A Practical Comparator of Cost Functions and its Applications ☆

Elvira Albert[a], Puri Arenas[a], Samir Genaim[a], Germán Puebla[b]

[a] DSIC, Complutense University of Madrid (UCM), Spain
[b] DLSIIS, Technical University of Madrid (UPM), Spain

Abstract

Automatic cost analysis has significantly advanced in the last few years. Nowadays, a number of cost analyzers exist which automatically produce upper- and/or lower-bounds on the amount of resources required to execute a program. Cost analysis has a number of important applications such as resource-usage verification and program synthesis and optimization. For such applications to be successful, it is not sufficient to have automatic cost analysis. It is also required to have automated means for handling the analysis results, which are in the form of *Cost Functions* (*CFs* for short) i.e., non-recursive expressions composed of a relatively small number of types of basic expressions. In particular, we need automated means for *comparing CFs* in order to prove that a *CF* is smaller than or equal to another one for all input values of interest. General function comparison is a hard mathematical problem. Rather than attacking the general problem, in this work we focus on comparing *CFs* by exploiting their syntactic properties and we present, to the best of our knowledge, the first practical *CF* comparator which opens the door to fully automated applications of cost analysis. We have implemented the comparator and made its source code available *online*, so that any cost analyzer can use it.

Keywords: Resource analysis, cost analysis, function comparison, upper/lower bounds

1. Introduction

Cost analysis [28, 12], a.k.a. resource analysis, aims at statically predicting the resource consumption of programs in terms of their input data sizes. Given a program, cost analysis produces a *Cost Function* (*CF* for short) which may correspond to an upper-bound or a lower-bound, depending on the kind of

☆This work was funded partially by the EU project FP7-ICT-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu) and by the Spanish projects TIN2008-05624 and TIN2012-38137.

Email addresses: `elvira@sip.ucm.es` (Elvira Albert), `puri@sip.ucm.es` (Puri Arenas), `samir.genaim@fdi.ucm.es` (Samir Genaim), `german@fi.upm.es` (Germán Puebla)

analysis performed. 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. Starting from the seminal cost analysis framework by Wegbreit [28], cost analyzers are often generic on the notion of *cost model*, e.g., they can be used to measure different resources, such as the number of instructions executed, the amount of memory allocated, the number of calls to a certain method, etc. Thus, *CFs* can be used to represent the usage of 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 *CFs*. This allows choosing an implementation with smaller cost or guaranteeing that the given resource-usage bounds are preserved. Essentially, given a program m, a set of linear constraints φ which impose size restrictions on the input values to m (e.g., that an argument is larger than a certain value or that the size of an array is non-zero), and a *CF* f_m^φ, we aim at comparing it with another *CF* bound b. Depending on the application, such functions can be automatically inferred by a resource analyzer (e.g., if we want to compare the efficiency of two implementations) or 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 positive function, e.g., $\mathsf{b} - f_m^\varphi \geq 0$ in the context φ. This is in general undecidable[1] and also non-trivial, as *CFs* involve non-linear subexpressions (e.g., exponential, polynomial and logarithmic subexpressions).

1.1. Summary of Contributions

As our first main contribution, we present 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 of the following steps, which constitute our main technical contributions:

1. Normalizing cost functions to a form which makes them amenable to be syntactically compared. This step includes handling operators like max and min (used to express the maximum and minimum of a set of expressions), and transforming arithmetic expressions into 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.

[1]Since variables range over the integers, undecidability follows from the undecidability of that of Hilbert's 10th problem: given a multivariate polynomial $p(\bar{x})$, decide whether $p(\bar{x}) = 0$ has an integer solution. This problem can be reduced to checking whether $p(\bar{x})^2$ is positive, which is an instance of comparing cost functions.

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.

The second main contribution is an implementation of the cost comparator that we have made available online at `costa.ls.fi.upm.es/comparator` to the resource analysis community and which is free software under the *General Public License* (GPL). We define there the syntax of the cost functions used in the implementation, and provide specifications of its interface functions, so that our comparator can be easily integrated in any resource analyzer.

A preliminary version of this work was presented at FOPARA'09 [3]. This article improves [3] in several aspects: (1) it notably improves the formalization of the comparison process, (2) it formally proves the correctness of the approach, (3) it extends the method to also handle lower bounds, (4) we present applications (including new ones) of the comparator, and (5) finally we provide a new implementation of our approach.

1.2. Organization of the Article

The rest of the paper is organized as follows. The next section introduces some background in cost analysis and cost functions and presents the syntax of the cost expressions (*CEs* for short) which we handle in this paper. Section 3 presents the problem of comparing two *CFs* in a context provided by means of constraints. In order to come up with practical ways to solve the problem, we propose means for handling the nat-, max- and min-operators and transform the comparison problem to that of comparing a series of expressions which no longer contain such operators. In Section 4, we introduce a novel approach to proving that a *CF* is smaller than another one. Our approach is based on a series of syntactic schemes which provide sufficient conditions to syntactically detect that a given expression is smaller than or equal to another one. The comparison is presented as a fixed point transformation in which we remove from *CEs* those subexpressions for which the comparison has already been proven until the left hand side expression becomes zero. Section 5 discusses several applications of our *CFs* comparator, namely its direct use to check the efficiency improvement of program optimizations, and for program verification and certification. Interestingly, in cases in which an upper bound cannot be found (for instance because the analyzer does not find an upper bound on the number of loop iterations), our comparator can be used to check that the resource consumption is below a given threshold. An overview of other approaches and related work is presented in Section 6 and some conclusions are presented in Section 7.

2. Background on Cost Analysis

This section introduces some background material on cost analysis and presents the syntax of the *CEs* studied in the paper. We start by introducing some notation. The sets of natural, integer and non-zero natural values

are denoted by $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{N}^+$, respectively. We write x, y, and z, to denote variables which range over $\mathbb{Z}$. The notation $\bar{t}$ stands for a sequence $t_1, \ldots, t_n$, for some $n>0$. A *linear expression* over a sequence of variables $\bar{x}$ is of the form $v_0 + v_1x_1 + \ldots + v_nx_n$, where $v_i \in \mathbb{Z}$, $0 \le i \le n$ and $x_i \in \bar{x}$ for all $1 \le i \le n$. Similarly, a *linear constraint* (over $\mathbb{Z}$) has the form $l_1 \le l_2$, where l_1 and l_2 are linear expressions. For simplicity we write $l_1 = l_2$ instead of $l_1 \le l_2 \wedge l_2 \le l_1$, and $l_1 < l_2$ instead of $l_1 + 1 \le l_2$. 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 $x > 2y$. Given any entity t, $vars(t)$ stands for the set of variables in t. We write φ 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 $\sigma(\varphi)$ takes the value *true*, and we say that φ is satisfiable if there exists an assignment σ such that $\sigma \models \varphi$.

2.1. Cost Functions

As already mentioned, cost analysis produces its results in terms of *cost functions*, which are functions of the type $\mathbb{Z}^n \mapsto \mathbb{N}$, where each input value to the cost function corresponds to an input value to the original program. However, whereas programs can have input arguments of many different types, cost functions are restricted to integer inputs. For this to be possible, input arguments which are not of type integer have to be abstracted to their *size*. Different size measures may be used for the same input type. Therefore, the analysis results, in order to be understandable, have to be accompanied by the size measure which has been applied to each argument.

It is customary to analyze programs (or methods) w.r.t. some initial *context constraint*. Essentially, given a method $m(\bar{x})$, the considered context constraint φ describes conditions on the sizes of the initial values of $\bar{x}$. With such information, a cost analyzer outputs a *cost function* $f_m^\varphi(\bar{x}_s) = e$, where e is a cost expression and $\bar{x}_s$ denotes the data sizes of $\bar{x}$. Thus, f_m^φ is a function of the input data sizes that provides information 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 $\bar{x}_s$. For simplicity, in the rest of the paper we simply write $\bar{x}$ instead of $\bar{x}_s$ to refer to the input values to a cost function. 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 [6], 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 [5] for details.*

Given the context constraint $\{n > 0\}$*, the analyzer outputs the* upper bound *cost function for the method* m *which is shown at the bottom of the figure. Since* m *contains two recursive calls, the complexity is exponential in* n*, namely we*

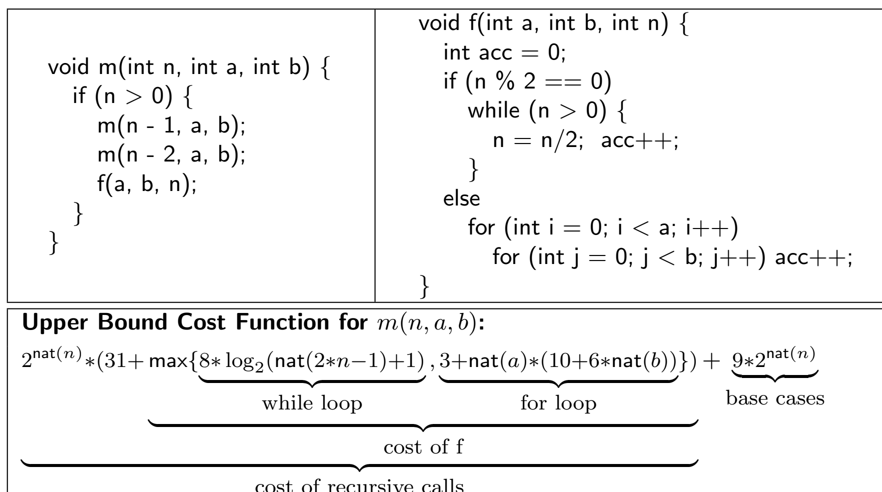

Figure 1: Running example and upper bound on the number of executed bytecode instructions.

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 how a conditional statement introduces a* max*-operator in the upper bound. The* then *branch executes the* while *loop, which has a logarithmic complexity because the loop counter is divided by 2 at each iteration. The* else *branch executes the* for *loop, which has a quadratic complexity since it contains a nested* for *loop. 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 9 instructions.*

Note that all expressions involving 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 due to the context constraint we know that $n > 0$. However, the* nat *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. We prefer to keep the* max*-operator separate from the* nat*-operator since* nat *is just a special case of* max*, and thus treating them separately will simplify their handling later.*

2.2. Cost Expressions

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

Definition 1 (cost expression). *Given a sequence of variables $\bar{x}$, a* cost expression *(CE) for $\bar{x}$ is a symbolic expression generated using the 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 $n, a \in \mathbb{N}^+$, $a \geq 2$, l is a linear expression over $\bar{x}$, S is a non-empty set of CEs, and $\mathsf{nat} : \mathbb{Z} \to \mathbb{N}$ is defined as $\mathsf{nat}(v) = \mathsf{max}(\{v, 0\})$. Given an assignment σ over $\bar{x}$ and a cost expression e, $\sigma(e)$ is the natural number which results from the evaluation of e w.r.t. σ. In the case of $\log_a(e)$, it should be interpreted as $\lfloor \log_a(e) \rfloor$, since CEs must be evaluable to natural numbers.

As already mentioned in Example 1 above, linear expressions are wrapped by nat, which stands for *natural value*. Logarithmic expressions contain a linear subexpression plus "1", which ensures that they cannot be evaluated to $\log_a(0)$. Also, max-subexpressions are used to represent the cost of disjunctive branches in the program, as seen in the running example.

From now on, we assume that the right hand side of a cost function for a method $m(\bar{x})$ is a cost expression over $\bar{x}$. This guarantees that *CEs* become fully evaluable to a natural number for any assignment over $\bar{x}$. If we ignore syntactic differences, one can say that cost analyzers produce cost functions which adhere to the syntax of *CEs* presented above.

3. Comparing Cost Functions containing nat- and max-Operators

We now study the problem of comparing two *CFs* over the same sequence of variables $\bar{x}$ which adhere to the syntax provided in Definition 1 above in the context of a set of constraints φ over $\bar{x}$. While the *CEs* syntax provides a compact format and it is good for human comprehension, the use of the nat- and max-operators poses important problems for comparing *CFs*. We now propose a syntactic transformation which converts cost functions represented using the syntax provided in Definition 1 into another syntax in which no max- nor nat-expressions are needed, which simplifies the task of automatically comparing pairs of *CFs*.

Definition 2 corresponds to the intuition of the condition under which a *CF* is smaller than another one within a certain context.

Definition 2 (smaller cost function in context). *Let f_1 and f_2 be two CFs over the sequence of variables $\bar{x}$. Let φ be a context constraint. We say that f_1 is smaller than f_2 in the context φ, denoted $f_1 \leq_\varphi f_2$ iff for all assignments σ over $\bar{x}$ s.t. $\sigma \models \varphi$ we have that $\sigma(f_1) \leq \sigma(f_2)$.*

The problem with this definition is that it cannot be applied directly in general because it would require evaluating the cost functions for all possible input values, which is an (infinitely) large set. As a result, we will try to provide other ways of statically proving that a *CF* is smaller than another one.

The transformation we propose is performed in two steps. In the first one we study how to handle nat-operators. In the second one we propose a mechanism for eliminating max-operators.

3.1. *Eliminating* nat*-Operators*

An important observation is that nat-expressions are in general not *statically evaluable*, i.e., we cannot evaluate them without knowing the values of the input variables to the cost function. However, an expression of the form $\mathsf{nat}(l)$ becomes statically evaluable if the context constraint φ considered guarantees that either $l \leq 0$, in which case the whole expression takes the value zero, or that $l > 0$, in which case the nat-wrapper can be removed. Therefore, if the context constraint allows it, cost expressions can be statically evaluated into nat-*free cost expressions*, whose syntax is presented below.

Definition 3 (nat-free cost expression). *Given a sequence of variables $\bar{x}$ and a context φ, a* nat-free cost expression *for $\bar{x}$ and φ is a symbolic expression which adheres to the grammar:*

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

where n and a are defined as in Definition 1 and A is a bounded *integer variable, i.e., the context φ must (1) contain a constraint of the form $A = l$ with l a linear expression over $\bar{x}$ and (2) $\varphi \models A > 0$.*

nat-free cost expressions are simpler than regular cost expressions in that they no longer contain nat-operators, however they may contain *bound* variables. Note that φ has to satisfy two properties: (1) guarantees that the value of A is fixed for any assignment over $\bar{x}$ and (2) guarantees that nat-free cost expressions do not take negative values.

The main idea in order to convert a regular cost expression containing a subexpression of the form $\mathsf{nat}(l)$ into a pair of nat-free cost expressions is that if the context constraint φ does not entail $l \leq 0$ nor $l > 0$, we can always split the context φ into two cases, $\varphi_1 = \varphi \cup \{l \leq 0\}$, where $l \leq 0$ trivially holds, and $\varphi_2 = \varphi \cup \{l > 0\}$, where $l > 0$ trivially holds. If we succeed to prove any property in both φ_1 and φ_2 separately, then the property holds for the initial φ context constraint. In order to be able to have a representation of *CFs* without nat-subexpressions, we will use functions defined by cases.

Definition 4 (nat-free cost function). *A* nat-free cost function *f over variables $\bar{x}$ is of the form $\{\langle e_1, \varphi_1\rangle, \ldots, \langle e_n, \varphi_n\rangle\}$, with $n \geq 1$ and where each e_i is a* nat*-free cost expression for $\bar{x}$ and φ_i. It should be interpreted as:*

$$f(\bar{x}) = \begin{cases} e_1 & \varphi_1 \\ \vdots & \vdots \\ e_n & \varphi_n \end{cases}$$

The φ_i constraints indicate the conditions under which the *CF* takes the value e_i. All *CFs* defined by cases have to satisfy that the constraints in different cases are *mutually exclusive* and that the union of all constraints cover all possible input values, i.e., for any input value there is exactly one case whose constraint is satisfied.

The process of converting a cost function represented by a *CE* e into a nat-free cost function can be conceptually split in two steps. In the first step we collect all linear expressions which appear inside nat-operators and we obtain a partition of the initial context constraint into a set of contexts which guarantee that all nat-expressions in e are statically evaluable. Then, in the second step we obtain the simplified version of e which can be considered in each case by statically evaluating all nat-operators as discussed above. We now present a function which computes a set of contexts where all considered nat-expressions are evaluable. We use $\mathcal{CE}$ to denote the set of all possible cost expressions, $\mathcal{L}$ to denote the set of all linear expressions and Φ to denote the set of all possible contexts. Also, given a cost expression e, we denote by $nats(e)$ the set $\{l \mid \mathsf{nat}(l) \text{ occurs in } e\}$.

Definition 5 (nat-evaluable contexts). *Let e be a CE and φ a satisfiable context constraint. We define the* $\mathsf{natContexts}$ *function* $\mathsf{natContexts}$*:* $\mathcal{CE} \times \Phi \mapsto 2^{\Phi}$ *as* $\mathsf{natContexts}(e, \varphi) = \tau_{\mathsf{nat}}(nats(e), \{\varphi\})$ *where* $\tau_{\mathsf{nat}} : 2^{\mathcal{L}} \times 2^{\Phi} \mapsto 2^{\Phi}$ *is defined as:*

$$\tau_{\mathsf{nat}}(L, C) = \begin{cases} C & L = \emptyset \\ \tau_{\mathsf{nat}}(L - \{l\}, \{\varphi' \in split(C, l) \mid \varphi' \textit{ is satisfiable}\}) & l \in L \end{cases}$$

where L *is a set of linear expressions,* C *is a set of context constraints, and* $split(C, l) = \{\varphi'' \ \cup \ \{l \leq 0\} \mid \varphi'' \in C\} \ \cup \ \{\varphi'' \ \cup \ \{l > 0\} \mid \varphi'' \in C\}$.

Note that the τ_{nat} function is recursive and it calls itself as many times as required until we obtain sufficiently many contexts for all nat-expressions to be statically evaluable. In each iteration, the *split* function potentially duplicates the number of contexts. However, some of the generated contexts by *split* are inconsistent (they contain incompatible constraints) and they are pruned away when calling τ_{nat} recursively.

The second argument to the $\mathsf{natContexts}$ function should correspond to the context φ in which the comparison of cost expressions will take place. The advantage of doing so is that we can prune away as soon as possible those cases which are incompatible with φ.

Example 2. *Let us consider the CE* $e = \mathsf{nat}(x){*}\mathsf{nat}(z{-}1)$ *whose behaviour we want to study under the context constraint* $\{x > 0\}$*. First we apply* τ_{nat} *on the set of* nat*-subexpressions* $nats(e) = \{x, z{-}1\}$ *in* e*. In the first iteration of* $\tau_{\mathsf{nat}}(\{x, z{-}1\}, \{\{x > 0\}\})$*, we need to compute* $split(\{\{x > 0\}\}, x)$*, which returns* $\{\{x > 0, x \leq 0\}, \{x > 0, x > 0\}\}$ *as result. Since* $\{x > 0, x \leq 0\}$ *is not satisfiable, the recursive call to* τ_{nat} *takes the form* $\tau_{\mathsf{nat}}(\{z{-}1\}, \{\{x > 0\}\})$*. Now, since* $split(\{\{x > 0\}\}, z{-}1) = \{\varphi_1, \varphi_2\}$*, where* $\varphi_1 = \{x > 0, z{-}1 \leq 0\}$ *and* $\varphi_2 = \{x > 0, z{-}1 > 0\}$ *are both satisfiable, the new iteration of* $\tau_{\mathsf{nat}}(\{\}, \{\varphi_1, \varphi_2\})$ *returns* $\{\varphi_1, \varphi_2\}$*, i.e.,* $\mathsf{natContexts}(e, \{x > 0\})$ *returns* $\{\varphi_1, \varphi_2\}$*.*

Let us consider now both cost expressions:

$$\begin{aligned} e_1 &= 2^{\mathsf{nat}(n)}{*}(31{+}8{*}\log_2(\mathsf{nat}(2{*}n{-}1){+}1)){+}9{*}2^{\mathsf{nat}(n)} \\ e_2 &= 2^{\mathsf{nat}(n)}{*}(3{+}\mathsf{nat}(a){*}(10{+}6{*}\mathsf{nat}(b))){+}9{*}2^{\mathsf{nat}(n)} \end{aligned}$$

coming from the upper bound cost function $m(n,a,b)$ *in Fig. 1 together with the context constraint* $\{n>0\}$. *Let us focus on* e_1, *where* $nats(e_1)=\{n, 2*n-1\}$. *It holds that* $\mathsf{natContexts}(e_1,\{n>0\})=\tau_{\mathsf{nat}}(\{n,2*n-1\},\{\{n>0\}\}) = \{\varphi_{11}\}$, *where* $\varphi_{11}=\{n>0, 2*n-1>0\}$. *Similarly, let us consider* e_2. *This time,* $nats(e_2)=\{n,a,b\}$, *and* $\mathsf{natContexts}(e_2,\{n>0\})=\tau_{\mathsf{nat}}(\{n,a,b\},\{\{n>0\}\})=\{\varphi_{21},\varphi_{22},\varphi_{23},\varphi_{24}\}$, *where:*

$$\begin{array}{lcllcl}\varphi_{21} & = & \{n>0, a\leq 0, b\leq 0\} & \varphi_{22} & = & \{n>0, a\leq 0, b>0\}\\ \varphi_{23} & = & \{n>0, a>0, b\leq 0\} & \varphi_{24} & = & \{n>0, a>0, b>0\}\end{array}$$

We now define a function which statically evaluates cost expressions in contexts where all nat-expressions are evaluable. As notation, we use capital letters to denote fresh integer variables which replace the nat-subexpressions. We write $e[a \mapsto b]$ to denote the expression obtained from e by replacing all occurrences of subexpression a in e with b.

Definition 6 (nat-evaluation). *Let* e *be a cost expression. Let* φ *be a satisfiable context constraint. The* nat*-evaluation of* e *w.r.t.* φ, *denoted* $e\downarrow_{\varphi}$ *is defined as*

$$e\downarrow_{\varphi}=\begin{cases}\langle e'\downarrow_{\varphi},\varphi\rangle, & \mathsf{nat}(l)\in e\wedge\varphi\models l\leq 0\\ \quad \textit{where } e'=e[\mathsf{nat}(l)\mapsto 0] & \\ \langle e'\downarrow_{\varphi},\varphi'\rangle, & \mathsf{nat}(l)\in e\wedge\varphi\models l>0\\ \quad \textit{where } e'=e[\mathsf{nat}(l)\mapsto A]\wedge & \\ \quad \varphi'=\varphi\cup\{A=l\} & \\ \langle e,\varphi\rangle & \textit{otherwise}\end{cases}$$

where A *stands for a fresh integer variable.*

Note that nat-evaluation not only removes nat-wrappers. Those linear expressions which are guaranteed to take positive values are replaced with a distinct fresh variable. The relation between the original linear expression l and the new variable is kept by adding a new equality constraint to the context constraint which binds the new variable A to the corresponding linear expression l. This is why we refer to A as a *bounded* variable.

Note that the use of bounded variables to replace nat-expressions is required to preserve the syntax of cost expressions in Definition 1, and more concretely the syntax of nat-free cost expressions in Definition 3. For instance, if we consider the cost expression $e = 1+\mathsf{nat}(x-y)$, and we transform it without using bounded variables, we would obtain $\langle 1+(x-y),\{x-y>0\}\rangle$ and $\langle 1,\{x-y\leq 0\rangle\}$, where $1+(x-y)$ is not a cost expression. However, by using bounded variables we generate $\langle 1+A,\{A=x-y, x-y>0\}\rangle$ and $\langle 1,\{x-y\leq 0\}\rangle$, where $1+A$ is in fact a nat-free cost expression. It is important to note that these new equality constraints involving bounded variables are added only to preserve the syntactic structure of cost expressions, but they do not affect the set of input values for which the context constraint is satisfied.

Example 3. *Let us consider the CEs e, e_1, e_2 and the context constraints φ_i, φ_{2j}, $1 \le i \le 2$, $1 \le j \le 4$ in Example 2. Then, the result of evaluating the CE e in the contexts φ_1, φ_2 is $e\downarrow_{\varphi_1} = \langle x{*}0, \varphi_1\rangle$, $e\downarrow_{\varphi_2} = \langle x{*}A, \varphi_2 \cup \{A{=}z{-}1\}\rangle$ respectively. Similarly, it holds:*

$$
\begin{array}{lrcl}
(1) & e_1\downarrow_{\varphi_{11}} & = & \langle 2^A{*}(31{+}8{*}\log_2(B{+}1)){+}9{*}2^A, \varphi_{11} \cup \{A{=}n, B{=}2{*}n{-}1\}\rangle \\
(2) & e_2\downarrow_{\varphi_{21}} & = & \langle 2^A{*}3{+}9{*}2^A, \varphi_{21} \cup \{A{=}n\}\rangle \\
(3) & e_2\downarrow_{\varphi_{22}} & = & \langle 2^A{*}3{+}9{*}2^A, \varphi_{22} \cup \{A{=}n, C{=}b\}\rangle \\
(4) & e_2\downarrow_{\varphi_{23}} & = & \langle 2^A{*}(3{+}B{*}10){+}9{*}2^A, \varphi_{23} \cup \{A{=}n, B{=}a\}\rangle \\
(5) & e_2\downarrow_{\varphi_{24}} & = & \langle 2^A{*}(3{+}B{*}(10{+}6{*}C)){+}9{*}2^A, \varphi_{24} \cup \{A{=}n, B{=}a, C{=}b\}\rangle
\end{array}
$$

We now provide a definition which shows the process of comparing two cost functions defined by cases. This is done by comparing them in a set of contexts where the nat-operators are statically evaluable. Also, in the following definition, $(e_1, e_2)\downarrow_\varphi$ indicates that we perform nat-evaluation of e_1 and e_2 simultaneously. This guarantees that if the same linear expression occurs in both e_1 and e_2 within nat-operators, the same bound integer variable is used in both *CEs*.

Definition 7 (smaller cost expression in nat-evaluable contexts). *Let e_1 and e_2 be two CEs. Let φ be a satisfiable context constraint. We say that e_1 is* smaller in nat-evaluable contexts *than e_2 in φ, denoted $e_1 \preceq_\varphi e_2$ iff $\forall \varphi' \in \mathsf{natContexts}(e_1 + e_2, \varphi)$ it holds that $e_1' \le_{\varphi''} e_2'$, where $(e_1, e_2)\downarrow_{\varphi'} = \langle (e_1', e_2'), \varphi''\rangle$.*

Definition 7 above allows statically reducing the general comparison problem into a set of smaller subproblems which cover the initial one. In the call to natContexts we use the addition operator '+' to build a single cost expression from e_1 and e_2. This way, identical linear expressions are replaced with the same bounded variable in both e_1 and e_2.

Theorem 1 (Correctness of comparison of *CEs* defined by cases). *Let e_1 and e_2 be two CEs. Let φ be a satisfiable context constraint. Then $e_1 \preceq_\varphi e_2$ iff $e_1 \le_\varphi e_2$.*

Proof. We prove both implications:

($\Rightarrow$). Suppose that $e_1 \preceq_\varphi e_2$. We want to prove that $e_1 \le_\varphi e_2$, i.e., for all assignments σ such that $\sigma \models \varphi$, it holds that $\sigma(e_1) \le \sigma(e_2)$. Since $\tau_{\mathsf{nat}}(nats(e_1 + e_2), \{\varphi\})$ covers all possible input values and all constraints in it are mutually exclusive, for any assignment $\sigma \models \varphi$ there exists a context $\varphi_1 \in \tau_{\mathsf{nat}}(nats(e_1 + e_2), \{\varphi\})$ such that $\sigma \models \varphi_1$, where by construction, $\varphi_1 = \varphi \cup \varphi_a$, for some φ_a. Then $(e_1, e_2)\downarrow_{\varphi_1} = \langle (e_1', e_2'), \varphi_1'\rangle$, and $\varphi_1' = \varphi \cup \varphi_a \cup \varphi_b$ for some φ_b, where φ_b contains elements of the form $A = l$, where A is a fresh variable and $\mathsf{nat}(l)$ occurs either in e_1 or e_2. Let us define σ' as σ plus the assignments $\sigma'(A) = l$, for all $A = l$ in φ_b. It holds $\sigma' \models \varphi_1'$ and thus $\sigma'(e_1') \le \sigma'(e_2')$. Then, by construction together with the definition of σ', it holds $\sigma'(e_1') = \sigma(e_1)$ and $\sigma'(e_2') = \sigma(e_2)$. Hence $e_1 \le_\varphi e_2$.

($\Leftarrow$). We want to prove that $e_1 \preceq_\varphi e_2$. Let us consider $\varphi_1 \in \tau_{\mathsf{nat}}(nats(e_1 + e_2), \{\varphi\})$. Then by construction $\varphi_1 = \varphi \cup \varphi_a$, for some φ_a. Now we compute $(e_1, e_2) \downarrow_{\varphi_1} = \langle (e_1', e_2'), \varphi_1' \rangle$, where $\varphi_1' = \varphi \cup \varphi_a \cup \varphi_b$ for some φ_b , where φ_b contains constraints of the form $A = l$, where A is a fresh variable and $\mathsf{nat}(l)$ occurs either in e_1 or e_2. We need to prove then that $e_1' \leq_{\varphi_1'} e_2'$. Let us choose any assignment σ such that $\sigma \models \varphi_1'$. Then $\sigma \models \varphi$. Thus since $e_1 \leq_\varphi e_2$ it holds that $\sigma(e_1) \leq \sigma(e_2)$. Then, because of the structure of φ_b together with $\sigma \models \varphi_b$, it holds that $\sigma(e_1) = \sigma(e_1')$ and $\sigma(e_2) = \sigma(e_2')$. Hence $\sigma(e_1') \leq \sigma(e_2')$. □

This theorem guarantees that it is correct to split the initial problem into smaller problems where no nat-operators appear anymore. The following section tells us a possible mechanism for comparing functions which potentially contain max-operators.

3.2. *Eliminating* max*-Operators*

Intuitively, a max-operator is a shortcut for allowing the use of multiple *CFs* instead of a single one. When evaluating a *CF* with max-operators, the *CF* with the larger value is taken. Note that max-operators are required when an analyzer is not able to determine which of several alternative cost expressions is larger. As an example, in method m in the running example, a max-operator is required because the cost of the then branch and that of the else branch depend on different input values, n for the then branch and a and b for the else branch. Therefore, neither of the branches is larger than the other for all input values.

Since all subexpressions in cost expressions are positive, it is always possible to convert an expression with a max-operator in an inner level into an equivalent one which has the max-operator at the outermost level. Therefore, from now on, cost expressions are represented as sets of cost expressions which contain no max-operators, since they all have been moved to the outermost level. The following definition transforms an initially singleton set of *CFs* into a set of max-free *CFs* which cover all possible costs comprised in the original function. We use $\mathcal{CF}$ to denote the set of all possible cost functions, and we say that a set M of *CFs* is max-free if no cost function in M contains a max-operator.

Definition 8 (max-free function). *Let e be a CF. We define the* maxFree *function* $\mathsf{maxFree} : CF \mapsto 2^{\mathcal{CF}}$ *as* $\mathsf{maxFree}(e) = \tau_{\mathsf{max}}(\{e\})$*, where* $\tau_{\mathsf{max}} : 2^{\mathcal{CF}} \mapsto 2^{\mathcal{CF}}$ *is defined as:*

$$\tau_{\mathsf{max}}(M) = \begin{cases} M & M \textit{ is } \mathsf{max}\textit{-free} \\ \tau_{\mathsf{max}}((M - \{e\}) \cup M') & e \in M,\ e \textit{ contains } \mathsf{max}(S) \\ & M' = \{e' \mid e' {=} e[\mathsf{max}(S) \mapsto e''], e'' {\in} S\} \end{cases}$$

In the above definition, each application of τ_{max} takes care of taking out one expression e from M containing a $\mathsf{max}(S)$ subexpression by replacing e with as many cost expressions as elements in the set S. This process is iteratively repeated until there are no more max-subexpressions to be transformed. The result of this operation is a max-free *CF*. A similar treatment of max-expressions, for case analysis, appeared in [16].

Example 4. *Let us consider the cost expression* $e{=}1{+}\mathsf{max}(\{\mathsf{nat}(x),\mathsf{max}(\{\mathsf{nat}(y),\mathsf{nat}(z{+}1)\})\})$*. To compute* $\mathsf{maxFree}(e)$ *we start by applying* τ_{max} *to the singleton set* $\{e\}$*. Suppose that we remove the outermost* max *first. Then we obtain the set* $\{1{+}\mathsf{nat}(x), 1{+}\mathsf{max}(\{\mathsf{nat}(y),\mathsf{nat}(z{+}1)\})\}$*. Now we apply* τ_{max} *again on* $\{1{+}\mathsf{nat}(x), 1{+}\mathsf{max}(\{\mathsf{nat}(y),\mathsf{nat}(z{+}1)\})\}$ *to obtain the set* $\{1{+}\mathsf{nat}(x), 1{+}\mathsf{nat}(y), 1{+}\mathsf{nat}(z{+}1)\}$*. Since this set no longer contains any* max*-subexpression, then* $\mathsf{maxFree}(e) = \{1{+}\mathsf{nat}(x), 1{+}\mathsf{nat}(y), 1{+}\mathsf{nat}(z{+}1)\}$.

Similarly, for the cost expression $m(n,a,b)$ *in Fig. 1,* $\mathsf{maxFree}(m(n,a,b))$ *returns the set* $\{e_1, e_2\}$ *by applying one iteration of* $\tau_{\mathsf{max}}(\{m(n,a,b)\})$*, where* e_1 *and* e_2 *are the cost expressions in Example 2.*

Definition 9 (Smaller *CE* containing max-operators). *Let* e_1 *and* e_2 *be two CEs and* φ *be a satisfiable context constraint. We say that* e_1 *is smaller than* e_2 *in max-free format, denoted as* $e_1 \unlhd_\varphi e_2$*, iff for all* $e \in \mathsf{maxFree}(e_1)$*, exists* $e' \in \mathsf{maxFree}(e_2)$ *s.t.* $e \preceq_\varphi e'$*.*

Note that the order in which we remove max- and nat-operators is irrelevant. In fact, the definition above can be applied to cost expressions with nat-operators, since the $e \preceq_\varphi e'$ relation can handle nat-operators if needed. Finally, the theorem below guarantees that the proposed mechanism for handling max-operators is correct.

Theorem 2 (Correctness of comparison of *CEs* with max-operators). *Let* e_1 *and* e_2 *be two CEs. Let* φ *be a satisfiable context. Then* $e_1 \unlhd_\varphi e_2$ *iff* $e_1 \leq_\varphi e_2$*.*

Proof. We prove both implications.

($\Rightarrow$). Let σ be an assignment such that $\sigma \models \varphi$. Because of the definition of $\mathsf{maxFree}(e_1)$ which generates all possible expressions subsumed in e_1, it holds that $\sigma(e_1) = \sigma(e')$, for some $e' \in \mathsf{maxFree}(e_1)$. Since $e_1 \unlhd_\varphi e_2$, then there exists $e'' \in \mathsf{maxFree}(e_2)$ such that $e' \preceq_\varphi e''$. Let us consider the set $Max = \{e \in \mathsf{maxFree}(e_2) \mid e' \preceq_\varphi e\}$ which is different from the empty set since e'' belongs to it. We distinguish two cases.

(a) There exists $e \in Max$ such that $\sigma(e) = e_2$. Then $\sigma(e_1) = \sigma(e') \leq \sigma(e) = \sigma(e_2)$. Hence $e_1 \leq_\varphi e_2$ and the result holds.

(b) Otherwise, because of the definition of max-operators, there exists $e \in \mathsf{maxFree}(e_2)$ such that $\sigma(e) = \sigma(e_2)$. Note that the evaluation of max-operators ensures that $\sigma(e_2) \geq \sigma(e''')$, for all $e''' \in \mathsf{maxFree}(e_2)$. Then $\sigma(e) \in Max$ and the result can be proven as in item (a).

($\Leftarrow$). Suppose that $e_1 \leq_\varphi e_2$, i.e., for all $\sigma \models \varphi$ it holds $\sigma(e_1) \leq \sigma(e_2)$. Let us consider any $e \in \mathsf{maxFree}(e_1)$. By definition of $\mathsf{maxFree}(e_1)$ it holds that $\sigma(e) \leq_\varphi \sigma(e_1) \leq_\varphi \sigma(e_2)$. Again, the construction of $\mathsf{maxFree}(e_2)$ ensures that there exists $e' \in \mathsf{maxFree}(e_2)$ such that $\sigma(e_2) = \sigma(e')$. Hence $e \preceq_\varphi e'$ and thus $e_1 \unlhd_\varphi e_2$. $\square$

Note that we have provided mechanisms for getting rid of both nat- and max-operators from cost expressions. The cost expressions we obtain after removing such operators are referred to as *flat cost expressions* and their syntax is defined below.

Definition 10 (flat cost expression). *Given a sequence of variables $\bar{x}$ and a context φ, a* flat cost expression *for $\bar{x}$ and φ is a symbolic expression which adheres to this grammar:*

$$e ::= n \mid A \mid e + e \mid e * e \mid \log_a(A + 1) \mid A^n \mid a^A$$

where $n, a \in \mathbb{N}^+$, $a \geq 2$ and A is an integer variable bound in φ.

3.3. Eliminating min*-Operators*

As already discussed, for some applications of cost analysis it is interesting to compute lower-bounds (which correspond to the best-case cost) rather than upper-bounds (which correspond to worst-case cost). For example, lower bounds are used for scheduling the distribution of tasks in parallel execution in such a way that it is not worth parallelizing a task unless its (lower-bound) resource consumption is sufficiently large (see, e.g., [9, 15]).

When *CFs* correspond to lower-bounds they use the min-operator instead of max, i.e., cost expressions are like those defined in Definition 1 but replacing $\mathsf{max}(S)$ by $\mathsf{min}(S)$. Essentially, the min-operator is used when there are multiple choices and we want to denote the minimum cost for them. For instance, the lower bound cost for an *if-then-else* instruction is expressed as the minimum of the costs of the *then* and *else* branches. Our comparator can also handle the comparison of lower bound functions. As in the case of max-operators, min-operators can also be promoted to the outermost level since we are dealing with positive subexpressions grouped by addition and multiplication only. Then, in order to compare two lower bounds e_1 and e_2, it is enough to replace Definition 9 by the following one.

Definition 11 (Smaller *CE* containing min-operators). *Let e_1 and e_2 be two CEs. Let φ be a satisfiable context constraint. We say that e_1 is smaller than e_2 in min-free format, denoted as $e_1 \ll_\varphi e_2$, iff there exists $e \in \mathsf{minFree}(e_1)$ such that for all $e' \in \mathsf{minFree}(e_2)$ we have $e \preceq_\varphi e'$.*

Analogously to $\mathsf{maxFree}$, we use $\mathsf{minFree}$ to denote the result of the iterative application of the operator τ_{min}, which is the counterpart of τ_{max} considering min-expressions instead of max-expressions.

Note that the quantifiers in Definition 11 and Definition 9 are reversed. I.e., in Definition 9 we required that for all elements in the set of max-free expressions for e_1 there exists one element in the max-free expressions for e_2 which is larger. In the case of min-operators, we need instead that there exists an element in the set of min-free expressions for e_1 such that all elements in the min-free expressions for e_2 are larger than it.

The theorem below guarantees that the proposed mechanism for handling min-operators is correct. The proof is analogous to that of Theorem 2.

Theorem 3 (Correctness of comparison of *CEs* with min-operators). *Let e_1 and e_2 be two CEs. Let φ be a satisfiable context constraint. Then $e_1 \ll_\varphi e_2$ iff $e_1 \leq_\varphi e_2$.*

4. Comparing Flat Cost Functions

The definitions in the previous section allow handling max- and nat-operators and reduce the problem of comparing *CEs* containing them to that of comparing a series of flat *CEs* which follow the syntax in Definition 10. In this section, we aim at defining a practical technique to syntactically check that a flat function is smaller or equal than another one in a given context constraint φ, i.e., the initial relation $\leq_\varphi$ of Definition 2 but this time restricted to flat cost expressions only.

The way in which we will compare flat *CEs* consist of two steps. In the first one we *normalize* flat *CEs* by removing additions which appear as arguments to multiplications, grouping identical terms together, etc. in order to make them easier to compare. Then, we define a series of *inclusion schemes* 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 quotient 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 expressions 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 one, or no more transformations can be applied, in which case we fail to prove that it is smaller.

Our approach is sound in the sense that whenever we determine that a *CE* is smaller than another one this is actually the case. However, since the problem is undecidable and the approach is obviously approximate, our approach is incomplete, as there are cases where a *CE* is actually smaller than another one, but we fail to prove so.

4.1. Normalization of Cost Expressions

In the sequel, given a sequence of variables $\bar{x}$ and a context φ, a *basic cost expression* b for $\bar{x}$ and φ has the form n, A, $\log_a(A+1)$, A^n, a^l, where $n, a \in \mathbb{N}^+$, $a \geq 2$, A is an integer variable bound in φ and l is a linear expression over the variables in b.

Basic cost expressions are generated from flat cost expressions, which in fact come from cost expressions by eliminating nat$-$ and max-operators. Thus, because of Definition 6, a bound variable A occurring in a flat cost expression clearly satisfies that $\varphi \models A > 0$. Furthermore, note that we write a^l instead of a^A as done in Definition 10. This is because when grouping together subexpressions, we may introduce additions in the exponent. For instance $2^A * 2^B$ stands

for the basic cost expression 2^{A+B}. Note also that since l is a linear expression on integer bound variables and it holds that $\varphi \models A > 0$, for any bound variable A, then $\varphi \models l > 0$.

Definition 12 (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.*

Any flat cost expression can be normalized by repeatedly applying the distributive property of multiplication over addition in order to promote to outer levels any additions which appear as arguments to multiplications. 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 5. *Let us consider the cost functions* $(1), \ldots, (5)$ *in Example 3. Normalization results in the following cost functions:*

$(1)_n$ $\langle 40{*}2^A{+}8{*}\log_2(B{+}1){*}2^A, \varphi_{11} \cup \{A{=}n, B{=}2{*}n{-}1\}\rangle$
$(2)_n$ $\langle 12{*}2^A, \varphi_{21} \cup \{A{=}n\}\rangle$
$(3)_n$ $\langle 12{*}2^A, \varphi_{22} \cup \{A{=}n, C{=}b\}\rangle$
$(4)_n$ $\langle 12{*}2^A{+}10{*}2^A{*}B, \varphi_{23} \cup \{A{=}n, B{=}a\}\rangle$
$(5)_n$ $\langle 12{*}2^A{+}10{*}B{*}2^A{+}6{*}B{*}C{*}2^A, \varphi_{24} \cup \{A{=}n, B{=}a, C{=}b\}\rangle$

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 in $\mathcal{P}_b$. 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 in $\mathcal{P}_{\mathcal{M}}$. Therefore, *a normalized cost expression* can be represented as a set of sets of basic cost expressions.

Example 6. *For the normalized cost expressions in Example 5, we obtain the following set representation:*

$(1)_s$ $\langle \{\{40, 2^A\}, \{8, \log_2(B{+}1), 2^A\}\}, \varphi_{11} \cup \{A{=}n, B{=}2{*}n{-}1\}\rangle$
$(2)_s$ $\langle \{\{12, 2^A\}\}, \varphi_{21} \cup \{A{=}n\}\rangle$
$(3)_s$ $\langle \{\{12, 2^A\}\}, \varphi_{22} \cup \{A{=}n, C{=}b\}\rangle$
$(4)_s$ $\langle \{\{12, 2^A\}, \{10, 2^A, B\}\}, \varphi_{23} \cup \{A{=}n, B{=}a\}\rangle$
$(5)_s$ $\langle \{\{12, 2^A\}, \{10, B, 2^A\}, \{6, B, C, 2^A\}\}, \varphi_{24} \cup \{A{=}n, B{=}a, C{=}b\}\rangle$

4.2. Product Comparison

We start by providing sufficient conditions which allow proving the $\leq_\varphi$ relation on the basic cost expressions. They will later be used 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 conditions under which $e_1 \leq_\varphi e_2$. Since the sufficient conditions provided in the table are over linear expressions, we can rely on existing linear constraint solving techniques to automatically prove them. Note that the linear expressions l_1, l_2 and l in entries 1, 3, 4 and 5

e_1	e_2	sufficient condition	adiv
l_1	l_2	$\varphi \models \{l_1{>}0, l_1{\le}l_2\}$	1
n	$\log_a(A{+}1)$	$\varphi \models \{a^n \le A{+}1\}$	1
l	A^n	$n > 1 \wedge \varphi \models \{l \le A\}$	A^{n-1}
l	$n^{l'}$	$n > 1 \wedge \varphi \models \{l \le l'\}$	$n^{l'-l}$
$\log_a(A{+}1)$	l	$\varphi \models \{l{>}0, A{+}1 \le l\}$	1
$\log_a(A{+}1)$	$\log_b(B{+}1)$	$a \ge b \wedge \varphi \models \{A \le B\}$	1
$\log_a(A{+}1)$	B^n	$n > 1 \wedge \varphi \models \{A{+}1 \le B\}$	B^{n-1}
$\log_a(A{+}1)$	n^l	$n > 1 \wedge \varphi \models \{l > 0, A + 1 \le l\}$	$n^{l-(A+1)}$
A^n	B^m	$n > 1 \wedge m > 1 \wedge n \le m \wedge \varphi \models \{A \le B\}$	B^{m-n}
A^n	m^l	$m > 1 \wedge \varphi \models \{n{*}A \le l\}$	m^{l-n*A}
n^l	$m^{l'}$	$n \le m \wedge \varphi \models \{l \le l'\}$	$m^{l'-l}$

Table 1: Sufficient conditions for proving that $e_1 \le_\varphi e_2$

respectively, stand for either a constant $n \ge 1$ or an integer variable bound in φ.

Let us explain some of the entries in the table. E.g., the first entry states that in order to prove that $l_1 \le_\varphi l_2$, the context constraint φ has to guarantee that $l_1 > 0$ and that $l_1 \le l_2$. Also, verifying that $A^n \le m^l$ is equivalent to verifying $\log_m(A^n) \le \log_m(m^l)$, which in turn is equivalent to verifying that $n * \log_m(A) \le l$ when $m > 1$ (i.e., $m \ge 2$ since m is an integer value). Therefore we can verify a stronger condition $n * A \le l$ which implies $n * \log_m(A) \le l$, since $\log_m(A) \le A$ when $m \ge 2$. As another example, in order to verify that $l \le n^{l'}$ it is enough to verify that $\log_n(l) \le l'$ when $n > 1$, which can be guaranteed if $l \le l'$. We use $e_1 \le_\varphi^{sc} e_2$ to denote that we can prove that $e_1 \le_\varphi e_2$ using the sufficient conditions in Table 1.

The "part" of e_2 which is not required in order to prove the above relation becomes the *under-approximated quotient* of the comparison operation, denoted $\mathsf{adiv}(e_1, e_2)$ and it is shown in the fourth column of Table 1. An essential idea in our approach is that adiv is a cost expression in our language and hence we can transitively apply our techniques to it. This requires having an under-approximated quotient instead of the exact one. For instance, when we compare $A \le_\varphi 2^B$, where $\varphi = \{1 \le A, A \le B\}$, the under-approximated quotient for $\frac{2^B}{A}$ is $\mathsf{adiv}(A, 2^B) = 2^{B-A}$, what means that $2^{B-A} \le_\varphi \frac{2^B}{A}$. Thus, in order to prove $e{*}A \le_\varphi e'{*}2^B$ it would be enough to check $e \le_\varphi e'{*}2^{B-A}$. Note that $e \le_\varphi e'{*}2^{B-A}$ implies $e \le_\varphi e'{*}2^{B-A} \le \frac{2^B}{A}$, i.e., $e{*}A \le_\varphi e'{*}2^B$.

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 $\le_\varphi$ 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 approximated quotient in $\mathcal{M}_2$ and use it for future comparisons if needed.

Definition 13 (product comparison reduction step). *Given $\mathcal{M}_1, \mathcal{M}_2$ in $\mathcal{P}_b$ and a satisfiable context constraint φ, we define a* product comparison reduction step *$\tau_* : (\mathcal{P}_b, \mathcal{P}_b) \mapsto (\mathcal{P}_b, \mathcal{P}_b)$ as follows: $\tau_*(\mathcal{M}_1, \mathcal{M}_2) = (\mathcal{M}_1 - \{e_1\}, \mathcal{M}_2 - \{e_2\} \cup \{\mathsf{adiv}(e_1, e_2)\})$ provided that $e_1 \in \mathcal{M}_1$, $e_2 \in \mathcal{M}_2$, and $e_1 \le_\varphi^{sc} e_2$.*

In each iteration, a product comparison reduction step selects a basic cost expression from each product being compared such that the context constraint allows proving the sufficient condition provided in Table 1. This process is repeated iteratively until the left hand side expression becomes empty, in which case the less or equal property is proved, or no more pairs of expressions satisfy the required sufficient conditions, in which case the property is not proved. 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. However, τ_* is not deterministic, since at each iteration there may be more than one pair of basic expressions whose sufficient conditions are satisfied. Therefore, if the property is not proved, other choices may be investigated until a proof is found or all possible paths are explored. The number of paths is finite since at each iteration the number of alternative pairs is finite.

Example 7. *Let us consider the product* $\{8, \log_2(1{+}B), 2^A\}$ *which is part of* $(1)_s$ *in Example 6. 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* τ_* *three times. In the first iteration, since we know by Table 1 that* $\log_2(1{+}B) \leq^{sc}_{\varphi} 2^{3*B}$ *and the* adiv *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* $2^A \leq^{sc}_{\varphi} 2^{2*B-1}$*, and add as* adiv $2^{2*B-A-1}$*. Finally, it remains to be checked that* $8 \leq^{sc}_{\varphi} 2^{2*B-A-1}$*. This problem is reduced to checking that* $\varphi \models 8 \leq 2{*}B{-}A{-}1$*, which it trivially true.*

The lemma below, used later, establishes the relation between $\leq_{\varphi}$ and $\leq^{sc}_{\varphi}$, and concretely shows that $\leq^{sc}_{\varphi}$ is a sound approximation of $\leq_{\varphi}$.

Lemma 1. *Let* e, e' *be two flat cost expressions. If* $e \leq^{sc}_{\varphi} e'$ *and* $\varphi \models e \geq 1$*, then it holds that* $\varphi \models \frac{e'}{e} \geq \mathsf{adiv}(e, e') \geq 1$*.*

Proof. We proceed by inspecting all possible entries in Table 1.

- $e{=}l_1$, $e'{=}l_2$, $\varphi \models \{l_1 \leq l_2\}$ and $\mathsf{adiv}(l_1, l_2) = 1$. Then it is trivial that $\varphi \models \frac{l_2}{l_1} \geq 1$.
- $e{=}n$, $e'{=}\log_a(A+1)$, $\varphi \models \{a^n \leq A+1\}$ and $\mathsf{adiv}(l_1, l_2) = 1$. Assume that $\log_a(A+1){=}x$. Then $a^x{=}A{+}1$. But $\varphi \models A{+}1 \geq a^n$, hence $\varphi \models a^x \geq a^n$. Since by definition $a \geq 2$ and $n \geq 1$, then $\varphi \models x \geq n$, i.e., $\varphi \models \log_a(A{+}1) \geq n$, i.e., $\varphi \models \dfrac{\log_a(A+1)}{n} \geq 1$.
- $e{=}l$, $e'{=}A^n$, $n > 1 \wedge \varphi \models \{l \leq A\}$ and $\mathsf{adiv}(l, A^n){=}A^{n-1}$. Then, since $\varphi \models \{l \leq A\}$, it holds that $\varphi \models \frac{A^n}{l} \geq \frac{A^n}{A}{=}A^{n-1}$. Since $\varphi \models A \geq 1$ and $n > 1$, then $\varphi \models A^{n-1} \geq 1$.
- $e{=}l$, $e'{=}n^{l'}$, $n > 1 \wedge \varphi \models \{l \leq l'\}$ and $\mathsf{adiv}(l, n^{l'}){=}n^{l'-l}$. Note that since $\varphi \models l \leq n^l$ then $\varphi \models \frac{n^{l'}}{l} \geq \frac{n^{l'}}{n^l}{=}n^{l'-l}$. Furthermore since $\varphi \models l' \geq l$ then $\varphi \models n^{l'-l} \geq 1$.

- $e{=}\log_a(A{+}1)$, $e'{=}l$, $\varphi \models \{l > 0, A{+}1 \leq l\}$. In this case we have that $\varphi \models \log_a(A{+}1) \leq l$. Hence $\varphi \models \frac{l}{\log_a(A{+}1)} \geq 1$.

- $e{=}\log_a(A{+}1)$, $e'{=}\log_b(B{+}1)$, $a \geq b \wedge \varphi \models \{A \leq B\}$. It holds $\varphi \models \log_a(A{+}1) \leq \log_b(B{+}1)$. Hence $\varphi \models \frac{\log_b(B+1)}{\log_a(A+1)} \geq 1$.

- $e{=}\log_a(A{+}1)$, $e'{=}B^n$, $n > 1 \wedge \varphi \models \{A{+}1 \leq B\}$ and $\mathsf{adiv}(\log_a(A{+}1), B^n){=}B^{n-1}$. Since $\varphi \models \{A{+}1 \leq B\}$ then $\varphi \models \log_a(A{+}1) \leq B$. Hence $\varphi \models \frac{B^n}{\log_a(A+1)} \geq \frac{B^n}{B}{=}B^{n-1}$. Now since $\varphi \models B \geq 1$ and $n > 1$ then $\varphi \models B^{n-1} \geq 1$.

- $e{=}\log_a(A{+}1)$, $e'{=}n^l$, $n > 1 \wedge \varphi \models l > 0 \wedge A{+}1 \leq l$ and $\mathsf{adiv}(\log_a(A{+}1), n^l)$ ${=}n^{l-(A+1)}$. It holds $\varphi \models \log_a(A{+}1) \leq A{+}1$. Since $\varphi \models A \geq 1$ and $n > 1$ then $\varphi \models A{+}1 \leq n^{A+1}$. Then $\varphi \models \frac{n^l}{\log_a(A{+}1)} \geq \frac{n^l}{n^{A+1}}{=}n^{l-(A+1)}$. Finally $\varphi \models A{+}1 \leq l$ then $\varphi \models l{-}(A{+}1) \geq 0$. Since $n \geq 1$ then $\varphi \models n^{l-(A+1)} \geq 1$.

- $e{=}A^n$, $e'{=}B^m$, $n > 1 \wedge m > 1 \wedge n \leq m \wedge \varphi \models \{A \leq B\}$, and $\mathsf{adiv}(A^n, B^m){=}B^{m-n}$. Then $\varphi \models A^n \leq B^n$. Hence $\varphi \models \frac{B^m}{A^n} \geq \frac{B^m}{B^n}{=}B^{m-n}$. Since $n \leq m$ and $\varphi \models B \geq 1$ then $\varphi \models B^{m-n} \geq 1$.

- $e{=}A^n$, $e'{=}m^l$, $m > 1 \wedge \varphi \models \{n{*}A \leq l\}$ and $\mathsf{adiv}(A^n, m^l){=}m^{l-n*A}$. It holds that $\varphi \models \log_m(A) \leq A$. Hence $\varphi \models n{*}\log_m(A) \leq n{*}A$. Then $\varphi \models \log_m(A^n) \leq \log_m(m^{n*A})$ and thus $\varphi \models A^n \leq m^{n*A}$. Now, it holds that $\varphi \models \frac{m^l}{A^n} \geq \frac{m^l}{m^{n*A}}{=}m^{l-(n*A)}$. Furthermore $\varphi \models n{*}A \leq l$ and $m > 1$. Hence $\varphi \models m^{l-(n*A)} \geq 1$.

- $e{=}n^l$, $e'{=}m^{l'}$, $n \leq m \wedge \varphi \models \{l \leq l'\}$ and $\mathsf{adiv}(n^l, m^{l'}){=}m^{l-l'}$. Hence $\varphi \models n^l \leq m^l$. Thus $\varphi \models \frac{m^{l'}}{n^l} \geq \frac{m^{l'}}{m^l}{=}m^{l'-l}$. Note also that $\varphi \models l \leq l'$. Then $\varphi \models m^{l-l'} \geq 1$ □

The following proposition 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 Example 7 above.

Proposition 1. *Given $\mathcal{M}_1, \mathcal{M}_2 \in \mathcal{P}_b$ and a satisfiable context constraint φ such that for all $e \in \mathcal{M}_1$ it holds that $\varphi \models e \geq 1$, we have that if $fp_*(\mathcal{M}_1, \mathcal{M}_2) = (\emptyset, _)$ then $\mathcal{M}_1 \leq_\varphi \mathcal{M}_2$.*

Proof. We proceed by induction on the number n of applications of fp_*.

(Base case (n=0)). Then $\mathcal{M}_1 = \emptyset$ and the result holds trivially since $\emptyset$ stands for the constant 1 and $1 \leq_\varphi \mathcal{M}_2$.

(Inductive case). Let us assume that the proposition holds for any $n \geq 0$ applications of fp_*. Consider now $n+1$ applications of fp_* where we distinguish the first application:

$$fp_*{}^1(\mathcal{M}_1, \mathcal{M}_2) = (\mathcal{M}_1 - \{e_1\}, (\mathcal{M}_2 - \{e'_1\}) \cup \{\mathsf{adiv}(e_1, e'_1)\})$$
$$fp_*{}^n(\mathcal{M}_1 - \{e_1\}, (\mathcal{M}_2 - \{e'_1\}) \cup \{\mathsf{adiv}(e_1, e'_1)\}) = (\emptyset, _)$$

where $e_1 \leq^{sc}_{\varphi} e'_1$. By induction hypothesis:

$$(\clubsuit)\ \mathcal{M}_1 - \{e_1\} \leq_{\varphi} \mathcal{M}_2 - \{e'_1\} \cup \{\mathsf{adiv}(e_1, e'_1)\}$$

Suppose that $\mathcal{M}_1 = e_1 * \ldots * e_m$ and $\mathcal{M}_2 = e'_1 * \ldots * e'_k$. From ($\clubsuit$) we have that $e_2 * \ldots * e_m \leq_{\varphi} \mathsf{adiv}(e_1, e'_1) * e'_2 * \ldots * e'_k$. Then, since $\varphi \models e_1 \geq 1$, it also holds $e_1 * e_2 * \ldots * e_m \leq_{\varphi} e_1 * \mathsf{adiv}(e_1, e'_1) * e'_2 * \ldots * e'_k$.

From Lemma 1, since $e_1 \leq^{sc}_{\varphi} e'_1$ and $\varphi \models e_1 \geq 1$, then $\varphi \models e_1 * \mathsf{adiv}(e_1, e'_1) \leq e'_1$. Thus $e_1 * e_2 * \ldots * e_m \leq_{\varphi} e'_1 * e'_2 * \ldots * e'_k$ and the result holds. □

4.3. Comparison of Sums of Products

We now aim at comparing two sums of products by relying on the product comparison of Section 4.2. In this case we are interested in having a notion of *approximated difference* when comparing products. The idea is that when we want to prove $k_1 * A \leq k_2 * B$, where $A \leq B$ 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 equal 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 14 (sum comparison reduction step). *Given $\mathcal{S}_1, \mathcal{S}_2 \in \mathcal{P}_{\mathcal{M}}$ and a satisfiable context constraint φ, we define a* sum comparison reduction step *$\tau_+ : (\mathcal{P}_{\mathcal{M}}, \mathcal{P}_{\mathcal{M}}) \mapsto (\mathcal{P}_{\mathcal{M}}, \mathcal{P}_{\mathcal{M}})$ as follows: $\tau_+(\mathcal{S}_1, \mathcal{S}_2) = (\mathcal{S}_1 - \{\mathcal{M}_1\}, (\mathcal{S}_2 - \{\mathcal{M}_2\}) \cup \mathcal{A})$ provided that $\mathcal{M}_1 \in \mathcal{S}_1$, $\mathcal{M}_2 \in \mathcal{S}_2$ and $fp_*(\mathcal{M}_1, \mathcal{M}_2) = (\emptyset, _)$ where:*

$$\mathcal{A} = \begin{cases} \emptyset & \mathsf{adiff}(\mathcal{M}_1, \mathcal{M}_2) \leq 0 \\ \{(\mathcal{M}_2 - \{\mathsf{constant}(\mathcal{M}_2)\}) \cup \{\mathsf{adiff}(\mathcal{M}_1, \mathcal{M}_2)\}\} & \textit{otherwise} \end{cases}$$

In order to compare sums of products, we apply 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 $fp_+(\mathcal{S}_1, \mathcal{S}_2)$. Again, τ_+ is not deterministic, since at each iteration there may be several pairs on which to apply the reduction step.

Example 8. *Let us consider the sum of products $(5)_s$ in Example 6 together with $\mathcal{S} = \{\{50, C, 2^B\}, \{9, D^2, 2^B\}\}$ and the context constraint $\varphi = \{A{\leq}B, B{\leq}C, C{\leq}D\} \cup \varphi_{24} \cup \{A{=}n, B{=}a, C{=}b\}$. We can prove that $(5)_s \leq_{\varphi} \mathcal{S}$ by applying τ_+ three times as follows:*

1. $\tau_+((5)_s,\mathcal{S}) = ((5)_s - \{\{12, 2^A\}\}, \mathcal{S}')$, *where* $\mathcal{S}' = \{\{38, C, 2^B\}, \{9, D^2, 2^B\}\}$. *This application of* τ_+ *is feasible since* $fp_*(\{12, 2^A\}, \{50, C, 2^B\}) = (\emptyset, _)$ *in the context* φ *and the difference constant part of such comparison is* 38.
2. *Now, we perform one more iteration of* τ_+ *and obtain as result* $\tau_+((5)_s - \{\{12, 2^A\}\}, \mathcal{S}') = ((5)_s - \{\{12, 2^A\}, \{10, B, 2^A\}\}, \mathcal{S}'')$, *where* $\mathcal{S}'' = \{\{28, C, 2^B\}, \{9, D^2, 2^B\}\}$. *Observe that in this case* $fp_*(\{10, B, 2^A\}, \{38, C, 2^B\}) = (\emptyset, _)$.
3. *Finally, one more iteration of* τ_+ *on the above sum of products, gives* $(\emptyset, \mathcal{S}''')$ *as result, where* $\mathcal{S}''' = \{\{28, C, 2^B\}, \{3, D^2, 2^B\}\}$.

In this last iteration we have used the fact that $\varphi \models B \leq D$ *in order to prove that* $fp_*(\{6, B, C, 2^A\}, \{9, D^2, 2^B\}) = (\emptyset, _)$ *within the context* φ.

Theorem 4. *Let* $\mathcal{S}_1$, $\mathcal{S}_2$ *be two sums of products and* φ *a satisfiable context constraint such that for all* $\mathcal{M} \in \mathcal{S}_1$, $e \in \mathcal{M}$ *it holds that* $\varphi \models e \geq 1$. *If* $fp_+(\mathcal{S}_1, \mathcal{S}_2) = (\emptyset, _)$ *then* $\mathcal{S}_1 \leq_\varphi \mathcal{S}_2$.

Proof. We proceed by induction on the number n of applications of fp_+.

(Base case (n=0)). Then $\mathcal{S}_1 = \emptyset$ and the result holds trivially since $\emptyset$ stands for the constant 0.

(Inductive case). Let us assume that the theorem holds for any $n \geq 0$ applications of fp_+. Consider now $n+1$ applications of fp_+ and let us focus on the first step:

$$fp_+{}^1(\mathcal{S}_1, \mathcal{S}_2) = (\mathcal{S}_1 - \{\mathcal{M}_1\}, (\mathcal{S}_2 - \{\mathcal{M}'_1\}) \cup \mathcal{A})$$
$$fp_+{}^n(\mathcal{S}_1 - \{\mathcal{M}_1\}, (\mathcal{S}_2 - \{\mathcal{M}'_1\}) \cup \mathcal{A}) = (\emptyset, _)$$

where, from Proposition 1, we have that $\mathcal{M}_1 \leq_\varphi \mathcal{M}'_1$. By induction hypothesis it holds:

(♠) $\mathcal{S}_1 - \{\mathcal{M}_1\} \leq_\varphi \mathcal{S}_2 - \{\mathcal{M}'_1\} \cup \mathcal{A}$

We distinguish two cases:

1. $\mathcal{A} = \emptyset$. Then the result follows from (♠) together with the fact that $\mathcal{M}_1 \leq_\varphi \mathcal{M}'_1$.
2. $\mathcal{A} \neq \emptyset$. Let us assume that $\mathcal{S}_1 = \{\mathcal{M}_1, \ldots, \mathcal{M}_m\}$, $\mathcal{S}_2 = \{\mathcal{M}'_1, \ldots, \mathcal{M}'_k\}$, $\mathcal{M}_1 = k_1 * e_1 * \ldots * e_l$, $\mathcal{M}'_1 = k'_1 * e'_1 * \ldots * e'_{l'}$ and $\mathcal{A} = \{(k'_1 - k_1) * e'_1 * \ldots * e'_{l'}\}$. From (♠) we know that $\{\mathcal{M}_2, \ldots, \mathcal{M}_m\} \leq_\varphi \mathcal{A} \cup \{\mathcal{M}'_2, \ldots, \mathcal{M}'_k\}$. Hence, it holds also that:

   (§) $\{\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_m\} \leq_\varphi \{\mathcal{M}_1\} \cup \mathcal{A} \cup \{\mathcal{M}'_2, \ldots, \mathcal{M}'_k\}$

   Then it is enough to check that $\mathcal{M}_1 + \mathcal{A} \leq_\varphi \mathcal{M}'_1$ to get the result. Again we distinguish two cases:

(a) $fp_*(\mathcal{M}_1, \mathcal{M}'_1)$ has compared k_1 with k'_1, i.e., $k'_1{=}k_1{+}a$. Then $e_1{*}\ldots{*}e_l \leq_\varphi e'_1{*}\ldots{*}e'_{l'}$. Let us consider any valuation σ such that $\sigma \models \varphi$. We want to prove that:

$$k_1{*}\sigma(e_1){*}\ldots{*}\sigma(e_l){+}a{*}\sigma(e'_1){*}\ldots{*}\sigma(e'_{l'}){\leq}$$
$$(k_1{+}a){*}\sigma(e'_1){*}\ldots{*}\sigma(e'_{l'})$$

But this is true because of $e_1{*}\ldots{*}e_l \leq_\varphi e'_1{*}\ldots{*}e'_{l'}$.

(b) $fp_*(\mathcal{M}_1, \mathcal{M}'_1)$ has compared k_1 with another basic expression different from k'_1. According to Table 1, then the constant k'_1 has not been used in $fp_*(\mathcal{M}_1, \mathcal{M}'_1)$, i.e., $k_1{*}e_1{*}\ldots{*}e_l \leq_\varphi e'_1{*}\ldots{*}e'_{l'}$. Hence, for any valuation σ such that $\sigma \models \varphi$, it trivially holds that:

$$k_1{*}\sigma(e_1){*}\ldots{*}\sigma(e_l) + a{*}\sigma(e'_1){*}\ldots{*}\sigma(e'_{l'}) \leq$$
$$(k_1 + a){*}\sigma(e'_1){*}\ldots{*}\sigma(e'_{l'})$$

□

Example 9. *For the sum of products in Example 8, we get* $fp_+((5)_s, \mathcal{S}) = (\emptyset, \mathcal{S}''')$*. Thus, according to the above theorem, it holds that* $(5)_s \leq_\varphi \mathcal{S}$*.*

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 15 (sum-product comparison reduction step). *Consider* $\mathcal{S}_1$ *and* $\mathcal{M}_2$*, where* $\mathcal{S}_1 \in \mathcal{P}_\mathcal{M}$*,* $\mathcal{M}_2 \in \mathcal{P}_b$*,* φ *is a satisfiable context constraint and for all* $\mathcal{M} \in \mathcal{S}_1$ *it holds that* $\varphi \models \mathcal{M}{>}1$*. Then, we define a* sum-product comparison reduction step $\tau_{(+,*)} : (\mathcal{P}_\mathcal{M}, \mathcal{P}_b) \mapsto (\mathcal{P}_\mathcal{M}, \mathcal{P}_b)$ *as follows:* $\tau_{(+,*)}(\mathcal{S}_1, \mathcal{M}_2) = (\mathcal{S}_1 - \{\mathcal{M}'_2\}, \mathcal{M}''_2)$ *provided that* $fp_*(\mathcal{M}'_2, \mathcal{M}_2) = (\emptyset, \mathcal{M}''_2)$*.*

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

Example 10. *Note that* $fp_{(+,*)}(\{\{A^3\}, \{A^2\}, \{A\}\}, \{A^6\})$ *returns* $(\emptyset, \emptyset)$ *w.r.t. the context constraint* $\varphi = \{A > 1\}$*. To this end, we apply* $\tau_{(+,*)}$ *three times. In the first iteration,* $fp_*(\{A^3\}, \{A^6\}) = (\emptyset, \{A^3\})$*. In the second iteration,* $fp_*(\{A^2\}, \{A^3\}) = (\emptyset, \{A\})$*. Finally in the third iteration* $fp_*(\{A\}, \{A\}) = (\emptyset, \emptyset)$*.*

When using $fp_{(+,*)}$ to compare sums of products, we can take advantage of having an approximated difference similar to the one defined in Section 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 $\mathsf{constant}(\mathcal{S})$ is defined as $\sum_{\mathcal{M}' \in \mathcal{S}} \mathsf{constant}(\mathcal{M}')$. Thus, if we compare $\{\{A^3\}, \{A^2\}, \{A\}\}$ with $\{4, A^6\}$, we can have as approximated difference 1, whose intended meaning is that after the comparison, $\{1, A^6\}$ remains from $\{4, A^6\}$ and it could be useful to reduce further addends.

Next we define the general sum reduction step, which illustrates the idea explained in the paragraph above. As notation, we use $\mathcal{P}_\mathcal{S}$ to denote the set $2^{\mathcal{P}_\mathcal{M}}$ and $\mathcal{S}_s$ to refer to one element of $\mathcal{P}_\mathcal{S}$.

Definition 16 (general sum comparison reduction step). *Let us consider $\mathcal{S}_s$ and $\mathcal{S}_2$, where $\mathcal{S}_s \in \mathcal{P}_\mathcal{S}$, $\mathcal{S}_2 \in \mathcal{P}_\mathcal{M}$ and φ is a satisfiable context constraint. We define a general sum comparison reduction step $\mu_+ : (\mathcal{P}_\mathcal{S}, \mathcal{P}_\mathcal{M}) \mapsto (\mathcal{P}_\mathcal{S}, \mathcal{P}_\mathcal{M})$ as follows: $\mu_+(\mathcal{S}_s, \mathcal{S}_2) = (\mathcal{S}_s - \{\mathcal{S}_1\}, (\mathcal{S}_2 - \{\mathcal{M}\}) \cup \mathcal{A})$, provided that $\mathcal{S}_1 \in \mathcal{S}_s$, $\mathcal{M} \in \mathcal{S}_2$, and $fp_{(+,*)}(\mathcal{S}_1, \mathcal{M}) = (\emptyset, _)$ where:*

$$\mathcal{A} = \begin{cases} \emptyset & \mathsf{adiff}(\mathcal{S}_1, \mathcal{M}) \leq 0 \\ \{(\mathcal{M} - \{\mathsf{constant}(\mathcal{M})\}) \cup \{\mathsf{adiff}(\mathcal{S}_1, \mathcal{M})\}\} & \textit{otherwise} \end{cases}$$

Similarly as we have done in previous definitions, 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 use $fp^g_+(\mathcal{S}_s, \mathcal{S}_2)$ to denote the result of applying μ_+ until there are no sets to transform.

Observe that the above reduction step does not replace the previous sum comparator reduction step in Definition 14 since it sometimes can be of less applicability, as $fp_{(+,*)}$ requires that all elements in the addition are strictly greater than one. Instead, it is used in combination with Definition 14 so that when we fail to prove the comparison by using the one-to-one comparison we attempt with the sum-product comparison above.

Lemma 2. *Let φ be a satisfiable context constraint and let $\mathcal{M}_1$ and $\mathcal{M}$ be products of basic expressions such that for all $e \in \mathcal{M}_1$ it holds that $\varphi \models e \geq 1$. If $fp_*(\mathcal{M}_1, \mathcal{M}) = (\emptyset, \mathcal{M}'_1)$ then $\mathcal{M}_1 * \mathcal{M}'_1 \leq_\varphi \mathcal{M}$.*

Proof. We proceed by induction on the number n of applications of fp_*.

(Base case: (n=0)). Then $\mathcal{M}_1 = \emptyset$ and the result holds trivially.

(Inductive case). Assume that the result holds for $n \geq 0$ applications of fp_*. Consider then the following $n+1$ applications of fp_*:

(1) $fp_*{}^1(\mathcal{M}_1, \mathcal{M}) = (\mathcal{M}_1 - \{e_1\}, (\mathcal{M} - \{e'_1\}) \cup \{\mathsf{adiv}(e_1, e'_1)\})$
(2) $fp_*{}^n(\mathcal{M}_1 - \{e_1\}, (\mathcal{M} - \{e'_1\}) \cup \{\mathsf{adiv}(e_1, e'_1)\}) = (\emptyset, \mathcal{M}'_1)$

where $e_1 \leq^{sc}_\varphi e_2$, and by induction hypothesis on (2):

(§) $(\mathcal{M}_1 - \{e_1\}) * \mathcal{M}'_1 \leq_\varphi \mathcal{M} - \{e'_1\} \cup \{\mathsf{adiv}(e_1, e'_1)\}$

We want to prove that $\mathcal{M}_1{*}\mathcal{M}'_1 \leq_\varphi \mathcal{M}$. Suppose that $\mathcal{M}_1{=}e_1{*}\ldots{*}\ e_m$ and $\mathcal{M}{=}e'_1{*}\ldots{*}e'_k$. We have to ensure that for all valuations σ such that $\sigma \models \varphi$, it holds that $\sigma(e_1){*}\ldots{*}\sigma(e_m){*}\sigma(\mathcal{M}'_1){\leq}\sigma(e'_1){*}\ldots{*}\sigma(e'_k)$. We distinguish two cases:

1. $\mathsf{adiv}(e_1, e'_1){=}1$. From Lemma 1 it holds that $\varphi \models e_1{*}\mathsf{adiv}(e_1, e_2) \leq e_2$. Then the result follows from $\sigma(e_1){\leq}\sigma(e_2)$ together with (§).
2. Otherwise, from Lemma 1 it follows that $\dfrac{\sigma(e'_1)}{\sigma(e_1)}{\geq}\sigma(\mathsf{adiv}(e_1, e'_1))$. Furthermore, to prove $\sigma(e_1){*}\ldots{*}\sigma(e_m){*}\sigma(\mathcal{M}'_1){\leq}\sigma(e'_1){*}\ldots{*}\sigma(e'_k)$ is equivalent to prove that $\sigma(e_2){*}\ldots{*}\sigma(e_m){*}\sigma(\mathcal{M}'_1){\leq}\dfrac{\sigma(e'_1)}{\sigma(e_1)}{*}\sigma(e'_2){*}\ldots{*}\sigma(e'_k)$. Then:

   From (§)
   $$\sigma(e_2){*}\ldots{*}\sigma(e_m){*}\sigma(\mathcal{M}') \leq \sigma(\mathsf{adiv}(e_1, e'_1)){*}\sigma(e'_2){*}\ldots{*}\sigma(e'_k)$$
   From Lemma 1
   $$\frac{\sigma(e'_1)}{\sigma(e_1)}{*}\sigma(e'_2){*}\ldots{*}\sigma(e'_k) \geq \sigma(\mathsf{adiv}(e_1, e'_1)){*}\sigma(e'_2){*}\ldots{*}\sigma(e'_k)$$

   Hence $\sigma(e_2){*}\ldots{*}\sigma(e_m){*}\sigma(\mathcal{M}'_1) \leq \dfrac{\sigma(e'_1)}{\sigma(e_1)}{*}\sigma(e'_2){*}\ldots{*}\sigma(e'_k)$ and the result holds. □

Lemma 3. *Let φ be a satisfiable context constraint. Given $\mathcal{S}$ and $\mathcal{M}$, where $\mathcal{S} \in \mathcal{P}_\mathcal{M}$, $\mathcal{M}' \in \mathcal{P}_b$, $\varphi \models \mathcal{M} > 1 \wedge \mathcal{S} > 1 \wedge \mathcal{M}' > 1$. If $\mathcal{S} \leq_\varphi \mathcal{M}'$ then $\mathcal{S}{+}\mathcal{M}_1 \leq_\varphi \mathcal{M}'{*}\mathcal{M}_1$.*

Proof. If $\mathcal{S} \leq_\varphi \mathcal{M}'$ then for all $\sigma \models \varphi$, it holds that $\sigma(\mathcal{S}) \leq \sigma(\mathcal{M}')$. But since $\varphi \models \mathcal{M} > 1 \wedge \mathcal{S} > 1 \wedge \mathcal{M}' > 1$, then $\sigma(\mathcal{M}) > 1$, $\sigma(\mathcal{S}) > 1$ and $\sigma(\mathcal{M}') > 1$. Then:

$$\begin{array}{lll} \sigma(\mathcal{S}){+}\sigma(\mathcal{M}_1) & \leq & \%\ \sigma(\mathcal{S}) > 1 \text{ and } \sigma(\mathcal{M}_1) > 1 \\ \sigma(\mathcal{S}){*}\sigma(\mathcal{M}_1) & \leq & \%\ \sigma(\mathcal{S}) \leq \sigma(\mathcal{M}') \text{ and } \sigma(\mathcal{M}') > 1 \\ \sigma(\mathcal{M}'){*}\sigma(\mathcal{M}) & & \end{array}$$

Hence, $\mathcal{S}{+}\mathcal{M}_1 \leq_\varphi \mathcal{M}'{*}\mathcal{M}_1$. □

Lemma 4. *Let φ be a satisfiable context constraint. Given $\mathcal{S}$ and $\mathcal{M}$, where $\mathcal{S} \in \mathcal{P}_\mathcal{M}$, $\mathcal{M} \in \mathcal{P}_b$ and for all $\mathcal{M}' \in \mathcal{S}$, $e \in \mathcal{M}'$, it holds that $\varphi \models e > 1$. If $fp_{(+,*)}(\mathcal{S}, \mathcal{M}) = (\emptyset, _)$ then $\mathcal{S} \leq_\varphi \mathcal{M}$.*

Proof. We proceed by induction on the number n of applications of $fp_{(+,*)}$.

(Base case (n=0)). Then $\mathcal{S} = \emptyset$ and the result holds trivially.

(Inductive case). Let us suppose that the result holds for all $n{\geq}0$ applications of $fp_{(+,*)}$. Let us consider $n{+}1$ applications of $fp_{(+,*)}(\mathcal{S}, \mathcal{M})$. We separate the first iteration from the remaining ones.

$$\begin{array}{lll} (1) & {fp_{(+,*)}}^1(\mathcal{S}, \mathcal{M}) = (\mathcal{S} - \{\mathcal{M}_1\}, \mathcal{M}'_1) & \%\ \text{one step} \\ (2) & {fp_{(+,*)}}^n(\mathcal{S} - \{\mathcal{M}_1\}, \mathcal{M}'_1) = (\emptyset, _) & \%\ n \text{ steps} \end{array}$$

where $fp_*(\mathcal{M}_1, \mathcal{M}) = (\emptyset, \mathcal{M}'_1)$. By induction hypothesis on (2), it holds that $\mathcal{S} - \{\mathcal{M}_1\} \leq_\varphi \mathcal{M}'_1$. Applying Proposition 1 to $fp_*(\mathcal{M}_1, \mathcal{M}) = (\emptyset, \mathcal{M}'_1)$ we obtain that $\mathcal{M}_1 \leq_\varphi \mathcal{M}$.

Let us suppose that $\mathcal{S} = \{\mathcal{M}_1, \ldots, \mathcal{M}_k\}$, written as $\mathcal{M}_1 + \ldots + \mathcal{M}_k$. Then:

$$\begin{array}{lcl} \mathcal{M}_2 + \ldots + \mathcal{M}_k \leq_\varphi \mathcal{M}'_1 & \Rightarrow & \text{\% Lemma 3} \\ \mathcal{M}_1 + \mathcal{M}_2 + \ldots + \mathcal{M}_k \leq_\varphi \mathcal{M}'_1 * \mathcal{M}_1 & \Rightarrow & \text{\% Lemma 2:} \\ & & \mathcal{M}'_1 * \mathcal{M}_1 \leq_\varphi \mathcal{M} \\ \mathcal{S} \leq_\varphi \mathcal{M} & & \end{array}$$

□

The following theorem establishes the correctness of the composite comparison.

Theorem 5 (composite inclusion). *Let φ be a satisfiable context constraint. Let $\mathcal{S}_1$, $\mathcal{S}_2$ be two sums of products such that for all $\mathcal{M}' \in \mathcal{S}_1$, $e \in \mathcal{M}'$ it holds $\varphi \models e > 1$. Let $\mathcal{S}_s$ be a partition of $\mathcal{S}_1$. If $fp^g_+(\mathcal{S}_s, \mathcal{S}_2) = (\emptyset, _)$ then $\mathcal{S}_1 \leq_\varphi \mathcal{S}_2$.*

Proof. We proceed by induction on the number n of applications of fp^g_+.

(Base case (n=0)). If $\mathcal{S}_s = \emptyset$ then the result holds trivially.

(Inductive case). Let us suppose the result holds for $n \geq 1$ and let us analyze the case $n+1$. We separate the first step from the rest ones. Thus:

(1) $fp^{g\,1}_+(\mathcal{S}_s, \mathcal{S}_2) = (\mathcal{S}_s - \{\mathcal{S}'\}, (\mathcal{S}_2 - \{\mathcal{M}\}) \cup \mathcal{A})$

where $fp_{(+,*)}(\mathcal{S}', \mathcal{M}) = (\emptyset, _)$ and $\mathcal{A} = \emptyset$ if $\mathsf{adiff}(\mathcal{S}', \mathcal{M}) \leq 0$; otherwise $\mathcal{A} = \{(\mathcal{M} - \{\mathsf{constant}(\mathcal{M})\}) \cup \{\mathsf{adiff}(\mathcal{S}', \mathcal{M})\}\}$. Let us consider now the remaining n applications of fp^g_+:

(2) $(fp^g_+)^n(\mathcal{S}_s - \{\mathcal{S}'\}, (\mathcal{S}_2 - \{\mathcal{M}\}) \cup \mathcal{A}) = (\emptyset, _)$

By induction hypothesis on (2): $\mathcal{S}_s - \{\mathcal{S}'\} \leq_\varphi (\mathcal{S}_2 - \{\mathcal{M}\}) \cup \mathcal{A}$. Now, since $fp_{(+,*)}(\mathcal{S}', \mathcal{M}) = (\emptyset, _)$, then from Lemma 4 it holds that $\mathcal{S}' \leq_\varphi \mathcal{M}$. We distinguish two cases:

- $\mathsf{adiff}(\mathcal{S}', \mathcal{M}) \leq 0$. Then:

  By induction hypothesis: $\mathcal{S}_s - \{\mathcal{S}'\} \leq_\varphi \mathcal{S}_2 - \{\mathcal{M}\} \quad \Rightarrow$
  By $\mathcal{S}' \leq_\varphi \mathcal{M}$: $\underbrace{(\mathcal{S}_s - \{\mathcal{S}'\}) \cup \{\mathcal{S}'\}}_{\mathcal{S}_1} \leq_\varphi (\mathcal{S}_2 - \{\mathcal{M}\}) \cup \{\mathcal{M}\}$

  Hence, it holds that $\mathcal{S}_1 \leq_\varphi \mathcal{S}_2$.

- $\mathcal{A} = \{(\mathcal{M} - \{\mathsf{constant}(\mathcal{M})\}) \cup \{\mathsf{adiff}(\mathcal{S}', \mathcal{M})\}\}$. Then, let us consider the multiset $\mathcal{M}'' = (\mathcal{M} - \mathsf{constant}(\mathcal{M})) \cup \{\mathsf{constant}(\mathcal{S}')\}$. Because of definition of $fp_{(+,*)}$ and Table 1, it holds also that:

  (§) $fp_{(+,*)}(\mathcal{S}', \mathcal{M}'') = (\emptyset, _)$

Thus, from Lemma 4 we have that $\mathcal{S}' \leq_\varphi \mathcal{M}''$. Now it is enough to reason as follows:

$$\begin{array}{l} \text{By induction hypothesis: } \mathcal{S}_s - \{\mathcal{S}'\} \leq_\varphi (\mathcal{S}_2 - \{\mathcal{M}\}) \cup \mathcal{A} \quad \Rightarrow \\ \text{By } (\S): \underbrace{(\mathcal{S}_s - \{\mathcal{S}'\}) \cup \{\mathcal{S}'\}}_{\mathcal{S}_s} \leq_\varphi \underbrace{((\mathcal{S}_2 - \{\mathcal{M}\}) \cup \mathcal{A}) \cup \{\mathcal{M}''\}}_{\mathcal{M}} \end{array}$$

□

5. Applications of Cost Function Comparators

In all applications of resource analysis, such as resource-usage verification, certification, program synthesis and optimization, it is essential to compare cost functions. In this section, we discuss how our cost function comparator can be used in the different applications.

5.1. Checking Effect of Program Optimization, Program Synthesis

Program Optimization is a research field which has received considerable attention and where technical advances can have significant impact. The aim of program optimization is, given a program P and some information about the environment in which P will be executed, to obtain another program P' which has the same semantics as P and a lower resource consumption.

Program optimization can be performed at different levels. Traditionally, two levels have been studied in depth: source-level optimizations and low-level optimizations. Source-level optimization handles programs written in high-level languages and optimization is materialized in terms of source-to-source transformations. The information which is exploited for optimizing programs is usually information about particular values of the inputs to the program, as in *Partial Evaluation* [21]. Low-level optimization considers transforming compiled code and usually the information which is exploited refers to particularities of the hardware where the code will run, such as the number of registers available, etc. Our comparator is useful for optimizations at both levels, as long as automatic cost analyzers exist at such levels. Nowadays, most analyzers handle source programs or bytecode programs, the latter can be considered an intermediate level between source and low-level code.

In general, program optimization is guided by a number of heuristics and (infinitely) many optimized programs can potentially be obtained from an initial program, depending on the heuristics used and the order in which they are applied. After an initial experimentation phase, the optimization strategy is usually fixed once and for all in order to obtain a single optimized program per initial program. However, it seems clear that there is no single optimization strategy which consistently produces the best results in all cases and that better results could be obtained by using different strategies for different programs. Furthermore, when aggressive transformations are considered, as in the case of partial evaluation, it turns out to be almost impossible to determine *a priori*

which optimization strategy would produce best results. Thus, another alternative consists in considering several optimization strategies in order to produce a *set of* candidates and then choose *a posteriori* the best optimized program among all candidates produced. A few frameworks exist which try to handle multiple optimization strategies. See e.g. [13, 26]. However, a major drawback of these approaches is how to choose the best candidate, even when we have a set of candidates at hand. This is because, in general, programs can receive (infinitely) many input values. So, what is the set of input values for which we should compare the behaviour of the candidates?

The availability of automatic resource analyzers allows statically obtaining an evaluation of the performance of the different candidates in the form of *CFs* which is valid for any input value. Thus, the existence of an automatic comparator for *CFs* can in principle be used for choosing the candidate with the best resource consumption. We note that the fact that a program P_1 has a smaller UB than another program P_2, though it often does, does not necessarily mean that P_1 is more efficient than P_2 due to the inaccuracy introduced by upper approximations. However, if the goal of the optimization process is to obtain a program whose resource consumption is guaranteed to be smaller than some user-provided bounds, this *a posteriori* optimization approach may allow obtaining programs with the required resource consumption in situations where the initial implementations did not meet the resource-usage requirements.

Another application area is that of *Program Synthesis* whose goal is to obtain an efficient implementation from some initial description, usually given in a formal specification language, whose goal is clarity more than efficiency. The situation is in fact quite similar to that of program optimization, since there are (infinitely) many implementations which correspond to a given specification and the final implementation is usually obtained by applying a number of heuristic program transformations. Again, being able to compare the efficiency of different candidate implementations is of much interest.

5.2. Verification of Cost Functions

The *CFs* comparator is the basis of resource usage verification. Essentially, the user states an assertion about the efficiency of the program (given as a *CF* u) which the resource analyzer will try to verify or falsify. For this purpose, the analyzer infers an upper bound *CF* f^φ from the program and an initial context φ and then simply uses the comparator to check whether $f^\varphi \leq_\varphi u$.

For some cost models, such as the number of instructions executed, it is quite convenient to specify assertions in asympotic form, i.e., u is in big O notation, denoted u_a. In this case, we first transform f^φ into asymptotic form f_a^φ (e.g., by applying the technique in [1]) and then we check $f_a^\varphi \leq_\varphi u_a$, as our checker works equally well on asymptotic *CFs*. Let us see an example.

Example 11. *Let us assume that we want to verify that the number of instructions required by the execution of a given method* $m(x, y)$ *in the context* $\varphi = \{x{>}1, y{>}1, x{\geq}y\}$ *is in the asymptotic order* $3^{\mathsf{nat}(4*x)}$*. Suppose that, by analyzing the method, we obtain the following upper bound CF:*

$$m^+(x,y) = 7{*}\mathsf{nat}(3{*}x{+}1){*}\mathsf{max}(\{100{*}\mathsf{nat}(x)^2{*}\mathsf{nat}(y)^4, 11{*}3^{\mathsf{nat}(y-1)}{*}\mathsf{nat}(x{+}5)^2\}) + 2{*}\log_2(\mathsf{nat}(x{+}2)){*}2^{\mathsf{nat}(y-3)}{*}\log(\mathsf{nat}(y{+}4)){*}\mathsf{nat}(2{*}x{-}2{*}y)$$

Its transformation into asymptotic form results in $3^{\mathsf{nat}(y)}{*}\mathsf{nat}(x)^3$*. Using our comparator, we prove, according to Table 1, that* $3^y \leq^{sc}_{\varphi} 3^x$*, where* $\mathsf{adiv}(3^y, 3^{4*x}) = 3^{4*x-y}$*. Similarly, since* $\varphi \models x{\geq}y$*, we can prove that* $x^3 \leq^{sc}_{\varphi} 3^{4*x-y}$*. Then* $3^{\mathsf{nat}(y)}{*}\mathsf{nat}(x)^3 \leq_{\varphi} 3^{\mathsf{nat}(4*x)}$*, i.e.,* $m^+(x,y) \in O(3^{\mathsf{nat}(4*x)})$ *w.r.t. the context* φ*.*

5.3. Verification of Cost Relations

In this section, we discuss a novel application of our *CFs* comparator to verify that a *cost relation system* (*CRS* for short) is bounded by a given *CF*. A *CRS* is the output of the first phase of cost analyzers which follow the traditional approach to cost analysis by Wegbreit [28]. Essentially, given an input program, automated cost analysis generates from it a *CRS* that defines its cost by means of recursive cost equations. Let us introduce some notation. A *CRS* is a set of *cost equations* of the form $\langle C(\bar{x}) = e + \sum_{j=1}^{k} D_j(\bar{y}_j), \varphi\rangle$, where C and D_j are cost relation symbols, e is a cost expression that we accumulate, and φ is a linear constraint. Intuitively, a cost equation states that the cost of $C(\bar{x})$ is e plus the sum of the costs of $D_1(\bar{y}_1), \ldots, D_k(\bar{y}_k)$. The linear constraint φ specifies the values of $\bar{x}$ for which the equation is applicable, and defines relations among the different variables. W.l.o.g., in what follows we assume that a *CRS* includes a single cost relation symbol, i.e., it is *stand-alone*. Namely, we have $D_j{=}C$, $1 \leq j \leq k$. In order to handle *CRS* with more than one cost relation symbol, we rely on the compositional approach of [4]. In a second phase of cost analyzers, an *upper bound* from the equations is obtained which is guaranteed to be larger than or equal to all evaluations of the *CRS*. Such upper bound is given as a *CF* which is not in recursive form.

The application that we describe in this section consists in, instead of applying the second step and solving the *CRS* into a *CF*, we can verify that a given *CF* is an upper bound for the *CRS*. The motivation for this application is threefold: (1) we might not have techniques to automatically infer an upper bound from the *CRS* but we can still check that a given *CF* is an upper bound, (2) we sometimes can check that f is an upper bound for the *CRS*, and f is strictly smaller than the upper bound that an automatic analyzer can infer, and (3) we can use it in resource usage certification (see Section 5.4 below). The soundness of this application is guaranteed by [10], where it is proven that, given a cost function $f(\bar{x})$, we have that it is an upper bound of a cost relation C if, for each equation $\langle C(\bar{x}) = e + \sum_{j=1}^{k} C(\bar{y}_j), \varphi\rangle$ for C, we have that $\varphi \models \forall \bar{x}, \bar{y}_j . f(\bar{x}) \geq e + \sum_{j=1}^{k} f(\bar{y}_j)$, i.e., by replacing the given cost function in the equation, the above *CF* comparison holds.

Example 12. *Consider the following CRS which illustrates motivation (2) above:*

$$\begin{array}{lcll} C(x) & = & 0 & \{x \geq 0, x \leq 3\} \\ C(x) & = & \mathsf{nat}(x) + C(x_1) + C(x_2) & \{x = x_1 + x_2, x_1 \geq 2, x_2 \geq 2\} \end{array}$$

Using the approach in [4] (implemented in the PUBS *system) we obtain an exponential upper bound* $\mathsf{nat}(x)*(2^{\mathsf{nat}(x-1)}-1)$*. However, we have that* $\mathsf{nat}(x)*\mathsf{nat}(x)$ *is a tighter upper bound for the CRS that* PUBS *is unable to compute. The main idea is that we can verify that the above CRS admits a quadratic bound. Concretely, it is enough to check that the following two formulas are valid:*

$$\begin{array}{lll} \textbf{(a)}\ \forall x: & \{x \geq 0, x \leq 3\} & \rightarrow f(x) \geq 0 \\ \textbf{(b)}\ \forall \bar{x}: & \{x{=}x_1{+}x_2, x_1{\geq}2, x_2{\geq}2\} & \rightarrow f(x) \geq \mathsf{nat}(x){+}f(x_1){+}f(x_2) \end{array}$$

where $\bar{x} = x, x_1, x_2$ *and* $f(x)$ *stands for the CF* $\mathsf{nat}(x) * \mathsf{nat}(x)$*. Since all variables are constrained to be non-negative, the* nat*-operator can be removed. In the following, in order to clarify the presentation, we avoid the set-notation. Instead we write cost expressions as sums of products than implicitly represent sets of sets of flat CFs. Formula* ***(a)*** *can be verified by checking* $0 \leq^{sc}_{\{x\geq 0, x\leq 3\}} x{*}x$*, what is a direct consequence of Table 1. With respect to formula* ***(b)****, the checking process can be done as follows: (1) replace variable* x *by* $x_1{+}x_2$*, that returns the new formula* $\forall \bar{x} : \varphi{\rightarrow}(x_1{+}x_2){*}(x_1{+}x_2){\geq}x_1{+}x_2{+}x_1{*}x_1{+}x_2{*}x_2$*, where* $\varphi{=}\{x{=}x_1{+}x_2, x_1{\geq}2,\ x_2{\geq}2\}$*; (2) normalize the expression* $(x_1{+}x_2){*}(x_1{+}x_2)$*, resulting in* $x_1{*}x_1{+}x_2{*}x_2\ {+}2{*}x_1{*}x_2$*; (3) check* $x_1{*}x_1{+}x_2{*}x_2{+}x_1{+}x_2 \leq_\varphi x_1{*}x_1{+}x_2{*}x_2{+}2{*}x_1{*}x_2$ *by firstly removing identical addends in both sides of the inequality and secondly applying Theorem 5 to check* $x_1{+}x_2 \leq_\varphi 2{*}x_1{*}x_2$ *as follows:*

1. $\tau_*(x_1, 2{*}x_1{*}x_2) = (\{\}, 2{*}x_2)$*;*
2. $\tau_*(x_2, 2{*}x_2) = (\{\}, 2)$*;*

and thus $\tau_{(+,*)}(x_1{+}x_2, 2{*}x_1{*}x_2) = (\emptyset, _)$*.*

The next example is borrowed from [11] and used to illustrate motivation (1) above. Solving this example requires being able to handle non-linear size relations which is rather expensive and thus several systems cannot support them (e.g., PUBS cannot solve it).

Example 13. *Consider the CRS taken from [11]:*

$$\begin{array}{lcll} D(x,y) & = & 0 & \{x < 2\} \\ D(x,y) & = & 1 + D(x+y, y-1) & \{x \geq 2\} \end{array}$$

for which, if y *is initially positive, the second equation can be applied* $y{+}1$ *times until* y *takes the negative value* -1 *and* x *is equal to* x'*, where* $x' = x + \frac{y(y+1)}{2}$*. From that point, the value of* x' *begins to decrease until it reaches a negative value. The number of applications of the second equation can be bounded now by* x'*. The problem is that the number of applications of the CRS cannot be bounded by a linear function and thus* PUBS *cannot solve it. An upper bound for this CRS can be specified as follows:*

$$f(x,y) = \left\{ \begin{array}{ll} \mathsf{nat}(x) & \{y \leq -1\} \\ (y+1) + \mathsf{nat}(x) + \frac{y(y+1)}{2} & \{y \geq 0\} \end{array} \right.$$

In order to verify that $f(x,y)$ *is an upper bound, it is enough to check the validity of the formulas:*

(c) $\forall x, y: \quad \{x < 2\} \quad \rightarrow f(x,y) \geq 0$
(d) $\forall x, y \quad \{x \geq 2\} \quad \rightarrow f(x,y) \geq 1{+}f(x+y, y-1)$

In the case of formula **(c)**, *we distinguish two cases. If $y \leq -1$, then $f(x,y) = \mathsf{nat}(x)$ and $0 \leq^{sc}_{\varphi_1} \mathsf{nat}(x)$ trivially holds, where $\varphi_1 = \{x < 2, y \leq -1\}$. In case of $y \geq 0$ and $x \leq 0$, $f(x,y) = y{*}y{+}3{*}y{+}2$. Then $0 \leq_{\varphi_2} y{*}y{+}3{*}y{+}2$ holds, where $\varphi_2 = \{y \geq 0, x < 0\}$. Otherwise, if $x > 0 \wedge x < 2$, then $f(x,y) = y{*}y{+}3{*}y{+}x{+}2$ and again $0 \leq_{\varphi_3} y{*}y + 3{*}y{+}x{+}2$ holds, where $\varphi_3 = \{y \geq 0, x > 0, x < 2\}$. For the formula* **(d)** *we reason similarly. Assuming that $y \leq -1$ then $f(x,y) = x$. We need to check that $1{+}x{+}y \leq_{\varphi_4} x$, where $\varphi_4 = \{y \leq -1, x \geq 2\}$. But it holds:*

1. *According to Table 1, $\tau_*(x,x) = (\{\},1)$ then $\tau_{(+,*)}(1{+}x{+}y, x) = (1{+}y, 1)$.*
2. *Similarly, $\tau_*(1,1) = (\{\},1)$ then $\tau_{(+,*)}(1{+}y, 1) = (y, 1)$.*
3. *Finally $\tau_{(+,*)}(y,1) = (\{\},1)$, since $y \leq -1$.*

Now, Theorem 5 guarantees the result. The last case is $y \geq 0$. Then $f(x,y) = y{*}y{+}3{*}y{+}2{*}x{+}2$ and $y{*}y{+}3{*}y{+}2{*}y{+}1 \leq_{\varphi_5} y{*}y{+}3{*}y{+}2{*}x{+}2$ follows from Theorem 4, where $\varphi_5 = \{x < 0 \wedge y \geq 0\}$.

5.4. Resource Usage Certification

Resource usage certification [24, 14, 18, 7, 8] has been proposed as a way to provide *resource guarantees* in the context of *mobile code*. Mobile code includes, for example running applets and/or plug-ins downloaded from the net in a web browser or a mobile phone. The purpose of resource certification is to consider resource usage bounds as security policies. This means that prior to executing a program, it must be guaranteed that the program satisfies a given resource usage policy in a context of interest φ. As an example, let us assume that $\varphi = \{t > 0\}$ and that the resource usage policy, or *policy* for short, for the program imposes that the maximum number of instructions executed is:

$$policy{=}60 * \mathsf{nat}(t)^2 + 120 * \mathsf{nat}(t) + 13$$

Let us also assume that a resource analyzer infers the following cost function for the code at hand $ub{=}24 * \mathsf{nat}(t) * \log_2(\mathsf{nat}(t) + 1) + 53 * \mathsf{nat}(t) + 12$. The code is acceptable only if $ub \leq_{\varphi} policy$, which is the case in our example, and the comparator succeeds to prove it.

The certification problem can be formulated in two ways:

- In a traditional scenario, we have an automatic system which given a program and a resource usage policy answers *yes* only if it succeeds to prove that the program satisfies the policy. The use of our *CF* comparator is clear, it is used to check that the *CF* yielded by the analyzer satisfies the policy.

- Alternatively, the *Proof Carrying Code* (PCC) approach splits this process in two steps: first, an automatic system run by the *code producer* obtains an upper bound on the resource usage of the program. Then, the producer

provides an unsigned bundle which contains the code, the upper bound obtained for it, and some *evidence* which can be used to efficiently check that the upper bound is correct. In our case, the evidence consists in the upper bounds computed for the CRSs in the program. Then, the *code consumer* has to have an automatic (and efficient) system for checking that the provided upper bound is actually valid for the code, by using the provided evidence.

As it is well known from the proof-carrying code [24] theory, the main advantage of the second scenario, i.e., PCC, is that the evidence only needs to be generated once and the checking process which occurs at the consumer side should be more efficient than computing the upper bounds from scratch. Essentially, the hard work is shifted from the code consumer to the code producer (i.e., the programmer and/or the compiler), which now has to not only produce the code, but also infer an upper bound and bundle both together with the evidence.

The consumer, instead of inferring the upper bound, has to check that the upper bound provided is correct. To do so, the consumer will set up a CRS from the program in a fully automatic way (see [5]) and will then check that the *CFs* provided in the evidence are sound upper bounds for this CRS (by using the checking process described in Section 5.3 above). Thus, our *CF* comparator is used to check that the provided upper bounds constitute a valid certificate of the program's efficiency. In essence, the process of inferring an upper bound from the CRS is more expensive than checking its validity by using our comparator. Finally, the code will be acceptable, provided that *policy* is guaranteed, i.e., $cert \leq_{\varphi} policy$, where *cert* are the upper bounds in the certificate. Again, we will use the comparator to prove this final step.

5.5. Simplification of Cost Expressions

As already discussed, cost expressions may contain max-subexpressions which originate from conditional statements in the program, such as if-then-else's, switch statements, etc. Whereas in some cases it is evident which of the alternatives corresponds to the worst case, in many cases it is not and therefore we allow the use of subexpressions of the form $\mathsf{max}(\{e_1, \ldots, e_n\})$ with $n \geq 1$ in cost expressions.

However, the availability of a *CF* comparator sometimes allows automatically removing cost expressions e_i from S when $\exists e_j \in S, i \neq j$ s.t. $e_i \leq_{\varphi} e_j$, where φ is the context constraint applicable to the max-subexpression. If the set S is reduced to a singleton, the max-wrapper can also be simplified away.

Similarly, min-subexpressions can also be simplified in the same way, but reordering the operands to the $\leq_{\varphi}$ relation. I.e., an expression e_i can be removed when we find another expression e_j s.t. $e_j \leq_{\varphi} e_i$.

6. Other Approaches and Related Work

In this section, we discuss other possible approaches to handle the problem of comparing cost functions. In [17], 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:

$$\begin{array}{l|l} w \cdot 2^x \geq 2^y & \{x \geq 0, x \geq y, w \geq 1\} \\ w \cdot Z_{2^x} \geq Z_{2^y} & \{x \geq 0, x \geq y, , w \geq 1, Z_{2^x} \geq Z_{2^y}\} \\ Z_{w \cdot 2^x} \geq Z_{2^y} & \{x \geq 0, x \geq y, , w \geq 1, Z_{2^x} \geq Z_{2^y}, Z_{w \cdot 2^x} \geq Z_{2^x}\} \end{array}$$

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 $e_1 \geq_\varphi e_2$ is to encode $e_1 <_\varphi e_2$ as a Boolean formula that simulates the behavior of the underlying machine architecture. Unsatisfiability of the Boolean formula can be checked using SAT solvers and implies that $e_1 \geq_\varphi e_2$. The drawback of this approach is that it requires fixing a maximum number of bits for representing the value of each variable 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.

Comparing cost functions amounts to checking the validity of $e_1 - e_2 \geq_\varphi 0$, which is equivalent to checking the validity of the first order formula $\varphi \rightarrow e_1 - e_2 \geq 0$. Computer algebra systems (e.g., Reduce [27]) can be used to check the validity of such formulas. When e_1 and e_2 are max expressions over polynomials, these techniques are even complete (assuming the variables are real-valued), however, they are computationally expensive – doubly exponential in the number of variables.

There are other numerical approaches to check the validity of $e_1 - e_2 \geq_\varphi 0$. The first one is to find the roots of $e_1 - e_2$, and check whether those roots satisfy the constraints φ. 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 of the region defined by φ, then we can ensure that the equation holds for all possible values satisfying φ. However, the problem of finding the roots with multiple

variables is hard in general and often not solvable. Computer algebra systems (e.g., Mathematica) can be used for finding the roots. 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. This can be done by finding the minimum values of expression $e_1 - e_2$, a problem that is more affordable using numerical methods [22]. If the minimum values in the region defined by φ 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 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 originating from realistic programs and their performance. We have not explored these issues yet and they remain as subject of future work.

Testing positivity of polynomials is a closely related problem. Practical aspects of this problem have received a considerable attention in the community of term-rewriting systems, mainly to infer polynomial interpretations, which are polynomials that satisfy some properties that can be reduced to positivity (see [20] and references thereof). Intuitively, the existence of such polynomial interpretations imply termination of a corresponding term-rewriting system. Any of these techniques can be used to compare polynomial cost functions, however, it can be used only on max- and nat-free expressions (since they are mostly developed for variables with natural values), and without context constraints. These techniques are powerful, and can be even complete for polynomials with small degrees [25], which are typically enough for proving termination. However, cost function often include polynomials of higher degrees since they describe the complexity of the overall program. Unlike our approach, these approaches are restricted to polynomials, however, there are some techniques [19] that allow using expressions of the form $\mathsf{max}\{0, P\}$ where P is a polynomial, they basically reduce to the previous case using approximations. We believe that the advantage of our approach, over these approaches, is in its modular nature. In particular, if we add more sophisticated expressions to our cost functions, then all we need is to supply a set of comparison rules for corresponding basic expressions.

After our work, [23] has presented a framework for interval-based resource usage verification developed in the context of CIAO-Prolog. The main novelty of the approach is the use of intervals for restricting the applicability of assertions both when the user expresses a resource policy and as the result of the comparison of user-provided assertions with the analysis results. The former is in fact a particular case of our context constraints which not only allow expressing intervals of values of variables but also allow expressing relations among values of variables, such as $x \geq y$, which is not expressible using intervals. The latter is an interesting proposal since it allow indicating what are the input values for which the comparison (resource policy) does not hold. However, the work in [23] does not propose any new method for comparing function. It simply resorts to the use of existing numerical methods for finding roots of polynomial

functions, as discussed above. Once the roots are available, it is determined in each interval between roots whether the comparison holds or not. This has the important problem that existing methods are only applicable to very simple functions and cannot be used for cost functions for realistic programs. In fact, all upper bounds presented in [23] are very simple and they do not contain any max- nor nat-operators.

On a completely different issue, it is worth mentioning that *CFs* representing average-case cost would be extremely useful for the applications of resource analysis. Unfortunately, automatic computation of average cost is rather hard and the current state of the art does not include automatic average cost analysis for realistic programs. It could also be the case that the syntactic form of *CFs* for average-case involves constructions which are not handled by the proposed comparator. Thus, we do not claim that our proposal is directly applicable to *CFs* representing average case. Though many of the ideas will be applicable, some extensions will probably be needed. It remains as future work to extend our comparator to handle average case. But this can only be done once automatic analyzers for average case of real-life programs become available. As a side comment, we do not want to underestimate the difficulty of handling average case *CFs*. A clear underestimation in the area of resource analysis took place when it was expected that finding closed form solutions of cost relations was a relatively simple problem. Once automatic cost analyzers for realistic programs became available it became evident that the problem was much harder than initially expected and that existing techniques for solving recurrence relations were not enough. The problem was not satisfactorily solved until dedicated frameworks [4] were developed.

7. Conclusions

In conclusion, we have proposed a novel approach to comparing cost functions which is able to handle cost functions obtained from the analysis of realistic programs. We propose a mechanism for handling nat-, max-, and min-operators which allows reducing the process of comparing cost functions containing them to the comparison a a series of (simpler) cost expressions, without any loss of precision. This mechanism can then be combined with any method for comparing functions without such operators, including the use of numerical methods. However, in our proposal we also present a novel approach for comparing flat cost functions which is not based on numerical methods and that is of greater applicability than such numerical methods, since the latter are generally restricted to polynomial functions. A possibility which could be explored and remains as future work is the use of numerical methods in cases where our technique is not precise enough to prove that a function is smaller than another one and numerical methods are applicable.

From the practical aspect, our prototype implementation performs well, less than 1 second, when comparing relatively small cost functions. However, it does not provide an answer in a reasonable time when applied to large expressions. This is because of the number of combinations of cost expression to consider

might be exponentially large. Note, however, that the implementation is just a prototype, and there is much room for performance improvements, for example one can discard many of those combinations by only considering the corresponding complexity classes. It is worth noting as well that when checking for $e_1 \geq_\varphi b$, the cost function e_1 is typically inferred by a cost analyser, while b is provided by the user (or vice versa for lower bounds). In such case, one can indeed expect b to be syntactically compact since it is provides by a user, while e_1 might be quite large because the analyser often fails to simplify it. This means that the number of combination of cost expressions to consider would be also small in this case.

Cost functions can represent both upper and lower bounds and our approach works equally well in both cases. Making cost functions comparisons automatically and efficiently is essential for any application of automatic cost analysis. The comparator is currently being integrated in the COSTA [6] and COSTABS [2] systems, where the applications of cost analysis described along the paper are put into practice. Besides, a standalone implementation has been made available online so that other resource analyzers can use it. Finally, 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.

[1] Elvira Albert, Diego Esteban Alonso-Blas, Puri Arenas, Samir Genaim, and Germán Puebla. Asymptotic Resource Usage Bounds. In Zhenjiang Hu, editor, *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*, volume 5904 of *Lecture Notes in Computer Science*, pages 294–310. Springer, 2009.

[2] Elvira Albert, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa, and Germán Puebla. COSTABS: A Cost and Termination Analyzer for ABS. In Oleg Kiselyov and Simon Thompson, editors, *Proceedings of the 2012 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2012, Philadelphia, Pennsylvania, USA, January 23-24, 2012*, pages 151–154. ACM Press, 2012.

[3] Elvira Albert, Puri Arenas, Samir Genaim, Israel Herraiz, and Germán Puebla. Comparing Cost Functions in Resource Analysis. In Marko C. J. D. van Eekelen and Olha Shkaravska, editors, *Foundational and Practical Aspects of Resource Analysis - First International Workshop, FOPARA 2009, Eindhoven, The Netherlands, November 6, 2009, Revised Selected Papers*, volume 6324 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2009.

[4] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011.

[5] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Cost Analysis of Java Bytecode. In Rocco De Nicola, editor,

Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer-Verlag, March 2007.

[6] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24-26, 2007, Revised Lectures*, volume 5382 of *Lecture Notes in Computer Science*, pages 113–132. Springer, 2008.

[7] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Resource Usage Analysis and its Application to Resource Certification. In Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *Lecture Notes in Computer Science*, pages 258–288. Springer, 2009.

[8] Elvira Albert, Richard Bubel, Samir Genaim, Reiner Hähnle, Germán Puebla, and Guillermo Román-Díez. Verified Resource Guarantees using COSTA and KeY. In Siau-Cheng Khoo and Jeremy G. Siek, editors, *Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2011, Austin, TX, USA, January 24-25, 2011*, SIGPLAN, pages 73–76. ACM, 2011.

[9] Elvira Albert, Samir Genaim, and Abu Naser Masud. On the Inference of Resource Usage Upper and Lower Bounds. *ACM Transactions on Computational Logic*, 14(3):22:1–22:35, 2013.

[10] Diego Esteban Alonso-Blas, Puri Arenas, and Samir Genaim. Precise Cost Analysis via Local Reasoning. In Dang Van Hung and Mizuhito Ogawa, editors, *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*, volume 8172 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2013.

[11] R. Bagnara and F. Mesnard. Eventual Linear Ranking Functions. *CoRR*, abs/1306.1901, 2013.

[12] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.

[13] Stephen-John Craig and Michael Leuschel. Self-Tuning Resource Aware Specialisation for Prolog. In Pedro Barahona and Amy P. Felty, editors, *Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 11-13 2005, Lisbon, Portugal*, pages 23–34. ACM, 2005.

[14] Karl Crary and Stephanie Weirich. Resource Bound Certification. In Mark N. Wegman and Thomas W. Reps, editors, *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000*, pages 184–198. ACM, 2000.

[15] Saumya K. Debray, Pedro López-García, Manuel Hermenegildo, and Nai-Wei Lin. Lower Bound Cost Estimation for Logic Programs. Technical Report TR CLIP20/95.0, T.U. of Madrid (UPM), Facultad Informática UPM, 28660-Boadilla del Monte, Madrid-Spain, December 1995.

[16] Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl. Maximal Termination. In Andrei Voronkov, editor, *Rewriting Techniques and Applications, 19th International Conference, RTA 2008, Hagenberg, Austria, July 15-17, 2008, Proceedings*, volume 5117 of *Lecture Notes in Computer Science*, pages 110–125. Springer, 2008.

[17] Bhargav S. Gulavani and Sumit Gulwani. A Numerical Abstract Domain Based on Expression Abstraction and Max Operator with Application in Timing Analysis. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2008.

[18] Manuel V. Hermenegildo, Elvira Albert, Pedro López-García, and Germán Puebla. Abstraction Carrying Code and Resource-Awareness. In *Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 11-13 2005, Lisbon, Portugal*, pages 1–11. ACM, July 2005.

[19] Nao Hirokawa and Aart Middeldorp. Tyrolean Termination Tool: Techniques and Features. *Information and Computation*, 205(4):474–511, 2007.

[20] Hoon Hong and Dalibor Jakus. Testing Positiveness of Polynomials. *Journal of Automed Reasoning*, 21(1):23–38, 1998.

[21] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall international series in computer science. Prentice Hall, New York, 1993.

[22] Scott Kirkpatrick, D. Gelatt Jr., and Mario P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.

[23] Pedro López-García, Luthfi Darmawan, Francisco Bueno, and Manuel V. Hermenegildo. Interval-based Resource Usage Verification: Formalization and Prototype. In Ricardo Peña, Marko C. J. D. van Eekelen, and Olha Shkaravska, editors, *Foundational and Practical Aspects of Resource Analysis - Second International Workshop, FOPARA 2011, Madrid, Spain, May 19, 2011, Revised Selected Papers*, volume 7177 of *Lecture Notes in Computer Science*, pages 54–71. Springer, 2012.

[24] G. Necula. Proof-Carrying Code. In Peter Lee, Fritz Henglein, and Neil D. Jones, editors, *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997)*, pages 106–119. ACM Press, 1997.

[25] Friedrich Neurauter, Aart Middeldorp, and Harald Zankl. Monotonicity Criteria for Polynomial Interpretations over the Naturals. In Jürgen Giesl and Reiner Hähnle, editors, *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, volume 6173 of *Lecture Notes in Computer Science*, pages 502–517. Springer, 2010.

[26] Claudio Ochoa and Germán Puebla. Oracle-Based Poly-Controlled Partial Evaluation. *Electronic Notes in Theoretical Computer Science*, 220(3):145–161, 2008.

[27] REDUCE Computer Algebra System. `http://reduce-algebra.sourceforge.net`.

[28] B. Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9):528–539, 1975.