller or equal than the following one $\{7, 2^{3*B}\}$ in the context $\varphi = \{A \leq B{-}1, B{\geq}10\}$. This can be done by applying the τ_* operator three times. In the first iteration, since we know by Table 1 that $\log_2(1{+}B) \leq_\varphi 2^{3*B}$ and the adiff is 2^{2*B-1}, we obtain the new sets $\{8, 2^A\}$ and $\{7, 2^{2*B-1}\}$. In the second iteration, we can prove that $2^A \leq_\varphi 2^{2*B-1}$, and add as adiff $2^{2*B-A-1}$. Finally, it remains to be checked that $8 \leq_\varphi 2^{2*B-A-1}$. This problem is reduced to checking that $\varphi \models 8 \leq 2{*}B{-}A{-}1$, which it trivially true. □

The following lemma states that if we succeed to transform $\mathcal{M}_1$ into the empty set, then the comparison holds. This is what we have done in the above example.

Lemma 1. *Given $\langle \mathcal{M}_1, \varphi_1 \rangle, \langle \mathcal{M}_2, \varphi_2 \rangle$ where $\mathcal{M}_1, \mathcal{M}_2 \in \mathcal{P}_b$ and for all $e \in \mathcal{M}_1$ it holds that $\varphi_1 \models e \geq 1$. If $fp_*(\mathcal{M}_1, \mathcal{M}_2) = (\emptyset, \_)$ then $\langle \mathcal{M}_1, \varphi_1 \rangle \unlhd \langle \mathcal{M}_2, \varphi_2 \rangle$.*

Note that the above operator is non-deterministic due to the (non-deterministic) choice of e_1 and e_2 in Def. 8. Thus, the computation of $fp_*(\mathcal{M}_1, \mathcal{M}_2)$ might not lead directly to $(\emptyset, \_)$. In such case, we can backtrack in order to explore other choices and, in the limit, all of them can be explored until we find one for which the comparison succeeds.

4.3 Comparison of Sums of Products

We now aim at comparing two sums of products by relying on the product comparison of Sec. 4.2. As for the case of basic cost expressions, we are interested in having a notion of approximated adiff when comparing products. The idea is that when we want to prove $k_1{*}A \leq k_2{*}B$ and $A \leq B$ and k_1 and k_2 are constant factors, we can leave as approximated difference of the product comparison the product $(k_2 - k_1){*}B$, provided $k_2 - k_1$ is greater or equal than zero. As notation, given a product $\mathcal{M}$, we use $\mathsf{constant}(\mathcal{M})$ to denote the constant factor in $\mathcal{M}$, which is equals to n if there is a constant $n \in \mathcal{M}$ with $n \in \mathbb{N}^+$ and, otherwise, it is 1. We use $\mathsf{adiff}(\mathcal{M}_1, \mathcal{M}_2)$ to denote $\mathsf{constant}(\mathcal{M}_2) - \mathsf{constant}(\mathcal{M}_1)$.

Definition 9 (sum comparison operator). *Given $\langle \mathcal{S}_1, \varphi_1 \rangle$ and $\langle \mathcal{S}_2, \varphi_2 \rangle$, where $\mathcal{S}_1, \mathcal{S}_2 \in \mathcal{P}_\mathcal{M}$, we define the* sum comparison operator $\tau_+ : (\mathcal{P}_\mathcal{M}, \mathcal{P}_\mathcal{M}) \mapsto (\mathcal{P}_\mathcal{M}, \mathcal{P}_\mathcal{M})$ *as follows: $\tau_+(\mathcal{S}_1, \mathcal{S}_2) = (\mathcal{S}_1 - \{\mathcal{M}_1\}, (\mathcal{S}_2 - \{\mathcal{M}_2\}) \cup \mathcal{A})$ iff $fp_*(\mathcal{M}_1, \mathcal{M}_2) = (\emptyset, \_)$ where:*

- *$\mathcal{A} = \{\ \}$ if $\mathsf{adiff}(\mathcal{M}_1, \mathcal{M}_2) \leq 0$;*
- *otherwise, $\mathcal{A} = (\mathcal{M}_2 - \{\mathsf{constant}(\mathcal{M}_2)\}) \cup \{\mathsf{adiff}(\mathcal{M}_1, \mathcal{M}_2)\}$.*

In order to compare sums of products, we apply the above operator τ_+ iteratively until there are no more elements to transform. As for the case of products, this process is finite because the size of $\mathcal{S}_1$ strictly decreases in each iteration. The result of this operation is denoted by $fp_+(\mathcal{S}_1, \mathcal{S}_2)$.

Example 6. Let us consider the sum of products $(3)_s$ in Ex. 4 together with $\mathcal{S} = \{\{50, C, 2^B\}, \{9, D^2, 2^B\}\}$ and the context constraint $\varphi = \{1{+}B{\le}D\}$. We can prove that $(3)_s \trianglelefteq \mathcal{S}$ by applying τ_+ three times as follows:

1. $\tau_+((3)_s, \mathcal{S}) = ((3)_s - \{\{34, 2^A\}\}, \mathcal{S}')$, where $\mathcal{S}' = \{\{16, C, 2^B\}, \{9, D^2, 2^B\}\}$. This application of the operator is feasible since $\mathit{fp}_*(\{34, 2^A\}, \{50, C, 2^B\}) = (\emptyset, \_)$ in the context $\varphi_3 \wedge \varphi$, and the difference constant part of such comparison is 16.
2. Now, we perform one more iteration of τ_+ and obtain as result $\tau_+((3)_s - \{\{34, 2^A\}\}, \mathcal{S}') = ((3)_s - \{\{34, 2^A\}, \{10, C, 2^A\}\}, \mathcal{S}'')$, where $\mathcal{S}'' = \{\{6, C, 2^B\}, \{9, D^2, 2^B\}\}$. Observe that in this case $\mathit{fp}_*(\{10, C, 2^A\}, \{\{16, C, 2^B\}) = (\emptyset, \_)$.
3. Finally, one more iteration of τ_+ on the above sum of products, gives $(\emptyset, \mathcal{S}''')$ as result, where $\mathcal{S}''' = \{\{6, C, 2^B\}, \{1, D^2, 2^B\}\}$.

In this last iteration we have used the fact that $\{1{+}B{\le}D\} \in \varphi$ in order to prove that $\mathit{fp}_*(\{8, \log_2(1{+}B), 2^A\}, \{9, D^2, 2^B\}) = (\emptyset, \_)$ within the context $\varphi_3 \wedge \varphi$. □

Theorem 1. *Let $\langle \mathcal{S}_1, \varphi_1 \rangle$, $\langle \mathcal{S}_2, \varphi_2 \rangle$ be two sum of products such that for all $\mathcal{M} \in \mathcal{S}_1$, $e \in \mathcal{M}$ it holds that $\varphi_1 \models e \geq 1$. If $\mathit{fp}_+(\mathcal{S}_1, \mathcal{S}_2) = (\emptyset, \_)$ then $\langle \mathcal{S}_1, \varphi_1 \rangle \trianglelefteq \langle \mathcal{S}_2, \varphi_2 \rangle$.*

Example 7. For the sum of products in Ex. 6, we get $\mathit{fp}_+((3)_s, \mathcal{S}) = (\emptyset, \mathcal{S}''')$. Thus, according to the above theorem, it holds that $\langle (3)_s, \varphi_3 \rangle \trianglelefteq \langle \mathcal{S}, \varphi \rangle$. □

4.4 Composite Comparison of Sums of Products

Clearly the previous schema for comparing sums of products is not complete. There are cases like the comparison of $\{\{A^3\}, \{A^2\}, \{A\}\}$ w.r.t. $\{\{A^6\}\}$ within the context constraint $A > 1$ which cannot be proven by using a one-to-one comparison of products. This is because a single product comparison would consume the whole expression A^6. We try to cover more cases by providing a *composite* comparison schema which establishes when a single product is greater than the addition of several products.

Definition 10 (sum-product comparison operator). *Consider $\langle \mathcal{S}_1, \varphi_1 \rangle$ and $\langle \mathcal{M}_2, \varphi_2 \rangle$, where $\mathcal{S}_1 \in \mathcal{P}_{\mathcal{M}}$, $\mathcal{M}_2 \in \mathcal{P}_b$ and for all $\mathcal{M} \in \mathcal{S}_1$ it holds that $\varphi_1 \models \mathcal{M} > 1$. Then, we define the* sum-product comparison operator *$\tau_{(+,*)} : (\mathcal{P}_{\mathcal{M}}, \mathcal{P}_b) \mapsto (\mathcal{P}_{\mathcal{M}}, \mathcal{P}_b)$ as follows: $\tau_{(+,*)}(\mathcal{S}_1, \mathcal{M}_2) = (\mathcal{S}_1 - \{\mathcal{M}_2'\}, \mathcal{M}_2'')$, where $\mathit{fp}_*(\mathcal{M}_2', \mathcal{M}_2) = (\emptyset, \mathcal{M}_2'')$.*

The above operator $\tau_{(+,*)}$ is applied while there are new terms to transform. Note that the process is finite since the size of S_1 is always decreasing. We denote by $\mathit{fp}_{(+,*)}(\mathcal{S}_1, \mathcal{M}_2)$ the result of iteratively applying $\tau_{(+,*)}$.

Example 8. By using the sum-product operator we can transform the pair $(\{\{A^3\}, \{A^2\}, \{A\}\}, \{A^6\})$ into $(\emptyset, \emptyset)$ in the context constraint $\varphi = \{A > 1\}$. To this end, we apply $\tau_{(+,*)}$ three times. In the first iteration, $\mathit{fp}_*(\{A^3\}, \{A^6\}) = (\emptyset, \{A^3\})$. In the second iteration, $\mathit{fp}_*(\{A^2\}, \{A^3\}) = (\emptyset, \{A\})$. Finally in the third iteration $\mathit{fp}_*(\{A\}, \{A\}) = (\emptyset, \emptyset)$. □

When using the sum-product comparison operator to compare sums of products, we can take advantage of having an approximated difference similar to the one defined in Sec. 4.3. In particular, we define the approximated difference of comparing $\mathcal{S}$ and $\mathcal{M}$, written $\mathsf{adiff}(\mathcal{S},\mathcal{M})$, as $\mathsf{constant}(\mathcal{M}){-}\mathsf{constant}(\mathcal{S})$, where $\mathsf{constant}(\mathcal{S}){=}\sum_{\mathcal{M}'\in\mathcal{S}}\mathsf{constant}(\mathcal{M}')$. Thus, if we compare $\{\{A^3\},\{A^2\},\{A\}\}$ is smaller or equal than $\{4, A^6\}$, we can have as approximated difference $\{A^6\}$, which is useful to continue comparing further summands. As notation, we use $\mathcal{P}_{\mathcal{S}}$ to denote the set of all sums of products and $\mathcal{S}_s$ to refer one element.

Definition 11 (general sum comparison operator). *Let us consider* $\langle\mathcal{S}_s,\varphi\rangle$ *and* $\langle\mathcal{S}_2,\varphi'\rangle$*, where* $\mathcal{S}_s\in\mathcal{P}_{\mathcal{S}}$ *and* $\mathcal{S}_2\in\mathcal{P}_{\mathcal{M}}$*. We define the general sum comparison operator* $\mu_+ : (\mathcal{P}_{\mathcal{S}},\mathcal{P}_{\mathcal{M}}) \mapsto (\mathcal{P}_{\mathcal{S}},\mathcal{P}_{\mathcal{M}})$ *as follows:* $\mu_+(\mathcal{S}_s,\mathcal{S}_2) = (\mathcal{S}_s - \{\mathcal{S}_1\}, (\mathcal{S}_2{-}\{\mathcal{M}\}){\cup}\mathcal{A})$*, where* $\mathit{fp}_{(+,*)}(\mathcal{S}_1,\mathcal{M}) = (\emptyset, \_)$ *and* $\mathcal{A} = \{\ \}$ *if* $\mathsf{adiff}(\mathcal{S}_1,\mathcal{M}) \leq 0$*; otherwise* $\mathcal{A} = (\mathcal{M} - \{\mathsf{constant}(\mathcal{M})\}) \cup \{\mathsf{adiff}(\mathcal{S}_1,\mathcal{M})\}$*.*

Similarly as we have done in definitions above, the above operator μ_+ is applied iteratively while there are new terms to transform. Since the cardinality of $\mathcal{S}_s$ decreases in each step the process is finite. We denote by $\mathit{fp}^g_+(\mathcal{S}_s,\mathcal{S}_2)$ to the result of applying the above iterator until there are no sets to transform.

Observe that the above operator does not replace the previous sum comparator operator in Def. 9 since it sometimes can be of less applicability since $\mathit{fp}_{(+,*)}$ requires that all elements in the addition are strictly greater than one. Instead, it is used in combination with Def. 9 so that when we fail to prove the comparison by using the one-to-one comparison we attempt with the sum-product comparison operator above.

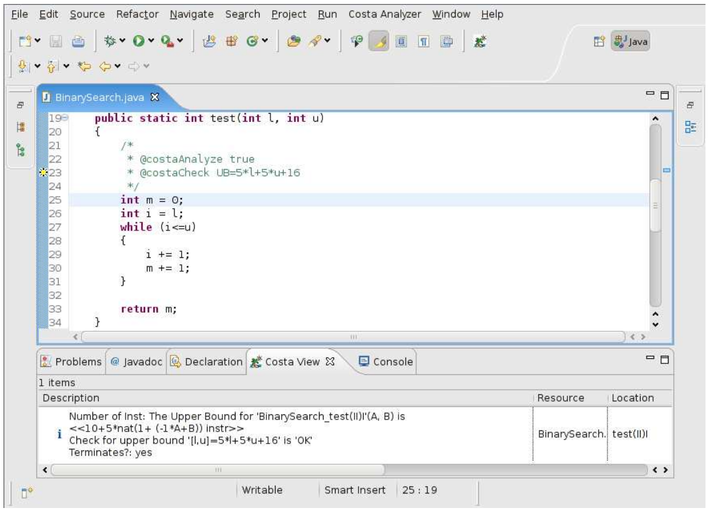
In order to apply the general sum comparison operator, we seek for partitions in the original $\mathcal{S}$ which meet the conditions in the definition above.

Theorem 2 (composite inclusion). *Let* $\langle\mathcal{S}_1,\varphi_1\rangle$*,* $\langle\mathcal{S}_2,\varphi_2\rangle$ *be two sum of products such that for all* $\mathcal{M}'\in\mathcal{S}_1$*,* $e\in\mathcal{M}'$ *it holds* $\varphi_1\models e{>}1$*. Let* $\mathcal{S}_s$ *be a partition of* $\mathcal{S}_1$*. If* $\mathit{fp}^g_+(\mathcal{S}_s,\mathcal{S}_2) = (\emptyset, \_)$ *then* $\langle\mathcal{S}_1,\varphi_1\rangle{\trianglelefteq}\langle\mathcal{S}_2,\varphi_2\rangle$*.*

5 Implementation and Experimental Evaluation

We have implemented our technique and it can be used as a back-end of existing non-asymptotic cost analyzers for average, lower and upper bounds (e.g., [8,2,10,4,5]), and regardless of whether it is based on the approach to cost analysis of [12] or any other. Currently, it is integrated within the COSTA System, and it can be tried out through its web interface which is available from `http://costa.ls.fi.upm.es`.

We first illustrate the application of our method in resource usage verification by showing the working mode of COSTA through its Eclipse plugin. Figure 2 shows a method which has been annotated to be analyzed (indicated by the annotation `@costaAnalyze true`) and its resulting upper bound compared against the cost function written in the assertion `@costaCheck`. The output of COSTA is shown in the *Costa view* (bottom side of the Figure). There, the upper bound



```java
    public static int test(int l, int u)
    {
        /*
         * @costaAnalyze true
         * @costaCheck UB=5*l+5*u+16
         */
        int m = 0;
        int i = l;
        while (i<=u)
        {
            i += 1;
            m += 1;
        }

        return m;
    }
```

1 items

Description	Resource	Location
Number of Inst: The Upper Bound for 'BinarySearch_test(II)I'(A, B) is <<10+5*nat(1+ (-1*A+B)) instr>> Check for upper bound '[l,u]=5*l+5*u+16' is 'OK' Terminates?: yes	BinarySearch.	test(II)I

Fig. 2. Screenshot of the COSTA plugin for Eclipse, showing how annotations are used to interact with COSTA

inferred by COSTA is displayed, together with the result of the comparison with the user's assertion. Besides, the verification of the upper bound is shown in the same line where the annotation is as a marker in the left side of the editor. If the verification fails, a warning marker is shown, instead of the star-like marker of Figure 2. Thus, by annotating the methods of interest with candidate upper bounds, it is possible to verify the resource usage of such methods, and to mark those methods that do not meet their resource usage specification.

In Table 2, we have performed some experiments which aim at providing some information about the accuracy and the efficiency of our technique. The first seven benchmark programs correspond to examples taken from the JOlden benchmark suite [11], the next two ones from the experiments in [1] and the last one is our running example. COSTA infers the upper bound cost functions for them which are shown in the second column of the table. All execution times shown are in milliseconds and have been computed as the average time of ten executions. The environment were the experiments were run was Intel Core2 Duo 1.20 GHz with 2 processors, and 4 GB of RAM.

The first column is the name of the benchmark. The second column is the expresion of the cost function. The next two columns show the time taken by our implementation of the comparison approach presented in the paper in two different experiments which we describe below. The next two columns include the term size of the cost function inferred by COSTA and normalized as explained in Section 4, and the term size of the product of the cost function by itself. The next two columns include the ratio between size and time; those are estimations of the number of terms processed by milisecond in the comparison. We use CF to refer to the cost function computed by COSTA.

Bench.	**Cost Function**	T_1	T_2	**Size**$_1$	**Size**$_2$	$Size/T_1$	$Size/T_2$
`bH`	$128 + 96 * \mathsf{nat}(x)$	0	0.2	6	11	N/A	N/A
`treeAdd`	$4 + (4 * \mathsf{nat}(x) + 1) + 40 * 2^{\mathsf{nat}(y-1)}$	8	18	11	41	1.40	2.28
`biSort`	$16 + (4 * \mathsf{nat}(x) + 1) * \mathsf{nat}(y - 1)$	15	39	9	33	0.60	0.85
`health`	$28 * (4^{\mathsf{nat}(x-1)} - 1)/3 + 28 * 4^{\mathsf{nat}(x-1)}$	7	23	21	115	3.00	5.00
`voronoi`	$20 * \mathsf{nat}(2 * x - 1)$	2	5	3	5	1.50	1.00
`mst`	$\mathsf{max}(12 + 4 * \mathsf{nat}(1 + x)$ $+\mathsf{nat}(1 + x) * (20 + 4 * \mathsf{nat}(1/4 * x))$ $+16 * \mathsf{nat}(1 + x) * \mathsf{nat}(1 + x) + 8 * \mathsf{nat}(1 + x),$ $4 + \mathsf{max}(16 + 4 * \mathsf{nat}(1 + x)$ $+\mathsf{nat}(1 + x) * (20 + 4 * \mathsf{nat}(1/4 * x))$ $+16 * \mathsf{nat}(1 + x) * \mathsf{nat}(1 + x) + 16 * \mathsf{nat}(1 + x),$ $20 + 4 * \mathsf{nat}(1 + x)+$ $+\mathsf{nat}(1 + x) * (20 + 4 * \mathsf{nat}(1/4 * x))+$ $4 * \mathsf{nat}(1/4 * x)))$	96	222	49	241	0.51	1.09
`em3d`	$93 + 4 * \mathsf{nat}(t) + 4 * \mathsf{nat}(y)+$ $\mathsf{nat}(t - 1) * (28 + 4 * \mathsf{nat}(y)) + 4 * \mathsf{nat}(t)+$ $4 * \mathsf{nat}(y) + \mathsf{nat}(t - 1) * (28 + 4 * \mathsf{nat}(y))+$ $4 * \mathsf{nat}(y)$	54	113	19	117	0.35	1.04
`multiply`	$9 + \mathsf{nat}(x) * (16 + 8 * \log_2(1 + \mathsf{nat}(2 * x - 3)))$	10	24	14	55	1.40	2.29
`evenDigits`	$49 + (\mathsf{nat}(z) * (37 + (\mathsf{nat}(y) * (32 + 27 * \mathsf{nat}(y))$ $+27 * \mathsf{nat}(y))) + \mathsf{nat}(y) * (32 + 27 * \mathsf{nat}(y))$ $+27 * \mathsf{nat}(y))$	36	94	29	195	0.81	2.07
`running`	$2^{\mathsf{nat}(x)} * (31 + (8 * \log_2(1 + \mathsf{nat}(2 * x - 1))+$ $+\mathsf{nat}(y) * (10 + 6 * \mathsf{nat}(z)))) + 3 * 2^{\mathsf{nat}(x)}$	40	165	34	212	0.85	1.28

Table 2. Experiments in Cost Function Comparison

T_1 Time taken by the comparison $CF \preceq rev(CF)$, where $rev(CF)$ is just the reversed version of CF. I.e., $rev(x + y + 1) = 1 + x + y$. The size of the expressions involved in the comparison is shown in the fifth column of the table (Size$_1$).

T_2 Time taken by the comparison $CF + CF \preceq CF * CF$, assuming that CF takes at least the value 2 for all input values. In this case, the size of the expression grows considerably and hence the comparison takes a longer time than the previous case. The size of the largest expression in this case is shown in the sixth column of the table (Size$_2$).

In all cases, we have succeeded to prove that the comparison holds. Ignoring the first benchmark, that took a negligible time, the ratio between size and time and falls in a narrow interval (1 or 2 terms processed by milisecond). Interestingly, for each one of the benchmarks (except `voronoi`), that ratio increases with term size, implying that the number of terms processed by milisecond is higher in more complex expressions. However, these performance measurements should be verified with a larger number of case studies, to verify how it varies with the size of the input. We leave that task as further work. In any case, we believe that our preliminary experiments indicate that our approach is sufficiently precise in practice and that the comparison times are acceptable.

6 Other Approaches and Related Work

In this section, we discuss other possible approaches to handle the problem of comparing cost functions. In [7], an approach for inferring non-linear invariants using a linear constraints domain (such as polyhedra) has been introduced. The idea is based on a *saturation* operator, which lifts linear constraints to non-linear ones. For example, the constraint $\Sigma a_i x_i = a$ would impose the constraint $\Sigma a_i Z_{x_i u} = au$ for each variable u. Here $Z_{x_i u}$ is a new variable which corresponds to the multiplication of x_i by u. This technique can be used to compare cost functions, the idea is to start by saturating the constraints and, at the same time, converting the expressions to linear expressions until we can use a linear domain to perform the comparison. For example, when we introduce a variable $Z_{x_i u}$, all occurrences of $x_i u$ in the expressions are replaced by $Z_{x_i u}$. Let us see an example where: in the first step we have the two cost functions to compare; in the second step, we replace the exponential with a fresh variable and add the corresponding constraints; in the third step, we replace the product by another fresh variable and saturate the constraints:

$$\begin{array}{l|l}
w \cdot 2^x \geq 2^y & \{x \geq 0, x \geq y, w \geq 0\} \\
w \cdot Z_{2^x} \geq Z_{2^y} & \{x \geq 0, x \geq y, Z_{2^x} \geq Z_{2^y}\} \\
Z_{w \cdot 2^x} \geq Z_{2^y} & \{x \geq 0, x \geq y, Z_{2^x} \geq Z_{2^y}, Z_{w \cdot 2^x} \geq Z_{2^y}\}
\end{array}$$

Now, by using a linear constraint domain, the comparison can be proved. We believe that the saturation operation is very expensive compared to our technique while it does not seem to add significant precision.

Another approach for checking that $e_1 \preceq e_2$ in the context of a given context constraint φ is to encode the comparison $e_1 \preceq e_2$ as a Boolean formula that simulates the behavior of the underlying machine architecture. The unsatisfiability of the Boolean formula can be checked using SAT solvers and implies that $e_1 \preceq e_2$. The drawback of this approach is that it requires fixing a maximum number of bits for representing the value of each variable in e_i and the values of intermediate calculations. Therefore, the result is guaranteed to be sound only for the range of numbers that can be represented using such bits. On the positive side, the approach is complete for this range. In the case of variables that correspond to integer program variables, the maximum number of bits can be easily derived from the one of the underlying architecture. Thus, we expect the method to be precise. However, in the case of variables that correspond to the size of data-structures, the maximum number of bits is more difficult to estimate.

Another approach for this problem is based on numerical methods since our problem is analogous to proving whether $0 \preceq b - f_m$ in the context ϕ_b. There are at least two numerical approaches to this problem. The first one is to find the roots of $b - f_m$, and check whether those roots satisfy the constraints ϕ_b. If they do not, a single point check is enough to solve the problem. This is because, if the equation is verified at one point, the expressions are continuous, and there is no sign change since the roots are outside the region defined by ϕ_b, then we can ensure that the equation holds for all possible values satisfying ϕ_b. However, the problem of finding the roots with multiple variables is hard in general and often not solvable. The second approach is based on the observation that there is no need to compute the actual values of the roots. It is enough to know whether there are roots in the region defined by ϕ_b. This can be done by finding the minimum values of expression $b - f_m$, a problem that is more affordable using numerical methods [9] . If the minimum values in the region defined by ϕ_b are greater than zero, then there are no roots in that region. Even if those minimum values are out of the region defined by ϕ_b or smaller than zero, it is not necessary to continue trying to find their values. If the algorithm starts to converge to values out of the region of interest, the comparison can be proven to be false. One of the open issues about using numerical methods to solve our problem is whether or not they will be able to handle cost functions output from realistic programs and their performance. We have not explored these issues yet and they remain as subject of future work.

7 Conclusions

In conclusion, we have proposed a novel approach to comparing cost functions which is relatively efficient and powerful enough for performing useful comparisons of cost functions. Making such comparisons automatically and efficiently is essential for any application of automatic cost analysis. Our approach could be combined with more heavyweight techniques, such as those based on numerical methods, in those cases where our approach is not sufficiently precise.

Acknowledgments. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-231620 *HATS* project, by the MEC under the TIN-2008-05624 *DOVES* and HI2008-0153 (Acción Integrada) projects, by the UCM-BSCH-GR58/08-910502 (GPD-UCM) , and the CAM under the S-0505/TIC/0407 *PROMESAS* project.

References

1. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *15th International Symposium on Static Analysis (SAS'08)*, volume 5079 of *Lecture Notes in Computer Science*, pages 221–237, 2008.
2. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *ESOP*, LNCS 4421, pages 157–172. Springer, 2007.
3. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. 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)*, number 5382 in Lecture Notes in Computer Science, pages 113–133. Springer, 2007.
4. V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric Prediction of Heap Memory Requirements. In *ISMM*. ACM Press, 2008.
5. W-N. Chin, H.H. Nguyen, C. Popeea, and S. Qin. Analysing Memory Resource Bounds for Low-Level Programs. In *ISMM*. ACM Press, 2008.
6. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL'77*, pages 238–252. ACM, 1977.
7. B. S. Gulavani and S. Gulwani. A Numerical Abstract Domain Based on Expression Abstraction and Max Operator with Application in Timing Analysis. In *CAV*, LNCS 5123, pages 370–384. Springer, 2008.

8. S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139. ACM, 2009.
9. S. Kirkpatrick, Jr. C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.
10. J. Navas, M. Méndez-Lojo, and M. Hermenegildo. User-Definable Resource Usage Bounds Analysis for Java Bytecode. In *BYTECODE*. Elsevier, 2009.
11. JOlden Suite. `http://www-ali.cs.umass.edu/DaCapo/benchmarks.html`.
12. B. Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9), 1975.