# Asymptotic Resource Usage Bounds

E. Albert[1], D. Alonso[1], P. Arenas[1], S. Genaim[1], and G. Puebla[2]

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

**Abstract.** When describing the resource usage of a program, it is usual to talk in *asymptotic* terms, such as the well-known "big O" notation, whereby we focus on the behaviour of the program for large input data and make a rough approximation by considering as equivalent programs whose resource usage grows at the same rate. Motivated by the existence of *non-asymptotic* resource usage analyzers, in this paper, we develop a novel transformation from a non-asymptotic cost function (which can be produced by multiple resource analyzers) into its asymptotic form. Our transformation aims at producing tight asymptotic forms which do not contain *redundant* subexpressions (i.e., expressions asymptotically subsumed by others). Interestingly, we integrate our transformation at the heart of a cost analyzer to generate asymptotic *upper bounds* without having to first compute their non-asymptotic counterparts. Our experimental results show that, while non-asymptotic cost functions become very complex, their asymptotic forms are much more compact and manageable. This is essential to improve scalability and to enable the application of cost analysis in resource-aware verification/certification.

## 1 Introduction

A fundamental characteristics of a program is the amount of resources that its execution will require, i.e., its *resource usage*. Typical examples of resources include execution time, memory watermark, amount of data transmitted over the net, etc. *Resource usage analysis* [15,14,8,2,9] aims at automatically estimating the resource usage of programs. Static resource analyzers often produce *cost bound functions*, which have as input the size of the input arguments and return bounds on the resource usage (or *cost*) of running the program on such input.

A well-known mechanism for keeping the size of cost functions manageable and, thus, facilitate human manipulation and comparison of cost functions is *asymptotic analysis*, whereby we focus on the behaviour of functions for large input data and make a rough approximation by considering as equivalent functions which grow at the same rate w.r.t. the size of the input date. The asymptotic point of view is basic in computer science, where the question is typically how to describe the resource implication of scaling-up the size of a computational problem, beyond the "toy" level. For instance, the big O notation is used to define *asymptotic upper bounds*, i.e, given two functions $f$ and $g$ which map natural numbers to real numbers, one writes $f \in O(g)$ to express the fact that there is a natural constant $m \geq 1$ and a real constant $c > 0$ s.t. for any $n \geq m$ we have that $f(n) \leq c * g(n)$. Other types of (asymptotic) computational complexity estimates are lower bounds ("Big Omega" notation) and asymptotically tight estimates, when the asymptotic upper and lower bounds coincide (written using "Big Theta"). The aim of *asymptotic resource usage analysis* is to obtain a cost function $f_a$ which is *syntactically simple* s.t. $f_n \in O(f_a)$ (correctness) and ideally also that $f_a \in \Theta(f_n)$ (accuracy), where $f_n$ is the non-asymptotic cost function.

The scopes of non-asymptotic and asymptotic analysis are complementary. Non-asymptotic bounds are required for the estimation of precise execution time (like in WCET) or to predict accurate memory requirements [4]. The motivations for inferring asymptotic bounds are twofold: (1) They are essential during program development, when the programmer tries to reason about the efficiency of a program, especially when comparing alternative implementations for a given functionality. (2) Non-asymptotic bounds can become unmanageably large expressions, imposing huge memory requirements. We will show that asymptotic bounds are syntactically much simpler, can be produced at a smaller cost, and, interestingly, in cases where their non-asymptotic forms cannot be computed.

The main techniques presented in this paper are applicable to obtain asymptotic versions of the cost functions produced by any cost analysis, including lower, upper and average cost analyses. Besides, we will also study how to perform a tighter integration with an upper bound solver which follows the classical approach to static cost analysis by Wegbreit [15]. In this approach, the analysis is parametric w.r.t. a *cost model*, which is just a description of the resources whose usage we should measure, e.g., time, memory, calls to a specific function, etc. and analysis consists of two phases. (1) First, given a program and a cost model, the analysis produces *cost relations* (CRs for short), i.e., a system of recursive equations which capture the resource usage of the program for the given cost model in terms of the sizes of its input data. (2) In a second step, *closed-form*, i.e., non-recursive, upper bounds are inferred for the CRs. How the first phase is performed is heavily determined by the programming language under study and nowadays there exist analyses for a relatively wide range of languages (see, e.g., [2,8,14] and their references). Importantly, such first phase remains the same for both asymptotic and non-asymptotic analyses and thus we will not describe it. The second phase is language-independent, i.e., once the CRs are produced, the same techniques can be used to transform them to closed-form upper bounds, regardless of the programming language used in the first phase. The important point is that this second phase can be modified in order to produce asymptotic upper bounds directly. Our main contributions can be summarized as follows:

1. We adapt the notion of *asymptotic complexity* to cover the analysis of realistic programs whose limiting behaviour is determined by the limiting behaviour of its loops.
2. We present a novel transformation from *non-asymptotic cost functions* into asymptotic form. After some syntactic simplifications, our transformation

detects and eliminates subterms which are *asymptotically subsumed* by others while preserving the complexity order.

3. In order to achieve motivation (2), we need to integrate the above transformation within the process of obtaining the cost functions. We present a tight integration into (the second phase of) a resource usage analyzer to generate directly asymptotic upper bounds without having to first compute their non-asymptotic counterparts.
4. We report on a prototype implementation within the COSTA system [3] which shows that we are able to achieve motivations (1) and (2) in practice.

## 2 Background: Non-Asymptotic Upper Bounds

In this section, we recall some preliminary definitions and briefly describe the method of [1] for converting *cost relations* (CRs) into upper bounds in *closed-form*, i.e., without recurrences.

### 2.1 Cost Relations

Let us introduce some notation. The sets of natural, integer, real, non-zero natural and non-negative real values are denoted respectively by $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, $\mathbb{N}^+$ and $\mathbb{R}^+$. We write $x$, $y$, and $z$, to denote variables which range over $\mathbb{Z}$. A *linear expression* has the form $v_0 + v_1x_1 + \ldots + v_nx_n$, where $v_i \in \mathbb{Z}$, $0 \leq i \leq n$. Similarly, a *linear constraint* (over $\mathbb{Z}$) has the form $l_1 \leq l_2$, where $l_1$ and $l_2$ are linear expressions. For simplicity we write $l_1 = l_2$ instead of $l_1 \leq l_2 \wedge l_2 \leq l_1$, and $l_1 < l_2$ instead of $l_1 + 1 \leq l_2$. The notation $\bar{t}$ stands for a sequence of entities $t_1, \ldots, t_n$, for some $n{>}0$. We write $\varphi$, $\phi$ or $\psi$, to denote sets of linear constraints which should be interpreted as the conjunction of each element in the set and $\varphi_1 \models \varphi_2$ to indicate that the linear constraint $\varphi_1$ implies the linear constraint $\varphi_2$. Now, the basic building blocks of cost relations are the so-called *cost expressions* $e$ which can be generated using this grammar:

$$e ::= r \mid \mathsf{nat}(l) \mid e + e \mid e * e \mid e^r \mid \log(\mathsf{nat}(l)) \mid n^{\mathsf{nat}(l)} \mid \max(S)$$

where $r \in \mathbb{R}^+$, $n \in \mathbb{N}^+$, $l$ is a linear expression, $S$ is a non empty set of cost expressions, $\mathsf{nat} : \mathbb{Z} \rightarrow \mathbb{N}$ is defined as $\mathsf{nat}(v){=}\max(\{v, 0\})$, and the base of the log is 2 (since any other base can be rewritten to 2). Observe that linear expressions are always wrapped by $\mathsf{nat}$ as we explain below.

*Example 1.* Consider the simple Java method m shown in Fig. 1, which invokes the auxiliary method g, where x is a linked list of boolean values implemented in the standard way. For this method, the COSTA analyzer outputs the cost expression $C_m^+{=}6{+}\mathsf{nat}(n{-}i){*}\max(\{21{+}5{*}\mathsf{nat}(n{-}1), 19{+}5{*}\mathsf{nat}(n{-}i)\})$ as an upper bound on the number of *bytecode* instructions that m executes. Each Java instruction is compiled to possibly several bytecode instructions, but this is not relevant to this work. We are assuming that an upper bound on the number of executed instructions in g is $C_g^+(a, b){=}4{+}5{*}\mathsf{nat}(b{-}a)$. Observe that the use of

```
static void m(List x, int i, int n){
  while (i<n){
    if (x.data) {g(i,n); i++;}
    else {g(0,i); n=n-1;}
    x=x.next;
  }}
```

(1) $\langle C_m(i,n) = 3$
$\quad , \varphi_1 = \{i \geq n\}\rangle$
(2) $\langle C_m(i,n) = 15 + C_g(i,n) + C_m(i',n)$
$\quad , \varphi_2 = \{i < n, i' = i+1\}\rangle$
(3) $\langle C_m(i,n) = 17 + C_g(0,i) + C_m(i,n')$
$\quad , \varphi_3 = \{i < n, , n' = n-1\}\rangle$

**Fig. 1.** Java method and CR.

nat is required in order to avoid incorrectly evaluating upper bounds to negative values. When $i \geq n$, the cost associated to the recursive cases has to be nulled out, this effect is achieved with $\mathsf{nat}(n-i)$ since it will evaluate to 0. □

W.l.o.g., we formalize our mechanism by assuming that all recursions are *direct* (i.e., all cycles are of length one). Direct recursion can be automatically achieved by applying *Partial Evaluation* [11] (see [1] for the technical details).

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

*Example 2.* The *cost relation* (CR for short) associated to method m is shown in Fig. 1 (right). The relations $C_m$ and $C_g$ capture, respectively, the costs of the methods m and g. Intuitively, in CRs, variables represent the sizes of the corresponding data structures in the program and in the case of integer variables they represent their integer value. Eq. 1 is a base case and captures the case where the loop body is not executed. It can be observed that we have two recursive equations (Eq. 2 and Eq. 3) which capture the respective costs of the then and else branches within the while loop. As the list x has been abstracted to its length, the values of x.data are not visible in the CR and the two equations have the same (incomplete) guard, which results in a non-deterministic CR. Also, variables which do not affect the cost (e.g., x) do not appear in the CR. How to automatically obtain a CR from a program is the subject of the first phase of cost analysis as described in Sec. 1. More details can be found in [2,8,14,15]. □

## 2.2 Non-Asymptotic Upper-Bounds

We now describe the approach of [1] to infer the upper bound of Ex. 1 from the equations in Ex. 2. It starts by computing upper bounds for CRs which do not depend on any other CRs, referred to as *standalone cost relations*, and continues by replacing the computed upper bounds on the equations which call such relations. For instance, after computing the upper bound for g shown in Ex. 1, the cost relation in Ex. 2 becomes standalone:

(1) $\langle C_m(i,n) = 3 \quad , \ \varphi_1 = \{i \geq n\}\rangle$
(2) $\langle C_m(i,n) = 15 + \mathsf{nat}(n-i) + C_m(i',n) \quad , \ \varphi_2 = \{i < n, i' = i+1\}\rangle$

(3) $\langle C_m(i,n) = 17 + \mathsf{nat}(i) + C_m(i,n') \ \ , \ \varphi_3 = \{i < n, n' = n-1\}\rangle$

Given a standalone CR made up of $nb$ base cases of the form $\langle C(\bar{x}){=}base_j, \varphi_j\rangle$, $1{\leq}j{\leq}nb$ and $nr$ recursive equations of the form, $\langle C(\bar{x}){=}rec_j{+}\sum_{i=1}^{k_j} C(\bar{y}_i), \varphi_j\rangle$, $1{\leq}j{\leq}nr$, an upper bound can be computed as:

$$(*) \quad C(\bar{x})^+ = \mathsf{I_b} * worst(\{base_1, \ldots, base_{nb}\}) + \mathsf{I_r} * worst(\{rec_1, \ldots, rec_{nr}\})$$

where $\mathsf{I_b}$ and $\mathsf{I_r}$ are, respectively, upper bounds of the number of visits to the base cases and recursive equations and $worst(\{Set\})$ denotes the worst-case (the maximum) value that the expressions in $Set$ can take. Below, we describe the method in [1] to approximate the above upper bound.

**Bounds on the Number of Application of Equations.** The first dimension of the problem is to bound the maximum number of times an equation can be applied. This can be done by examining the structure of the CR (i.e., the number of explicit recursive calls in the equations), together with how the values of the arguments change when calling recursively (i.e., the linear constraints).

We first explain the problem for equations that have at most one recursive call in their bodies. In the above CR, when calling $C_m$ recursively in (2), the first argument $i$ of $C_m$ increases by 1 and in (3) the second argument $n$ decreases by 1. Now suppose that we define a function $f(a,b) = b - a$. Then, we can observe that $\varphi_2 \models f(i,n) > f(i',n) \wedge f(i,n) \geq 0$ and $\varphi_3 \models f(i,n) > f(i,n') \wedge f(i,n) \geq 0$, i.e, for both equations we can guarantee that they will not be applied more than $\mathsf{nat}(f(i_0,n_0)) = \mathsf{nat}(n_0 - i_0)$ times, where $i_0$ and $n_0$ are the initial values for the two variables. Functions such as $f$ are usually called *ranking functions* [13]. Given a cost relation $C(\bar{x})$, we denote by $f_C(\bar{x})$ a ranking function for all loops in $C$. Now, consider that we add an equation that contains two recursive calls:

$$(4) \ \langle C_m(i,n) = C_m(i,n') + C_m(i,n') \ \ , \ \varphi_4 = \{i < n, n' = n-1\}\rangle$$

then the recursive equations would be applied in the worst-case $\mathsf{I_r} = 2^{\mathsf{nat}(n-i)} - 1$ times, which in this paper, we simplify to $\mathsf{I_r} = 2^{\mathsf{nat}(n-i)}$ to avoid having negative constants that do not add any technical problem to asymptotic analysis. This is because each call generates 2 recursive calls, and in each call the argument $n$ decreases at least by 1. In addition, unlike the above examples, the base-case equation would be applied in the worst-case an exponential number of times. In general, a CR may include several base-case and recursive equations whose guards, as shown in the example, are not necessarily mutually exclusive, which means that at each evaluation step there are several equations that can be applied. Thus, the worst-case of applications is determined by the fourth equation, which has two recursive calls, while the worst cost of each application will be determined by the first equation, which contributes the largest direct cost. In summary, the bounds on the number of application of equations are computed as follows:

$$\mathsf{I_r} = \begin{cases} \mathsf{nr}^{\mathsf{nat}(f_C(\bar{x}))} & \text{if } \mathsf{nr} > 1 \\ \mathsf{nat}(f_C(\bar{x})) & \text{otherwise} \end{cases} \qquad \mathsf{I_b} = \begin{cases} \mathsf{nr}^{\mathsf{nat}(f_C(\bar{x}))} & \text{if } \mathsf{nr} > 1 \\ 1 & \text{otherwise} \end{cases}$$

where $\mathsf{nr}$ is the maximum number of recursive calls which appear in a single equation. A fundamental point to note is that the (linear) combination of variables which approximates the number of iterations of loops is wrapped by $\mathsf{nat}$.

This will influence our definition of asymptotic complexity. In logarithmic cases, we can further refine the ranking function and obtain a tighter upper bound. If each recursive equation satisfies $\varphi_j \models f_C(\bar{x}) \geq k * f_C(\bar{y}_i)$, $1 \leq i \leq nr$, where $k > 1$ is a constant, then we can infer that $\mathsf{I_r}$ is bounded by $\lceil \log_k(\mathsf{nat}(f_C(\bar{x}))+1) \rceil$, as each time the value of the ranking function decreases by $k$. For instance, if we replace $\varphi_2$ by $\varphi'_2 = \{i<n, i'=i*2\}$ and $\varphi_3$ by $\varphi'_3 = \{i<n, n'=n/2\}$ (and remove equation 4) then the method of [1] would infer that $\mathsf{I_r}$ is bound by $\lceil \log_k(\mathsf{nat}(n-i)+1) \rceil$.

**Bounds on the Worst Cost of Equations.** As it can be observed in the above example, in each application the corresponding equation might contribute a non-constant number of cost units. Therefore, it is not trivial to compute the worst-case (the maximum) value of all of them. In order to infer the maximum value of such expressions automatically, [1] proposes to first infer *invariants* (linear relations) between the equation's variables and the initial values. For example, the cost relation $C_m(i,n)$ admits as invariant for the recursive equations the formula $\mathcal{I}$ defined as $\mathcal{I}((i_0,n_0),(i,n)) \equiv i \geq i_0 \wedge n \leq n_0 \wedge i < n$, which captures that the values of $i$ (resp. $n$) are greater (resp. smaller) or equal than the initial value and that $i$ is smaller than $n$ at all iterations. Once we have the invariant, we can *maximize* the expressions w.r.t. these values and take the maximal:

$$\mathit{worst}(\{\mathit{rec}_1,\ldots,\mathit{rec}_{nr}\}) = \max(\mathit{maximize}(\mathcal{I},\{\mathit{rec}_1,\ldots,\mathit{rec}_{nr}\}))$$

The operator *maximize* receives an invariant $\mathcal{I}$ and a set of expressions to be maximized and computes the maximal value of each expression independently and returns the corresponding set of maximized expressions in terms of the initial values (see [1] for the technical details). For instance, in the original CR (without Eq. (4)), we compute $\mathit{worst}(\{\mathit{rec}_1,\mathit{rec}_2\}) = \max(\mathit{maximize}(\mathcal{I},\{\mathsf{nat}(n-i),\mathsf{nat}(i)\}))$ which results in $\mathit{worst}(\{\mathit{rec}_1,\mathit{rec}_2\}) = \max(\{\mathsf{nat}(n_0 - i_0),\mathsf{nat}(n_0 - 1)\})$. The same procedure can be applied to the expressions in the base cases. However, it is unnecessary in our example, because the base case is a constant and therefore requires no maximization. Altogether, by applying Equation (*) to the standalone CR above we obtain the upper bounds shown in Ex. 1.

**Inter-Procedural.** In the above examples, all CRs are standalone and do not call any other equations. In the general case, a cost relation can contain $k$ calls to external relations and $n$ recursive calls: $\langle C(\bar{x}) = e + \sum_{i=1}^{k} D_i(\bar{y}_i) + \sum_{j=1}^{n} C(\bar{z}_j), \varphi\rangle$ with $k \geq 0$. After computing the upper bounds $D_i^+(\bar{y}_i)$ for the standalone CRs, we replace the computed upper bounds on the equations which call such relations, i.e., $\langle C(\bar{x}) = e + \sum_{i=1}^{k} D_i^+(\bar{y}_i) + \sum_{j=1}^{n} C(\bar{z}_j), \varphi\rangle$.

## 3 Asymptotic Notation for Cost Expressions

We now present extended versions of the standard definition of the asymptotic notations *big O* and *big Theta*, which handle functions with multiple input arguments, i.e., functions of the form $\mathbb{N}^n \mapsto \mathbb{R}^+$.

**Definition 2 (big O, big Theta).** *Given two functions $f, g : \mathbb{N}^n \mapsto \mathbb{R}^+$, we say that $f \in O(g)$ iff there is a real constant $c > 0$ and a natural constant $m \geq 1$*

*such that, for any $\bar{v} \in \mathbb{N}^n$ such that $v_i \geq m$, it holds that $f(\bar{v}) \leq c * g(\bar{v})$. Similarly, $f \in \Theta(g)$ iff there are real constants $c_1 > 0$ and $c_2 > 0$ and a natural constant $m \geq 1$ such that, for any $\bar{v} \in \mathbb{N}^n$ such that $v_i \geq m$, it holds that $c_1 * g(\bar{v}) \leq f(\bar{v}) \leq c_2 * g(\bar{v})$.*

The big $O$ refers to asymptotic upper bounds and the big $\Theta$ to asymptotically tight estimates, when the asymptotic upper and lower bounds coincide. The asymptotic notations above assume that the value of the function increases with the values of the input such that the function, unless it has a constant asymptotic order, takes the value $\infty$ when the input is $\infty$. This assumption does not necessarily hold when CRs are obtained from realistic programs. For instance, consider the loop in Fig. 1. Clearly, the execution cost of the program increases by increasing the number of iterations of the loop, i.e., $n-i$, the ranking function. Therefore, in order to observe the limiting behavior of the program we should study the case when $\mathsf{nat}(n-i)$ goes to $\infty$, i.e., when, for example, $n$ goes to $\infty$ and $i$ stays constant, but not when both $n$ and $i$ go to $\infty$. In order to capture this asymptotic behaviour, we introduce the notion of nat-free cost expression, where we transform a cost expression into another one by replacing each nat-expression with a variable. This guarantees that we can make a consistent usage of the definition of asymptotic notation since, as intended, after some threshold $m$, larger values of the input variables result in larger values of the function.

**Definition 3 (nat-free cost expressions).** *Given a set of cost expression $E = \{e_1, \ldots, e_n\}$, the nat-free representation of $E$, is the set $\tilde{E} = \{\tilde{e}_1, \ldots, \tilde{e}_n\}$ which is obtained from $E$ in four steps:*

1. *Each nat-expression $\mathsf{nat}(a_1x_1 + \cdots + a_nx_n + c) \in E$ which appears as an exponent is replaced by $\mathsf{nat}(a_1x_1 + \cdots + a_nx_n)$;*
2. *The rest of nat-expressions $\mathsf{nat}(a_1x_1 + \cdots + a_nx_n + c) \in E$ are replaced by $\mathsf{nat}(\frac{a_1}{b}x_1 + \cdots + \frac{a_n}{b}x_n)$, where $b$ is the* greatest common divisor *(gcd) of $|a_1|, \ldots, |a_n|$, and $|\cdot|$ stands for the absolute value;*
3. *We introduce a fresh (upper-case) variable per syntactically different nat-expression.*
4. *We replace each nat-expression by its corresponding variable.*

Cases 1 and 2 above have to be handled separately because if $\mathsf{nat}(a_1x_1 + \cdots + a_nx_n + c)$ is an exponent, we can remove the $c$, but we cannot change the values of any $a_i$. E.g., $2^{\mathsf{nat}(2x+1)} \notin O(2^{\mathsf{nat}(x)})$. This is because $4^x \notin O(2^x)$. Hence, we cannot simplify $2^{\mathsf{nat}(2x)}$ to $2^{\mathsf{nat}(x)}$. In the case that $\mathsf{nat}(a_1x_1 + \cdots + a_nx_n + c)$ does not appear as an exponent, we can remove $c$ and normalize all $a_i$ by dividing them by the *gcd* of their absolute values. This allows reducing the number of variables which are needed for representing the nat-expressions. It is done by using just one variable for all nat expressions whose linear expressions are *parallel* and grow in the same direction. Note that removing the independent term plus dividing all constants by the gcd of their absolute values provides a canonical representation for linear expressions. They satisfy this property iff their canonical representation is the same. This allows transforming both $\mathsf{nat}(2x+3)$ and $\mathsf{nat}(3x+5)$ to $\mathsf{nat}(x)$, and $\mathsf{nat}(2x+4y)$ and $\mathsf{nat}(3x+6y)$ to $\mathsf{nat}(x+2y)$.

*Example 3.* Given the following cost function:

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

Its $\mathsf{nat}$-free representation is:

$$5{+}7*A{*}\max(\{100*A^2{*}B^4, 11*3^B{*}A^2\}){+}2{*}\log(A){*}2^B{*}\log(B){*}C$$

where $A$ corresponds to $\mathsf{nat}(x)$, $B$ to $\mathsf{nat}(y)$ and $C$ to $\mathsf{nat}(x{-}y)$. □

**Definition 4.** *Given two cost expressions* $e_1, e_2$ *and its* $\mathsf{nat}$*-free correspondence* $\tilde{e}_1, \tilde{e}_2$*, we say that* $e_1{\in}O(e_2)$ *(resp.* $e_1{\in}\Theta(e_2)$*) if* $\tilde{e}_1{\in}O(\tilde{e}_2)$ *(resp.* $\tilde{e}_1{\in}\Theta(\tilde{e}_2)$*).*

The above definition lifts Def. 2 to the case of cost expressions. Basically, it states that in order to decide the asymptotic relations between two cost expressions, we should check the asymptotic relation of their corresponding $\mathsf{nat}$-free expressions. Note that by obtaining their $\mathsf{nat}$-free expressions simultaneously we guarantee that the same variables are syntactically used for the same linear expressions.

In some cases, a cost expression might come with a set of constraints which specifies a class of input values for which the given cost expression is a valid bound. We refer to such set as *context constraint*. For example, the cost expression of Ex. 3 might have $\varphi{=}\{x{\geq}y, x{\geq}0, y{\geq}0\}$ as context constraint, which specifies that it is valid only for non-negative values which satisfy $x{\geq}y$. The context constraint can be provided by the user as an input to cost analysis, or collected from the program during the analysis.

The information in the context constraint $\varphi$ associated to the cost expression can sometimes be used to check whether some $\mathsf{nat}$-expressions are guaranteed to be asymptotically larger than others. For example, if the context constraint states that $x \geq y$, then when both $\mathsf{nat}(x)$ and $\mathsf{nat}(y)$ grow to the infinite we have that $\mathsf{nat}(x)$ asymptotically subsumes $\mathsf{nat}(y)$, this information might be useful in order to obtain more precise asymptotic bounds. In what follows, given two $\mathsf{nat}$-expressions (represented by their corresponding $\mathsf{nat}$-variables $A$ and $B$), we say that $\varphi{\models}A \succeq B$ if $A$ asymptotically subsumes $B$ when both go to $\infty$.

## 4 Asymptotic Orders of Cost Expressions

As it is well-known, by using $\Theta$ we can partition the set of all functions defined over the same domain into *asymptotic orders*. Each of these orders has an infinite number of members. Therefore, to accomplish the motivations in Sect. 1 it is required to use one of the elements with simpler syntactic form. Finding a good representative of an asymptotic order becomes a complex problem when we deal with functions made up of non-linear expressions, exponentials, polynomials, and logarithms, possibly involving several variables and associated constraints. For example, given the cost expression of Ex. 3, we want to automatically infer the asymptotic order "$3^{\mathsf{nat}(y)} * \mathsf{nat}(x)^3$".

Apart from simple optimizations which remove constants and normalize expressions by removing parenthesis, it is essential to remove *redundancies*, i.e., subexpressions which are asymptotically subsumed by others, for the final expression to be as small as possible. This requires effectively comparing subexpressions of different lengths and possible containing multiple complexity orders. In this section, we present the basic definitions and a mechanism for transforming non-asymptotic cost expressions into non-redundant expressions while preserving the asymptotic order. Note that this mechanism can be used to transform the output of any cost analyzer into an non-redundant, asymptotically equivalent one. To the best of our knowledge, this is the first attempt to do this process in a fully automatic way. Given a cost expression $e$, the transformations are applied on its $\tilde{e}$ representation, and only afterwards we substitute back the $\mathsf{nat}$-expressions, in order to obtain an asymptotic order of $e$, as defined in Def. 4.

## 4.1 Syntactic Simplifications on Cost Expressions

First, we perform some syntactic simplifications to enable the subsequent steps of the transformation. Given a $\mathsf{nat}$-free cost expression $\tilde{e}$, we describe how to simplify it and obtain another $\mathsf{nat}$-free cost expression $\tilde{e}'$ such that $\tilde{e} \in \Theta(\tilde{e}')$. In what follows, we assume that $\tilde{e}$ is not simply a constant or an arithmetic expression that evaluates to a constant, since otherwise we simply have $\tilde{e} \in O(1)$. The first step is to transform $\tilde{e}$ by removing constants and max expressions, as described in the following definition.

**Definition 5.** *Given a* $\mathsf{nat}$*-free cost expression* $\tilde{e}$*, we denote by* $\tau(\tilde{e})$ *the cost expression that results from* $\tilde{e}$ *by: (1) removing all constants; and (2) replacing each subexpression* $max(\{\tilde{e}_1, \ldots, \tilde{e}_m\})$ *by* $(\tilde{e}_1 + \ldots + \tilde{e}_m)$*.*

*Example 4.* Applying the above transformation on the $\mathsf{nat}$-free cost expression of Ex. 3 results in: $\tau(\tilde{e}){=}A{*}(A^2{*}B^4 + 3^B{*}A^2){+}\log(A){*}2^B{*}\log(B){*}C$. $\square$

**Lemma 1.** $\tilde{e} \in \Theta(\tau(\tilde{e}))$

Once the $\tau$ transformation has been applied, we aim at a further simplification which safely removes sub-expressions which are asymptotically subsumed by other sub-expressions. In order to do so, we first transform a given cost expression into a *normal form* (i.e., a sum of products) as described in the following definition, where we use *basic* $\mathsf{nat}$*-free cost expression* to refer to expressions of the form $2^{r*A}$, $A^r$, or $\log(A)$, where $r$ is a real number. Observe that, w.l.o.g., we assume that exponentials are always in base 2. This is because an expression $n^A$ where $n > 2$ can be rewritten as $2^{\log(n)*A}$.

**Definition 6 (normalized $\mathsf{nat}$-free cost expression).** *A* normalized $\mathsf{nat}$*-free cost expression is of the form* $\Sigma_{i=1}^{n} \Pi_{j=1}^{m_i} b_{ij}$ *such that each* $b_{ij}$ *is a basic* $\mathsf{nat}$*-free cost expression.*

Since $b_1 * b_2$ and $b_2 * b_1$ are equal, it is convenient to view a product as the multi-set of its elements (i.e., basic $\mathsf{nat}$-free cost expressions). We use the letter $M$ to denote such multi-set. Also, since $M_1{+}M_2$ and $M_2{+}M_1$ are equal, it is convenient to view the sum as the multi-set of its elements, i.e., products (represented as multi-sets). Therefore, a normalized cost expression is a multi-set of multi-sets of basic cost expressions. In order to normalize a $\mathsf{nat}$-free cost expression $\tau(\tilde{e})$ we will repeatedly apply the distributive property of multiplication over addition in order to get rid of all parenthesis in the expression.

*Example 5.* The normalized expression for $\tau(\tilde{e})$ of Ex. 4 is $A^3{*}B^4{+}2^{\log(3)*B}{*}A^3{+}\log(A){*}2^B{*}\log(B){*}C$ and its multi-set representation is $\{\{A^3, B^4\}, \{2^{\log(3)*B}, A^3\}, \{\log(A), 2^B, \log(B), C\}\}$ $\square$

## 4.2 Asymptotic Subsumption

Given a normalized nat-free cost expression $\tilde{e} = \{M_1, \ldots, M_n\}$ and a context constraint $\varphi$, we want to remove from $\tilde{e}$ any product $M_i$ which is *asymptotically subsumed* by another product $M_j$, i.e., if $M_j \in \Theta(M_j + M_i)$. Note that this is guaranteed by $M_i \in O(M_j)$. The remaining of this section defines a decision procedure for deciding if $M_i \in O(M_j)$. First, we define several *asymptotic subsumption templates* for which it is easy to verify that a single basic nat-free cost expression $b$ subsumes a complete product. In the following definition, we use the auxiliary functions pow and deg of basic nat-free cost expressions which are defined as: $\mathsf{pow}(2^{r*A}) = r$, $\mathsf{pow}(A^r) = 0$, $\mathsf{pow}(\log(A)) = 0$, $\mathsf{deg}(A^r) = r$, $\mathsf{deg}(2^{r*A}) = \infty$, and $\mathsf{deg}(\log(A)) = 0$. In a first step, we focus on basic nat-free cost expression $b$ with one variable and define when it asymptotically subsumes a set of basic nat-free cost expressions (i.e., a product). The product might involve several variables but they must be subsumed by the variable in $b$.

**Lemma 2 (asymptotic subsumption).** *Let $b$ be a basic* nat*-free cost expression, $M = \{b_1, \cdots, b_m\}$ a product, $\varphi$ a context constraint, $vars(b) = \{A\}$ and $vars(b_i) = \{A_i\}$. We say that $M$ is asymptotically subsumed by $b$, i.e., $\varphi \models M \in O(b)$ if for all $1 \le i \le m$ it holds that $\varphi \models A \succeq A_i$ and one of the following holds:*

1. *if $b = 2^{r*A}$, then*
   (a) *$r > \Sigma_{i=1}^{m}\mathsf{pow}(b_i)$; or*
   (b) *$r \ge \Sigma_{i=1}^{m}\mathsf{pow}(b_i)$ and every $b_i$ is of the form $2^{r_i*A_i}$;*
2. *if $b = A^r$, then*
   (a) *there is no $b_i$ of the form $\log(A_i)$, then $r \ge \Sigma_{i=1}^{m}\mathsf{deg}(b_i)$; or*
   (b) *there is at least one $b_i$ of the form $\log(A_i)$, and $r \ge 1 + \Sigma_{i=1}^{m}\mathsf{deg}(b_i)$*
3. *if $b = \log(A)$, then $m = 1$ and $b_1 = \log(A_1)$*

Let us intuitively explain the lemma. For exponentials, in point 1a, we capture cases such as $3^A = 2^{\log(3)*A}$ asymptotically subsumes $2^A * A^2 * \ldots * \log(A)$ where in "…" we might have any number of polynomial or logarithmic expressions. In 1b, we ensure that $3^A$ does not embed $3^A * A^2 * \log(A)$, i.e., if the power is the same, then we cannot have additional expressions. For polynomials, 2a captures that the largest degree is the upper bound. Note that an exponential would introduce an $\infty$ degree. In 2b, we express that there can be many logarithms and still the maximal polynomial is the upper bound, e.g., $A^2$ subsumes $A * \log(A) * \log(A) * \ldots * \log(A)$. In 3, a logarithm only subsumes another logarithm.

*Example 6.* Let $b = A^3$, $M = \{\log(A), \log(B), C\}$, where $A$, $B$ and $C$ corresponds to $\mathsf{nat}(x)$, $\mathsf{nat}(y)$ and $\mathsf{nat}(x{-}y)$ respectively. Let us assume that the context constraint is $\varphi = \{x \ge y, x \ge 0, y \ge 0\}$. $M$ is asymptotically subsumed by $b$ since $\varphi \models (A \succeq B) \wedge (A \succeq C)$, and condition 2b in Lemma 2 holds. $\square$

The basic idea now is that, when we want to check the subsumption relation on two expression $M_1$ and $M_2$ we look for a partition of $M_2$ such that we can prove the subsumption relation of each element in the partition by a different basic $\mathsf{nat}$-free cost expression in $M_1$. Note that $M_1$ can contain additional basic $\mathsf{nat}$-free cost expressions which are not needed for subsuming $M_2$.

**Lemma 3.** *Let $M_1$ and $M_2$ be two products, and $\varphi$ a context constraint. If there exists a partition of $M_2$ into $k$ sets $P_1, \ldots, P_k$, and $k$ distinct basic $\mathsf{nat}$-free cost expressions $b_1, \ldots, b_k \in M_1$ such that $P_i \in O(b_i)$, then $M_2 \in O(M_1)$.*

*Example 7.* Let $M_1 = \{2^{\log(3)*B}, A^3\}$ and $M_2 = \{\log(A), 2^B, \log(B), C\}$, with the context constraint $\varphi$ as defined in Ex. 6. If we take $b_1 = 2^{\log(3)*A}$, $b_2 = A^3$, and partition $M_2$ into $P_1 = \{2^B\}$, $P_2 = \{\log(A), \log(B), C\}$ then we have that $P_1 \in O(b_1)$ and $P_2 \in O(b_2)$. Therefore, by Lemma 3, $M_2 \in O(M_1)$. Also, for $M_2' = \{A^3, B^4\}$ we can partition it into $P_1' = \{B^4\}$ and $P_2' = \{A^3\}$ such that $P_1' \in O(b_1)$ and $P_2' \in O(b_2)$ and therefore we also have that $M_2' \in O(M_1)$. $\square$

**Definition 7 (asymp).** *Given a cost expression $e$, the overall transformation* `asymp` *takes $e$ and returns the cost expression that results from removing all subsumed products from the normalized expression of $\tau(\tilde{e})$, and then replace each* $\mathsf{nat}$*-variable by the corresponding* $\mathsf{nat}$*-expression.*

*Example 8.* Consider the normalized cost expression of Ex. 5. The first and third products can be removed, since they are subsumed by the second one, as explained in Ex. 7. Then $\mathtt{asymp}(e)$ would be $2^{\log(3)*\mathsf{nat}(y)} * \mathsf{nat}(x)^3 = 3^{\mathsf{nat}(y)} * \mathsf{nat}(x)^3$, and it holds that $e \in \Theta(\mathtt{asymp}(e))$. $\square$

In the following theorem, we ensure that after eliminating the asymptotically subsumed products, we preserve the asymptotic order.

**Theorem 1 (soundness).** *Given a cost expression $e$ and a context constraint $\varphi$, then $\varphi \models e \in \Theta(\mathtt{asymp}(e))$.*

## 4.3 Implementation in COSTA

We have implemented our transformation and it can be used as a back-end of existing non-asymptotic cost analyzers for average, lower and upper bounds (e.g., [9,2,12,5,7]), and regardless of whether it is based on the approach to cost analysis of [15] or any other. We plan to distribute it as free software soon. Currently, it can be tried out through a web interface available from the COSTA web site: `http://costa.ls.fi.upm.es`. COSTA is an abstract interpretation-based COSt and Termination Analyzer for Java bytecode which receives as input a bytecode program and (a choice of) a resource of interest, and tries to obtain an upper bound of the resource consumption of the program.

In our first experiment, we use our implementation to obtain asymptotic forms of the upper bounds on the memory consumption obtained by [4] for the JOlden suite [10]. This benchmark suite was first used by [6] in the context of memory usage verification and is becoming a standard to evaluate memory usage analysis [5,4]. None of the previous approaches computes asymptotic bounds. We are able to obtain accurate asymptotic forms for all benchmarks in the suite and the transformation time is negligible (less than 0.1 milliseconds in all cases). As a simple example, for the benchmark `em3d`, the non-asymptotic upper bound is $8{*}\mathsf{nat}(d{-}1){*}\mathsf{nat}(b){+}8{*}\mathsf{nat}(d){+}8{*}\mathsf{nat}(b)\ {+}56{*}\mathsf{nat}(d{-}1){+}16{*}\mathsf{nat}(c)\ {+}73$ and we transform it to $\mathsf{nat}(d){*}\mathsf{nat}(b){+}\mathsf{nat}(c)$. The remaining examples can be tried online in the above `url`.

## 5 Generation of Asymptotic Upper Bounds

In this section we study how to perform a tighter integration of the asymptotic transformation presented Sec. 4 within resource usage analyses which follow the classical approach to static cost analysis by Wegbreit [15]. To do this, we reformulate the process of inferring upper bounds sketched in Sect. 2.2 to work directly with asymptotic functions at all possible (intermediate) stages. The motivation for doing so is to reduce the huge amount of memory required for constructing non-asymptotic bounds and, in the limit, to be able to infer asymptotic bounds in cases where their non-asymptotic forms cannot be computed.

**Asymptotic CRS.** The first step in this process is to transform cost relations into asymptotic form before proceeding to infer upper bounds for them. As before, we start by considering standalone cost relations. Given an equation of the form $\langle C(\bar{x}){=}e{+}\sum_{i=1}^{k} C(\bar{y}_i), \varphi\rangle$ with $k \geq 0$, its associated *asymptotic* equation is $\langle C_A(\bar{x}){=}\mathtt{asymp}(e){+}\sum_{i=1}^{k} C_A(\bar{y}_i), \varphi\rangle$. Given a cost relation $C$, its asymptotic cost relation $C_A$ is obtained by applying the above transformation to all its equations. Applying the transformation at this level is interesting in order to simplify both the process of computing the worst case cost of the recursive equations and the base cases when computing Eq. $(*)$ as defined in Sect. 2.2.

*Example 9.* Consider the following CR:

$\langle C(a,b) = \underline{\mathsf{nat}(a+1)^2}\ \ ,\ \{a{\geq}0, b{\geq}0\}\rangle$
$\langle C(a,b) = \underline{\mathsf{nat}(a{-}b){+}\log(\mathsf{nat}(a{-}b))}{+}C(a',b')\ \ ,\ \{a{\geq}0, b{\geq}0, a'{=}a{-}2, b'{=}b{+}1\}\rangle$
$\langle C(a,b) = \underline{2^{\mathsf{nat}(a+b)}{+}\mathsf{nat}(a){*}\log(\mathsf{nat}(a))}{+}C(a',b')\ \ ,\ \{a{\geq}0, b{\geq}0, a'{=}a{+}1, b'{=}b{-}1\}\rangle$

By replacing the underlined expressions by their corresponding `asymp` expressions as explained in Theorem 1, we obtain the asymptotic relation:

$\langle C_A(a,b) = \mathsf{nat}(a)^2\ \ ,\ \{a{\geq}0, b{\geq}0\}\rangle$
$\langle C_A(a,b) = \mathsf{nat}(a{-}b){+}C_A(a',b')\ \ ,\ \{a{\geq}0, b{\geq}0, a'{=}a{-}2, b'{=}b{+}1\}\rangle$
$\langle C_A(a,b) = 2^{\mathsf{nat}(a+b)}{+}C_A(a',b')\ \ ,\ \{a{\geq}0, b{\geq}0, a'{=}a{+}1, b'{=}b{-}1\}\rangle$

In addition to reducing their sizes, the process of maximizing the `nat` expressions is more efficient since there are fewer `nat` expressions in the asymptotic CR. □

An important point to note is that, while we can remove all constants from $e$, it is essential that we keep the constants in the size relations $\varphi$ to ensure soundness. This is because they are used to infer the ranking functions and to compute the invariants, and removing such constants might introduce imprecision and more important soundness problems as we explain in the following examples.

*Example 10.* The above relation admits a ranking function $f(a,b)=\mathsf{nat}(2a+3b+1)$ which is used to bound the number of applications of the recursive equations. Clearly, if we remove the constants in the size relations, e.g., transform $a'=a-2$ into $a'=a$, the resulting relation is non-terminating and we cannot find a ranking function. Besides, removing constants from constraints which are not necessarily related to the ranking function also might result in incorrect invariants. For example, changing $n'=n+1$ to $n'=n$ in the following equation:

$$\langle C(m,n) = \mathsf{nat}(n) + C(m',n') \ , \ \{m>0, m'<m, n'=n+1\}\rangle$$

would result in an invariant which states that the value of $n$ is always equal to the initial value $n_0$, which in turn leads to the upper-bound $\mathsf{nat}(m_0)*\mathsf{nat}(n_0)$ which is clearly incorrect. A possible correct upper-bound is $\mathsf{nat}(m_0)*\mathsf{nat}(n_0+m_0)$ which captures that the value of $\mathsf{nat}(n)$ increases up to $\mathsf{nat}(n_0+m_0)$. $\square$

**Asymptotic Upper Bounds.** Once the standalone CR is put into asymptotic form, we proceed to infer an upper bound for it as in the case of non-asymptotic CRs and then we apply the transformation to the result. Let $C_A(\bar{x})$ be an asymptotic cost relation. Let $C_A^+(\bar{x})$ be its upper bound computed as defined in Eq. ($*$). Its asymptotic upper bound is $C^+_{asymp}(\bar{x}) = \mathtt{asymp}(C_A^+(\bar{x}))$. Observe that we are computing $C_A^+(\bar{x})$ in a non-asymptotic fashion, i.e., we do not apply `asymp` to each $\mathsf{l_b}$, $\mathsf{l_r}$, *worst* in ($*$), but only to the result of combining all elements. We could apply `asymp` to the individual elements and then to the result of their combination again. In practice, it almost makes no difference as this operation is really inexpensive.

*Example 11.* Consider the second CR of Ex. 9. The analyzer infers the invariant $\mathcal{I} = \{0\leq a\leq a_0, 0\leq b\leq b_0, a\geq 0, b\geq 0\}$, from which we maximize $\mathsf{nat}(a)^2$ to $\mathsf{nat}(a_0)^2$, $\mathsf{nat}(a-b)$ to $\mathsf{nat}(a_0)$ (since the maximal value occurs when $b$ becomes 0), and $2^{\mathsf{nat}(a+b)}$ to $2^{\mathsf{nat}(a_0+b_0)}$. The number of applications of the recursive equations is $\mathsf{nat}(2a_0+3b_0+1)$ (see Ex. 10). By applying Eq. ($*$), we obtain the upper bound: $C_A^+(a,b) = \mathsf{nat}(2a+3b+1) * \max(\{\mathsf{nat}(a), 2^{\mathsf{nat}(a+b)}\}) + \mathsf{nat}(a)^2$. Applying `asymp` to the above upper bound results in: $C^+_{asymp}(a,b) = 2^{\mathsf{nat}(a+b)} * \mathsf{nat}(2a+3b)$. $\square$

**Inter-procedural.** The practical impact of integrating the asymptotic transformation within the solving method comes when we consider relations with calls to external relations and compose their asymptotic results. This is because, when the number of calls and equations grow, the fact that we manipulate more compact asymptotic expressions is fundamental to enable the scalability of the system. Consider a cost relation with $k$ calls to external relations and $n$ recursive calls: $\langle C(\bar{x})=e+\sum_{i=1}^{k} D_i(\bar{y}_i)+\sum_{j=1}^{n} C(\bar{z}_j), \varphi\rangle$ with $k\geq 0$. Let $D^+_{i_{asymp}}(\bar{y}_i)$ be the asymptotic upper bound for $D_i(\bar{y}_i)$. $C^+_{asymp}(\bar{x})$ is the asymptotic upper bound of the standalone relation $\langle C(\bar{x})=e+\sum_{i=1}^{k} D^+_{i_{asymp}}(\bar{y}_i)+\sum_{j=1}^{n} C(\bar{z}_j), \varphi\rangle$.

**Theorem 2 (soundness).** *$C^+(\bar{x}) \in O(C^+_{asymp}(\bar{x}))$.*

| **Bench.** | $\mathbf{T}_{ub}$ | $\mathbf{T}_{aub}$ | $\mathbf{Size}_{ub}$ | $\mathbf{Size}_{aub}$ | **#Eq** | $\frac{\mathbf{Size}_{ub}}{\mathbf{\#Eq}}$ | $\frac{\mathbf{Size}_{aub}}{\mathbf{\#Eq}}$ | $\frac{\mathbf{Size}_{ub}}{\mathbf{Size}_{aub}}$ |
|---|---|---|---|---|---|---|---|---|
| BST | 0 | 0 | 23 | 4 | 31 | 0.74 | 0.13 | 5.75 |
| Fibonacci | 0 | 0 | 47 | 9 | 39 | 1.21 | 0.23 | 5.22 |
| Hanoi | 0 | 0 | 67 | 14 | 48 | 1.39 | 0.29 | 4.78 |
| MatMult | 0 | 0 | 152 | 38 | 67 | 2.27 | 0.56 | 4.00 |
| Delete | 0 | 4 | 320 | 65 | 100 | 3.20 | 0.65 | 4.92 |
| FactSum | 4 | 4 | 717 | 95 | 117 | 6.12 | 0.81 | 7.54 |
| SelectOrd | 0 | 4 | 1447 | 155 | 136 | 10.63 | 1.14 | 9.33 |
| ListInter | 4 | 16 | 3804 | 257 | 173 | 21.98 | 1.48 | 14.80 |
| EvenDigits | 4 | 20 | 7631 | 400 | 191 | 39.95 | 2.09 | 19.07 |
| Cons | 12 | 32 | 15268 | 585 | 214 | 71.34 | 2.73 | 26.09 |
| Power | 24 | 40 | 24265 | 588 | 223 | 108.81 | 2.63 | 41.26 |
| MergeList | 96 | 60 | 48536 | 828 | 245 | 198.10 | 3.37 | 58.61 |
| ListRev | 140 | 76 | 48545 | 829 | 254 | 191.12 | 3.26 | 58.55 |
| Incr | × | 112 | × | 1126 | 282 | × | 3.99 | × |
| Concat | × | 164 | × | 1538 | 296 | × | 5.19 | × |
| ArrayRev | × | 232 | × | 2127 | 305 | × | 6.97 | × |
| Factorial | × | 284 | × | 2130 | 314 | × | 6.78 | × |
| DivByTwo | × | 328 | × | 2135 | 323 | × | 6.60 | × |
| Polynomial | × | 436 | × | 2971 | 346 | × | 8.58 | × |
| MergeSort | × | 440 | × | 3234 | 385 | × | 8.40 | × |

**Table 1.** Scalability of asymptotic cost expressions

Note that the soundness theorem, unlike Th. 1, guarantees only that the asymptotic expression is $O$ and not $\Theta$. Let us show an example.

*Example 12.* Consider $ub{=}\mathsf{nat}(a{-}b{+}1){*}2^{\mathsf{nat}(c)}{+}5$ and $\mathtt{asymp}(ub){=}\mathsf{nat}(a{-}b){*}2^{\mathsf{nat}(c)}$. Plugging $ub$ in a context where $b{=}a{+}1$ results in 5 (since then $\mathsf{nat}(a{-}b{+}1){=}0$). Plugging $\mathtt{asymp}(ub)$ in the same context results in $2^{\mathsf{nat}(c)}$ which is clearly less precise. □

Intuitively, the source of the loss of precision is that, when we compute the asymptotic upper bound, we are looking at the cost in the limiting behavior only and we might miss a particular point in which such cost becomes zero. In our experience, this does not happen often and it could be easily checked before plugging in the asymptotic result, replacing the upper bound by zero.

## 5.1 Experimental Results on Scalability

In this section, we aim at studying how the size of cost expressions (non-asymptotic vs. asymptotic) increases when larger CRs are used, i.e., the scalability of our approach. To do so, we have used the benchmarks of [1] shown in Table 1. These benchmarks are interesting because they cover the different complexity order classes, as it can be seen, the benchmarks range from constant to exponential complexity, including polynomial and divide and conquer. The source code of such programs is also available at the COSTA web site.

As in [1], in order to assess the scalability of the approach, we have connected together the CRs for the different benchmarks by introducing a call from each CR to the one appearing immediately above it in the table. Such call is always introduced in a recursive equation. Column **#Eq** shows the number of equations in the corresponding benchmarks. Reading this column top-down, we can see that when we analyze BST we have 31 equations. Then, for Fibonacci, the number of equations is 39, i.e., its 8 equations plus the 31 which have been previously accumulated. Progressively, each benchmark adds its own number of equations to the one above. Thus, in the last row we have a CR with all the equations connected, i.e., we compute an upper bound of a CR with at least 20 nested loops and 385 equations.

Columns $\mathbf{T}_{ub}$ and $\mathbf{T}_{aub}$ show, respectively, the times of composing the non-asymptotic and asymptotic bounds, after discarding the time common part for both, i.e., computing the ranking functions and the invariants. It can be observed that the times are negligible from BST to EvenDigits, which are the simplest benchmarks and also have few equations. The interesting point is that when cost expressions start to be considerably large, $\mathbf{T}_{ub}$ grows significantly, while $\mathbf{T}_{aub}$ remains small. This is explained by the sizes of the expressions they handle, as we describe below. For the columns that contain "$\times$", COSTA has not been able to compute a non-asymptotic upper bound because the underlying Prolog process has run out of memory.

Columns $\mathbf{Size}_{ub}$ and $\mathbf{Size}_{aub}$ show, respectively, the sizes of the computed non-asymptotic and asymptotic upper bounds. This is done by regarding the upper bound expression as a tree and counting its number of nodes, i.e., each operator and each operand is counted as one. As for the time, the sizes are quite small for the simplest benchmarks, and they start to increase from SelectOrd. Note that for these examples, the size of the non-asymptotic upper bounds is significantly larger than the asymptotic. Columns $\frac{\mathbf{Size}_{ub}}{\mathbf{\#Eq}}$ and $\frac{\mathbf{Size}_{aub}}{\mathbf{\#Eq}}$ show, resp., the size of the non-asymptotic and asymptotic bounds per equation. The important point is that while this ratio seems to grow exponentially for non-asymptotic upper bounds, $\frac{\mathbf{Size}_{aub}}{\mathbf{\#Eq}}$ grows much more slowly. We believe that this demonstrates that our approach is scalable, even if the implementation is still preliminary.

## 6 Conclusions and Future Work

We have presented a general asymptotic resource usage analysis which can be combined with existing non-asymptotic analyzers by simply adding our transformation as a back-end or, interestingly, integrated into the mechanism for obtaining upper bounds of recurrence relations. This task has been traditionally done manually in the context of complexity analysis. When it comes to apply it to an automatic analyzer for a real-life language, there is a need to develop the techniques to infer asymptotic bounds in a precise and effective way. To the best of our knowledge, our work is the first one which presents a generic and fully automatic approach. In future work, we plan to adapt our general framework to infer asymptotic lower-bounds on the cost and also to integrate our work into a proof-carrying code infrastructure.

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

## References

1. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *15th International Symposium on Static Analysis (SAS'08)*, volume 5079 of *Lecture Notes in Computer Science*, pages 221–237, 2008.
2. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *ESOP*, LNCS 4421, pages 157–172. Springer, 2007.
3. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *FMCO'07*, number 5382 in LNCS, pages 113–133. Springer, 2008.
4. E. Albert, S. Genaim, and M. Gómez-Zamalloa. Live Heap Space Analysis for Languages with Garbage Collection. In *ISMM*. ACM Press, 2009.
5. V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric Prediction of Heap Memory Requirements. In *ISMM*. ACM Press, 2008.
6. W.-N. Chin, H. H. Nguyen, S. Qin, and M. C. Rinard. Memory Usage Verification for OO Programs. In *Proc. of SAS'05*, volume 3672 of *LNCS*, pages 70–86, 2005.
7. W-N. Chin, H.H. Nguyen, C. Popeea, and S. Qin. Analysing Memory Resource Bounds for Low-Level Programs. In *ISMM*. ACM Press, 2008.
8. S. K. Debray and N. W. Lin. Cost analysis of logic programs. *ACM TOPLAS*, 15(5):826–875, November 1993.
9. S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139. ACM, 2009.
10. JOlden Suite Collection. http://www-ali.cs.umass.edu/DaCapo/benchmarks.html.
11. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, 1993.
12. J. Navas, M. Méndez-Lojo, and M. Hermenegildo. User-Definable Resource Usage Bounds Analysis for Java Bytecode. In *BYTECODE*. Elsevier, 2009.
13. A. Podelski and A. Rybalchenko. A Complete Method for the Synthesis of Linear Ranking Functions. In *5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, Lecture Notes in Computer Science, pages 239–251. Springer, 2004.
14. D. Sands. Complexity Analysis for a Lazy Higher-Order Language. In *ESOP'00*, volume 432 of *LNCS*, pages 361–376. Springer, 1990.
15. B. Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9), 1975.