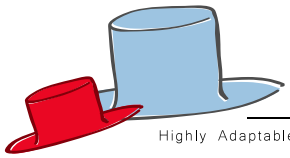


Project N°: **FP7-231620**

Project Acronym: **HATS**

Project Title: **Highly Adaptable and Trustworthy Software using Formal Models**

Instrument: **Integrated Project**

Scheme: **Information & Communication Technologies**

Future and Emerging Technologies

Deliverable D4.2
Report on Resource Guarantees

Due date of deliverable: (T24)

Actual submission date: 1st March 2011

Start date of the project: **1st March 2009** Duration: **48 months**

Organisation name of lead contractor for this deliverable: **UPM**

Final version

Integrated Project supported by the 7th Framework Programme of the EC		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Executive Summary:
Report on Resource Guarantees

This document summarizes deliverable D4.2 of project FP7-231620 (HATS), an Integrated Project supported by the 7th Framework Programme of the EC within the FET (Future and Emerging Technologies) scheme. Full information on this project, including the contents of this deliverable, is available online at `http://www.hats-project.eu`.

The report contains the basic framework for *resource usage analysis* (a.k.a. *cost* analysis) of ABS programs as well as a series of advanced issues which aim at improving the efficiency, the accuracy and the applicability of the proposed framework. Besides, we have studied the *verification* of the resource bounds obtained by the resource usage analyzer by using the KeY system. The report contains conceptually three parts:

1. *Cost analysis of ABS.* This part develops the framework for resource guarantees (or resource consumption) analysis of ABS programs. To the best of our knowledge, it is the first static cost analysis for concurrent object-oriented programs. The analysis is field-sensitive, as proposed in [2, 8]. It includes a general notion of resource that can be instantiated to measure both traditional and concurrency-related types of resources. One of the interesting notions of resource is the *task-level* of an application, which refers to the maximum number of tasks that are spawned along any execution of the program. This cost model has been published in [3]. The overall cost analysis framework is under revision for its publication.

2. *Advanced issues in cost analysis.* Next, we focus on a series of advanced issues in cost analysis which are of general interest in the context of cost analysis of any language, and can be applied in particular to the ABS cost analysis framework above:

   - *Component-based approach.* First, we present the modular extension of the cost analysis framework where the objective is that different components can be analyzed independently and the results then composed together. The compositional framework has been recently presented in [58].
   - *Asymptotic upper bounds.* The results of cost analysis are usually precise, non-asymptotic upper bounds (UBs). For most applications in HATS, their asymptotic versions can be of further interest because they are simpler, allow ignoring implementation details and are more efficient to obtain. We have designed an automatic transformation of non-asymptotic UBs into asymptotic form, published in [5].
   - *Checking against specifications.* The next step is to define a technique, presented in [6], to compare the UBs automatically generated by a cost analyzer against resource specifications provided by the user (or the system vendor). This will allow us to actually verify that the software will safely run on the actual configuration.
   - *Accurate upper and lower bounds.* Clearly, improving the accuracy of the results is crucial for all applications of cost analysis (verification, certification, optimization, etc). We have presented in [14] a novel technique to infer more precise UBs than [9] and which, furthermore, can be dually applied to infer lower bounds.

3. *Verification of resource guarantees using KeY+*COSTA. In spite of being based on theoretically sound techniques, the resource analyzer may contain bugs which render the resource guarantees thus obtained not completely trustworthy. We have investigated, and published in [4], an approach to formally verify the correctness of such resource guarantees using the KeY system.

List of Authors

Elvira Albert (UCM)
Puri Arenas (UCM)
Richard Bubel (CTH)
Samir Genaim (UCM)
Miguel Gómez-Zamalloa (UCM)
Reiner Hähnle (CTH)
Germán Puebla (UPM)

Contents

Chapter 1

Introduction

One of the most important features of a program is its resource consumption. By resource, we mean not only traditional cost measures (e.g., memory consumption, executed instructions) but also concurrency-related measures (e.g., tasks spawned, requests to remote servers). In the present HATS Deliverable D4.2, we develop a novel *cost analysis* framework [63] (a.k.a. *resource usage* analysis) for concurrent ABS programs.

Cost analysis aims at *statically* (i.e., without having to run the program) inferring sound approximations of the resource consumption of executing the program. The classical approach to cost analysis by Wegbreit dates back to 1975 [63]. It consists of two phases. In the first phase, given a program and a *cost model*, the analysis produces *cost recurrence equations* (or simply *cost relations—CRs* for short), i.e., a system of recursive equations which capture the cost of the program in terms of the size of its input data. Cost analyzers are usually parametric on the cost model, e.g., cost models widely used are the number of executed instructions, memory allocated, number of calls to methods, etc. (see, e.g., [12]). In the second phase, once *CRs* are generated, analyzers try to compute *closed-forms* for them, i.e., cost expressions which are not in recursive form.

COSTA is a state-of-the-art cost analyzer initially developed for sequential Java bytecode programs, which already existed before the project HATS started. The main challenge in HATS, which is described in this deliverable in detail, has been to apply the main techniques in COSTA to the context of ABS programs, concurrency being the most difficult aspect. Automatically inferring the resource usage of concurrent programs is challenging because of the inherent complexity of concurrent behaviors. Computations in a concurrent system can be suspended and resumed and shared variables can be modified by different tasks that interact with each other. A cost analyzer must consider all possible paths and interactions in order to infer a sound approximation of the resource consumption.

In addition to traditional applications (e.g., optimization [63], verification and certification of resource consumption [33]), cost analysis opens up interesting applications in the context of concurrent programming. In general, having anticipated knowledge of the resource consumption of the different components which constitute a system is useful for distributing the workload. Upper bounds (UBs) can be used to predict that one component may receive a large amount of remote requests, while other siblings are idle most of the time. An interesting instance of our framework is the task-level analysis (published in [3]) which infers the maximum number of tasks spawned along the program execution. Also, our framework allows instantiating the different components with the particular features of the infrastructure on which they are deployed (memory available, etc.). Then, the analysis can be used to detect the components that consume more resources and may introduce bottlenecks. Lower bounds (LBs) on the resource usage can be used to decide whether it is worth executing locally a task or requesting remote execution.

After introducing some preliminary notions, in Chapter 3, we propose, to the best of our knowledge, the first static cost analysis for concurrent object-oriented (OO) programs. It includes a general notion of resource that can be instantiated to measure both traditional and concurrency-related types of resources. The main novelties of this analysis are:

1. We introduce a sound size analysis for concurrent execution. The analysis is *field-sensitive*, i.e., it

tracks (the sizes/values of) data stored in the heap whenever it is sound to do so. Intuitively, no information about object fields can be assumed when a task is started (entry points of methods) or resumed (after a processor release point). This is because, since ABS makes no assumption on the scheduling, there is no control on which other tasks may have executed previously and whether they have modified the values of fields.

2. We lift the standard definition of cost used in sequential programming to the distributed setting by relying on the notion of *cost center*. Cost centers [57] were originally introduced for profiling functional executions, but their use in static analysis is new. In our setting, each cost center represents a (potentially distributed) component and allows us to separate their costs.

3. We present a novel form of cost recurrence relations (*RRs*), which is parametric w.r.t. the notion of cost center, and which uses the field-sensitive size relations of item 1. From the obtained equations, it is possible to obtain both upper and lower bounds on the resources consumed by the different components of the system. Importantly, these recurrence equations can be solved by using standard solvers developed for cost analysis of sequential programs.

4. We increase the accuracy of the field-sensitive size analysis by means of *class invariants* [52] which contain information on the shared memory. In most cases, the required invariants can be automatically generated.

5. We report on a prototype implementation (which can be used online through a web interface) and evaluate it on concurrent ABS applications developed within HATS.

Chapter 4 presents a number of advanced issues in the field of resource analysis. The techniques proposed are of general interest in cost analysis for any programming language and, indeed, most of them were initially developed (during the first year of HATS) in the context of the COSTA system for cost analysis of Java bytecode. Besides, the same techniques can be applied directly to cost analysis of ABS programs and, in particular, to the framework presented in Chapter 3. Chapter 4 is divided in the following four parts:

- *Component-based approach.* Cost analyzers typically obtain UBs of medium size applications by means of *global* analysis, in the sense that all the code used by such applications has to be analyzed. However, global analysis has important weaknesses, such as its high memory requirements and its lack of efficiency, since often some parts of the code have to be analyzed over and over again, libraries being a paramount example of this. We have developed and implemented an extension of cost analysis in order to make it modular by allowing separate analysis of individual methods. The compositional framework has been presented at FACS'2010 [58].

- *Asymptotic UBs.* A well-known mechanism for keeping the size of cost functions manageable and, thus, facilitate human manipulation and comparison of cost functions is the *asymptotic analysis*. 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. We have designed an automatic transformation of non-asymptotic UBs into asymptotic form, published at APLAS'09 [5]. The scopes of non-asymptotic and asymptotic analysis are complementary. Non-asymptotic bounds are required for the estimation of precise execution time, like in worst cost execution time (WCET) or to predict accurate memory requirements [13]. 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 have shown 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.

- *Checking against specifications.* In all applications of resource analysis, such as resource-usage verification, program synthesis and optimization, etc., it is necessary to compare cost functions. This allows choosing an implementation with smaller cost or to guarantee that the given resource-usage bounds are preserved. Essentially, given a method m, a cost function f_m and a context (set of linear constraints) ϕ_m which impose size restrictions (e.g., that a variable in m is larger than a certain value or that the size of an array is non zero, etc.), we aim at comparing it with another cost function bound $\mathtt{b}$ and corresponding size constraints $\phi_{\mathtt{b}}$. We have developed a new method, presented at FOPARA'09 [6], to compare the UBs automatically generated by a cost analyzer against resource specifications provided by the user (or the system vendor). This will allow us to actually verify that the software will safely run on the actual configuration.

- *Accurate upper and lower bounds.* Needless to say, precision is fundamental for most applications of cost analysis. For instance, the UBs are widely used to estimate the space and time requirements of programs execution and provide resource guarantees [33]. Lack of precision can make the system fail to prove the resource usage requirements imposed by the software client. For example, it makes much difference for the precision we gain if we infer $\frac{1}{2}n^2$ instead of n^2 for a given method. With the latter UB, an execution with n=10 will be rejected if we have only 50 resources available, while with the former one it is accepted. LBs 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 LB resource consumption is sufficiently large. Precision will be essential here to achieve a satisfactory scheduling. We have published at VMCAI'11 [14] a novel technique to infer more precise UBs which, furthermore, can be dually applied to infer LBs.

Finally, Chapter 5 presents how, using KeY, it is possible to formally and automatically verify the correctness of the UBs obtained by cost analysis. There is a growing awareness, both in industry and academia, of the crucial role of formally proving the correctness of systems. Verifying the correctness of modern static analyzers (like COSTA) is rather challenging, among other things, because of the sophisticated algorithms used in them, their evolution over time, and, possibly, proprietary considerations. A simpler alternative is to construct a validating tool [55] which, after every run of the analyzer, formally confirms that the results are correct and, optionally, generates correctness proofs. We will see how it is possible to verify the *resource guarantees* obtained by static analysis using the KeY system. Realizing the cooperation between COSTA and KeY requires a number of non-trivial extensions of both systems, which are described in more detail in the corresponding chapter.

Chapter 2

Preliminaries

In this chapter, we recall some preliminary definitions which are useful to understand the next chapters.

2.1 Linear Constraints

Let us first introduce some notation. The sets of natural, integer, rational, real, non-zero natural, non-negative rational and non-negative real values are denoted respectively by $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{N}^+$, $\mathbb{Q}^+$ 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 $l_1 + 1 \leq l_2$. Note that the constraints with rational coefficients can be always transformed to equivalent constraints with integer coefficients, e.g., $\frac{1}{2}x > y$ is equivalent to $x > 2y$. The notation $\bar{t}$ stands for a sequence of entities $t_1, \ldots, t_n$, for some $n > 0$, and $vars(t)$ to refer to the set of variables that appear syntactically in an entity t. We write φ, ϕ or ψ (possibly subscripted and/or superscripted) to denote sets of linear constraints which should be interpreted as the conjunction of each element in the set. A solution for φ is an assignment $\sigma : vars(\varphi) \mapsto \mathbb{Z}$ for which φ is satisfiable. The set of all solutions (assignments) of φ is denoted by $[\![\varphi]\!]$. We use $\varphi_1 \models \varphi_2$ to indicate that $[\![\varphi_1]\!] \subseteq [\![\varphi_2]\!]$. We use $\sigma(t)$ or $t\sigma$ to bind each $x \in vars(t)$ to $\sigma(x)$, $\exists\bar{x}.\varphi$ for the elimination of the variables $\bar{x}$ from φ, and $\bar{\exists}\bar{x}.\varphi$ for the elimination of all variables but $\bar{x}$ from φ.

2.2 Cost Relations

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

Definition 2.2.1 (cost expressions)**.** *A symbolic expression e is a* cost expression *if it can be generated using the grammar below:*

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

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

is not a valid cost expression, and shbut rather Cost expressions are symbolic expressions which represent the resources we accumulate and are the non-recursive building blocks for defining *CRs* and for the closed-form UBs that we infer for them. Importantly, linear expressions are always wrapped by nat in order to avoid negative cost. E.g., instead of writing $x + y$ we write $\mathsf{nat}(x + y)$, which lifts $x + y$ to 0 when $x + y < 0$. Observe that cost expressions are monotonic in their nat sub-expressions, i.e., replacing $\mathsf{nat}(l) \in e$ by $\mathsf{nat}(l')$ such that $l' \geq l$ results in a cost expression e' such that $e' \geq e$. This property is fundamental for the correctness results in the next sections.

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

Figure 2.1: Java method and *CRS*.

Definition 2.2.2 (cost relation system)**.** *A* cost relation system *(CRS) 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* φ *is a set of linear constraints over* $\bar{x} \cup vars(e) \bigcup_{i=1}^{k} \bar{y}_i$*.*

As notation, we use *CR* to denote a single cost relation, i.e., to the set of cost equations in a *CRS* for a cost relation symbol C. A cost equation $\langle C(\bar{x}) = e + \sum_{i=1}^{k} D_i(\bar{y}_i), \varphi\rangle$ states that the cost of $C(\bar{x})$ is e plus the sum of the cost of all $D_i(\bar{y}_i)$, where the linear constraints φ contain the applicability conditions for the equation as well as size relations for the equation variables.

Example 2.2.3. *Consider the simple Java method* m *shown in Figure 2.1, which invokes the auxiliary method* g*, where* x *is a linked list of boolean values implemented in the standard way. The CR associated to method* m *is shown at the right. The relations* C_m *and* C_g *capture, respectively, the costs of the methods* m *and* g*. Intuitively, in a 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. Equation (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 (equation (2) and equation (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 CRS and the two equations have the same (incomplete) guard, which results in a non-deterministic CRS. Also, variables which do not affect the cost (e.g.,* x*) do not appear in the CRS.*

W.l.o.g., we make two assumptions on the *CRS* defined above:

(a) *Direct recursion*: all recursions are *direct* (i.e., cycles in the call graph are of length one). Direct recursion can be automatically achieved by applying partial evaluation [47] as described in [7]; and

(b) *Standalone CRs: CRs* do not depend on any other *CR*, i.e., the equations do not contain external calls, and thus have the form $\langle C(\bar{x}) = e + \sum_{j=1}^{n} C(\bar{y}_j), \varphi\rangle$. This can be assumed because our approach is compositional. We start by computing bounds for the *CRs* which do not depend on any other *CR*. Then, we continue by replacing the computed bounds on the equations which call such a relation which in turn become standalone. This operation is repeated until no more *CR* needs to be solved. In what follows, *CR* refers to a *standalone CR* in direct recursive form, unless we explicitly state otherwise.

Example 2.2.4. *Assuming that an UB on the number of executed instructions in* g *is* $C_g^+(i,n) = 4 + 5 * \mathsf{nat}(n-i)$*, the CR in Example 2.2.3 becomes standalone by replacing* $C_g(i,n)$ *by* $C_g^+(i,n)$ *in Equation (2), and* $C_g(0,i)$ *by* $C^+(0,i) = 4 + 5 * \mathsf{nat}(i)$ *in equation (3). The resulting standalone CR, after adding constants, results in:*

$$
\begin{array}{lllll}
(1) & \langle C_m(i,n) & = & 3, & \varphi_1 = \{i \geq n\}\rangle \\
(2) & \langle C_m(i,n) & = & 19 + 5 * \mathsf{nat}(n-i) + C_m(i',n), & \varphi_2 = \{i < n, i' = i+1\}\rangle \\
(3) & \langle C_m(i,n) & = & 21 + 5 * \mathsf{nat}(i) + C_m(i,n'), & \varphi_3 = \{i < n, n' = n-1\}\rangle
\end{array}
$$

In order to compute an upper bound for the above CR, the COSTA *analyzer first computes an upper bound for the number of iterations of the loop. For this example, such a bound is* $n - i$*. Afterwards, the* nat*-subexpressions* $\mathsf{nat}(n-i)$ *and* $\mathsf{nat}(i)$ *in equations (2) and (3), respectively, must be "maximized" according to*

the constraints in φ_2 and φ_3. For equation (2), the maximum value results in the same expression. However, in equation (3), the constraint $i < n$ establishes that variable i takes its maximum value when $i = n - 1$. Hence, the maximization of $\mathsf{nat}(i)$ *results in* $\mathsf{nat}(n-1)$*. As a result, the* COSTA *analyzer outputs the cost expression:*

$$C_m^+(i,n) = 3 + \mathsf{nat}(n-i) * \mathsf{max}(\{19 + 5 * \mathsf{nat}(n-i), 21 + 5 * \mathsf{nat}(n-1)\})$$

as an UB on the number of bytecode *instructions that* m *executes. Each Java instruction is compiled to possibly several bytecode instructions. Observe that the use of* nat *is required in order to avoid incorrectly evaluating UBs 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*.*

The evaluation of a *CR* symbol C for a given valuation $\bar{v}$ (integer values), denoted $C(\bar{v})$ is done as follows:

(1) choose a matching equation from those defining C, e.g. $\langle C(\bar{x}) = e + \sum_{i=1}^{k} C(\bar{y}_i), \varphi \rangle$;

(2) choose a solution $\sigma \in [\![\bar{x} \wedge \varphi]\!]$, which indicates that the chosen equation is applicable;

(3) recursively evaluate each recursive call $C(\sigma(\bar{y}_i))$; and

(4) the cost is $\sigma(e)$ plus the results of all calls.

Note that due to the non-deterministic choices in 1 and 2 we might have several solutions for $C(\bar{v})$. The set of all answers for $C(\bar{v})$ is denoted by $Ans(C(\bar{v}))$. Furthermore, even if the original program is deterministic, due to the abstractions performed during the generation of the *CRs*, it might happen that several results can be obtained for a given $C(\bar{v})$. Correctness of the underlying analysis used to obtain the *CRs* must ensure that the actual cost is one of such solutions (see [9]). This makes it possible to use *CRs* to infer both, upper and lower bounds from them.

Example 2.2.5. *Let us evaluate $C_m(0,2)$. We have two matching equations (2) and (3). Applying (2) we get $19 + 2 * 5 + C_m(1,2)$. Similarly the application of* (3) *generates $21 + 0 + C_m(0,1)$. For $C_m(1,2)$ and $C_m(0,1)$ both equations (2) and (3) can be applied again. If we continue executing all possible derivation steps until reaching the base case, the final result for $C(0,2)$ is any of $\{45, 48, 56, 58\}$. The actual cost is guaranteed to be one of these values.*

As shown in the above example, the evaluation of a *CR* for a method m returns a set of values, each one of them being a correct solution. We say that $C^+(\bar{x}) = e$ is an UB for *CR* C if $C^+(\bar{v}) \geq \mathsf{max}(Ans(C(\bar{v})))$. For the above example note that $\{45, 48, 56, 58\}$ are less than $C_m^+(0,2) = 61$. We say that $C^-(\bar{x}) = e$ is an LB for *CR* C if $C^-(\bar{v}) \leq \min(Ans(C(\bar{v})))$.

Chapter 3

Cost Analysis of ABS

This chapter presents, to the best of our knowledge, the first cost analysis for OO concurrent programs. Our analysis is developed on the ABS language, whose concurrency model is based on the notion of concurrently running (groups of) objects, in the spirit of the actor-based and active-objects approaches [59, 61]. These models aim at taking advantage of the inherent concurrency implicit in the notion of object in order to provide programmers with high-level concurrency constructs that help in producing concurrent applications more modularly and in a less error-prone way.

The rest of the chapter is organized as follows. Section 3.1 overviews the ABS language and the example on which we will illustrate our analysis. In Section 3.2, we introduce the notion of cost and the cost models that we want to approximate by means of static analysis. Section 3.3 presents our basic framework for cost analysis of OO programs. In order to increase accuracy, we improve the abstraction of fields in Section 3.4. Finally, the implementation and experimental evaluation is described in Section 3.5.

3.1 Overview of the ABS Language

The concurrency model of Java and C# is based on threads which share memory and are scheduled preemptively, i.e., they can be suspended or activated at any time. To prevent threads from undesired interleavings, low-level synchronization mechanisms such as locks have to be used. Experience has shown that software written in such a thread-based model is error-prone, difficult to debug, verify and maintain. In order to overcome these problems, several concurrency models that take advantage of the inherent concurrency implicit in the notion of object have been developed [59, 61, 46, 34, 52]. They provide programmers with simple language extensions which allow programming concurrent applications with relatively little effort. In particular, the ABS language [35, 42] inherits the concurrency model of Creol [46, 34] and extends it with the possibility of grouping objects together, as in JCoBoxes [59]. For simplicity, we do not consider object groups and assume that all objects are independent. Also, in the presentation, but not in the implementation, we exclude interfaces and interface inheritance.

3.1.1 Informal Description and P2P Example

An ABS *program* consists of a functional, sequential part (with data-type and function declarations) and an imperative, concurrent part (with interfaces, class definitions, and a `main` method to configure the initial state). This distinction allows combining encapsulation and data transfer between objects in such a way that: standard objects are passed by reference and used to make asynchronous calls, while to transfer information between objects, functional data is used. Note that functions and functional data play the role of passive objects in ASP [24] or immutable objects in JCoBox [59]. In order to illustrate our approach, we use as running example a peer-to-peer (P2P) distributed application, adapted from [46]. Figure 3.1 shows the functional fragment of the program with a number of types (only *String* and *Int* are predefined) and some functions which can be executed using standard strict evaluation. In our example, a *FileName* (*FN* for short) is represented as a *String* and the contents of a *File* are represented as a list of *Packet*s. Node

```
data List⟨A⟩ = Nil | Cons(A,List⟨A⟩);
data Set⟨A⟩ = EmptySet | Insert(A,Set⟨A⟩);
data Pairs⟨A,B⟩ = Pair(A,B);
data Map⟨A,B⟩ = EmptyMap |
                InsAss(Pairs⟨A,B⟩,Map⟨A,B⟩);
type FN, Packet = String;
type FNs = Set⟨String⟩;
type File = List⟨Packet⟩;
type Catalog = List⟨Pairs⟨Node,FNs⟩⟩;

def Int length⟨A⟩(List⟨A⟩ list) =
  case list { Nil ⇒ 0 ;
              Cons(p, l) ⇒ 1+length(l);}

def A nth⟨A⟩(List⟨A⟩ list, Int n) =
  case n { 0 ⇒ head(list);
           _ ⇒ nth(tail(list),n-1);}

def B lookup⟨A,B⟩(Map⟨A,B⟩ ms, A k) =
  case ms { InsAss(Pair(k,y),_) ⇒ y;
            InsAss(_,tm) ⇒ lookup(tm,k);}
```

```
def List⟨A⟩ app⟨A⟩(List⟨A⟩ l1, List⟨A⟩ l2) =
  case l1 {
    Nil ⇒ l2;
    Cons(h,t) ⇒ Cons(h,app(t,l2));}

def Bool contains⟨A⟩(Set⟨A⟩ s, A e) =
  case s {
    EmptySet ⇒ False;
    Insert(e, _) ⇒ True;
    Insert(_, xs) ⇒ contains(xs, e);}

def Node findServer(FN f, Catalog c) =
  case c {
    Nil ⇒ null;
    Cons(Pair(s, fs), r) ⇒
      case contains(fs, f) {
        True ⇒ s;
        False ⇒ findServer(f, r); };
  }
```

Figure 3.1: Functional Sequential Part of ABS Implementation of P2P Network

is a class defined in Figure 3.2 together with the rest of the imperative program (which consists of three classes). Class Node reflects that peers can act both as clients and servers. A P2P network is formed by a set of interconnected peers which make the files stored in their database (an object of class DB) available to other peers, without the need for central coordination. In our simple implementation, the only coordination is by means of an object of class Network, whose code is not shown due to space limitations. It suffices to know that nodes learn who their neighbors are by invoking getNeighbors from such an object. A node acting as client triggers computation with searchFile, which first finds a neighbor node s which can provide the file and then asks s for the file using reqFile, which in turn makes a number of activations of the remote method getPacks on s. Whenever possible, size packages are transferred at a time. The field size is set by the constructor of the Node class and then remains constant.

Concurrent objects. The central concept of our concurrency model is that of *concurrent objects*. This means that, conceptually, each object has a dedicated processor and encapsulates a *local heap* which is not accessible from outside this object. This is achieved by making fields accessible only from *this* object. Any other object can only access such fields through method calls. Concurrent objects live in a distributed environment with asynchronous and unordered communication between objects. Such communication is by means of asynchronous method calls that can be seen as triggers of concurrent activity (so-called *tasks*) in the called object. Thus, an object has a set of tasks to be executed and, among them, at most one task is *active* and the others are *suspended* on a task queue. As there is only one active task in each object and each object can only access fields of its own object, data-races cannot occur in ABS. Figure 3.3 shows to the left a possible main method. It creates a network configuration with three nodes, two databases and one Network object (admin). In the graphical representation to the right, we can see that these six objects become distinct concurrent entities which communicate with each other by means of asynchronous calls (shown as labeled arrows) and use *future* variables (shown at the top right) to eventually return/retrieve the results. In n_1, n_2 and n_3, we depict the fact that concurrent objects have their own heaps, their queue of pending tasks and an active task (if any). Though not shown graphically, the same holds for db_1, db_2,

```
class DB {
  Map⟨FN,File⟩ dbf;
  DB(Map⟨FN,File⟩ db){dbf = db;}
  File getFile(FN fn) {
    return lookup(dbf,fn); }
  Int lengthDB(FN fn) {
    return length(lookup(dbf,fn)); }
  Unit storeFile(FN fn, File f) {
    dbf=InsAss(Pair(fn,f),dbf); }
}
class Node {
  DB db; Catalog catalog;
  List⟨Node⟩ myNeighbors;
  Network admin; Int size;
  Node(DB dbf, Int s) {db = dbf; size = s;}
  Catalog availFiles(List⟨Node⟩ sList)
    {...}
  List⟨Packet⟩ getPacks(FN fn, Int ps, Int n) {
    File f = Nil; File res = Nil;
    Fut⟨File⟩ ff;
    ff = db ! getFile(fn);
    await ff?; f = ff.get;
    while(ps > 0) {
      res = Cons(nth(f,n+ps-1),res);
      ps = ps-1;
    }
    return res; }
  Int lengthNode(FN fn) {
    Fut⟨Int⟩ length;
    length=db ! lengthDB(fn);
    await length?; return length.get; }
  Unit reqFile(Node s, FN fn) {
    Fut⟨Int⟩ l1; Fut⟨List⟨Packet⟩⟩ l2;
    File f = Nil;
    List⟨Packet⟩ ps = Nil;
    Int i = 0; Int incr = 0;
    l1 = s ! lengthNode(fn);
    await l1?; i = l1.get;
    while (i > 0) {
      if (size > i) incr = i;
      else incr = size;
      i = i - incr;
      l2 = s ! getPacks(fn,incr,i);
      await l2?; ps = l2.get;
      f = app(ps,f); }
    db ! storeFile(fn,f); }
  Unit searchFile(FN f) {
    Fut⟨Catalog⟩ c;
    Fut⟨List⟨Node⟩⟩ f;
    Node s;
    await admin != null;
    f = admin ! getNeighbors(this);
    await f?; myNeighbors = f.get;
    c = this ! availFiles(myNeighbors);
    await c?; catalog = c.get;
    s = findServer(f, catalog);
    if (s != null)
      this ! reqFile(s,f); }
}
class Network {
  Node node1, node2, node3;
  Network(Node n1, Node n2, Node n3) {...}
  List⟨Node⟩ getNeighbors(Node caller){...}
}
```

Figure 3.2: OO Concurrent Part of ABS Implementation of P2P Network

and admin.

Future variables and Synchronization. Process scheduling is by default non-deterministic, but controlled by *processor release points* and *future variables* in a cooperative way. After asynchronously calling $f := o \,!\, m(\overline{e})$, the caller may proceed with its execution without blocking on the call. Here f is a future variable which refers to a return value which has yet to be computed. There are two operations on future variables, which control external synchronization. First, the guard **await** $f?$ suspends the active task (allowing other tasks in the object to be scheduled) unless a return to the call associated with f has arrived. Second, the return value is retrieved using $f.\mathbf{get}$, which blocks all execution in the object until the return value is available. An unconditional **release** instruction (not used in the example) suspends the current task and lets a pending task in.

3.1.2 A Rule-based Intermediate Language

As customary in the formalization of static analyses for realistic languages, we develop our analysis on an intermediate representation (IR) which removes syntactic sugar and allows a clearer and more concise presentation. Similar representations are used for Java (and Java bytecode) and .NET, e.g., those in [62, 10,

```
db = InsAss(Pair("file0",
        Cons("a", Cons("b",
        Cons("c", Nil)))))
      InsAss(Pair("file1",
        Cons("d", Cons("e", Nil))),
      EmptyMap));
db1 = new DB(EmptyMap);
db2 = new DB(db);
n1 = new Node(db1, 2);
n2 = new Node(db1, 1);
n3 = new Node(db2, 1);
admin = new NetWork(n1, n2, n3);
n1 ! setAdmin(admin);
n2 ! setAdmin(admin);
n3 ! setAdmin(admin);
n1 ! searchFile("file0");
n2 ! searchFile("file1");
```

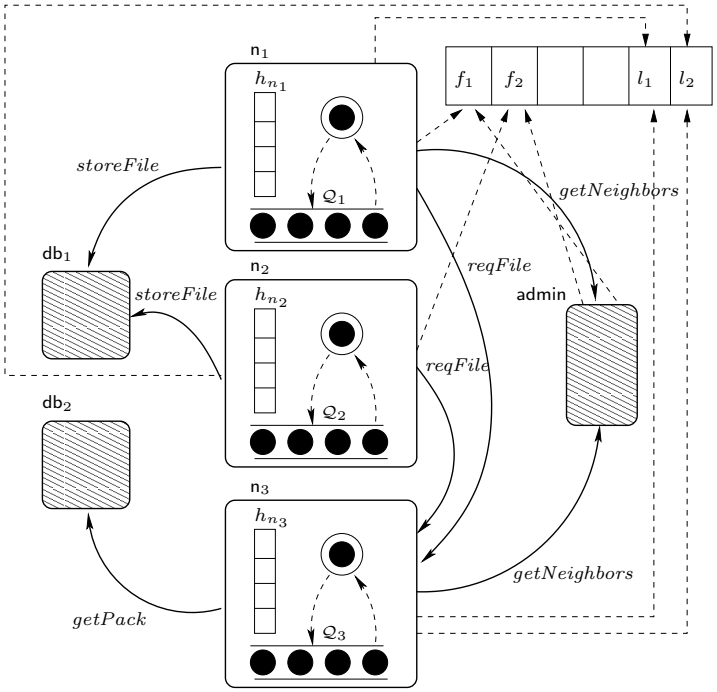


Figure 3.3: Main method (left). Overview of Concurrent Activity (right).

60, 37]. Essentially, all these tools work by first building the control flow graph (CFG) from the program and then representing each block of the CFG in the intermediate language. In our IR, a program consists of a set of *procedures* defined by means of *rules*. Namely, each method in the original program is represented by one or more procedures in the IR, which in turn are defined by one or more *guarded* rules. As the translation is identical to the one for Java programs, we do not go into the technical details (see [62, 10, 60, 37]), but rather illustrate it by example.

Example 3.1.1. *Figure 3.4 shows the CFG (right) and the IR (left) of method* reqFile. *Loops are extracted in separate CFGs to enable compositional cost analysis on them. It can be observed that the method is represented by four procedures, reqFile, loop, if and if^c, each of them defined by means of guarded rules.* $\overline{inp}$ *stands for* $\langle s, fn, f, ps, i, incr, l_2\rangle$ *and* $\overline{out}$ *for* $\langle f, ps, l_2, i, incr\rangle$. *Guards in rules state the conditions under which the corresponding blocks in the CFG can be executed. When there is more than one successor in the CFG, we just create a* continuation procedure *and a corresponding call in the rule. Blocks in the continuation will be in turn defined by means of guarded rules (with mutually exclusive conditions). As a result of the translation, all forms of iteration in the program are represented by means of* recursive *calls. The input variables to the entry procedure for* reqFile *are those of the corresponding method and additionally the reference to the this object. When calling a block, we pass as arguments all local variables that are needed in the block. The heap remains as an implicit global variable. Observe that there is almost a one-to-one correspondence between instructions in the original program and in its IR, except for data type declarations (which are not needed for the analysis) and the* while *loop (which has been replaced by a call to the procedure defining the loop).*

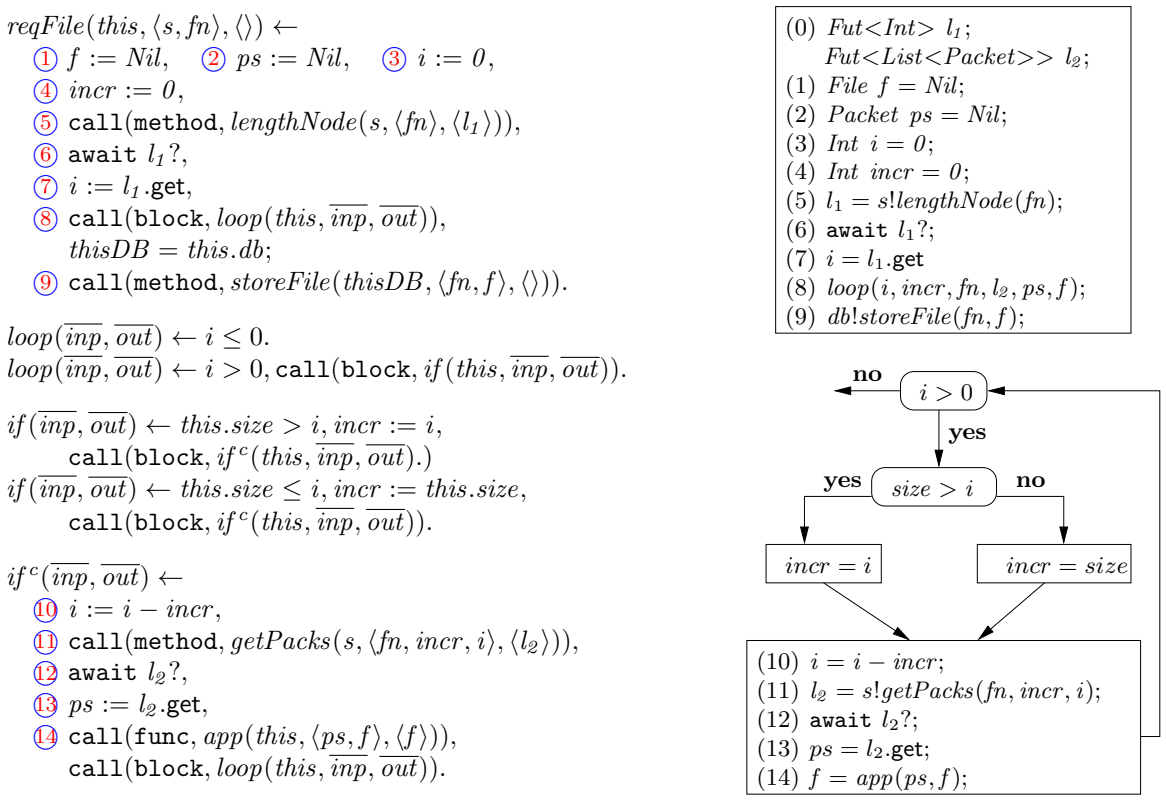


Figure 3.4: The IR and CFG for method reqFile.

3.1.3 The Abstract Syntax

Rules in the IR adhere to this grammar:

$$
\begin{array}{lcl}
\mathit{rule} & ::= & m(\mathit{this}, \bar{x}, \bar{y}) \leftarrow g, b_1, \ldots, b_n. \\
g & ::= & \mathit{true} \mid g \wedge g \mid x? \mid \mathit{exp}\ \mathit{op}\ \mathit{exp} \mid \mathit{match}(x,t) \mid \mathit{nonmatch}(x,t) \\
b & ::= & x := \mathit{exp} \mid \mathit{this}.f := \mathit{exp} \mid x := \mathsf{new}\ C \mid \\
 & & \texttt{call}(\mathit{callType}, m(\mathit{receiver}, \bar{x}, \bar{y})) \mid \texttt{await}\ g \mid \texttt{release} \mid x := y.\texttt{get} \\
\mathit{exp} & ::= & \mathsf{null} \mid \mathit{aexp} \mid t \mid \mathit{this}.f \\
\mathit{aexp} & ::= & x \mid n \mid \mathit{aexp} - \mathit{aexp} \mid \mathit{aexp} + \mathit{aexp} \mid \mathit{aexp} * \mathit{aexp} \mid \mathit{aexp} / \mathit{aexp} \\
t & ::= & x \mid \mathit{Co}(\bar{t}) \\
\mathit{op} & ::= & < \mid > \mid = \mid \neq \mid \leq \mid \geq
\end{array}
$$

where $m(\mathit{this}, \bar{x}, \bar{y})$ is the *head* of the rule, *this* is the identifier of the object on which the method or function is executing, g specifies the conditions for the rule to be applicable and $b_1, \ldots, b_n$ is the rule *body*. Calls are of the form $\texttt{call}(\mathit{callType}, m(\mathit{receiver}, \bar{x}, \bar{y}))$ where $\mathit{callType} \in \{\texttt{method}, \texttt{block}, \texttt{func}\}$ in order to distinguish between calls to methods, intermediate blocks and functions; *receiver* is a variable that refers to the receiver object; the variables $\bar{x}$ are the formal parameters; and the variables $\bar{y}$ the return values. For blocks and functions, variable *receiver* is always *this*. For methods, $\bar{y}$ is either empty or contains a single output variable. We assume that future variables are used in `await` instructions but not in rule guards. Guards of the form $\mathit{match}(x,t)$ and $\mathit{nonmatch}(x,t)$ in the IR simulate **case**-expressions used to handle functional data. We assume $x \notin \mathit{vars}(t)$. Note that $\mathit{match}(x,t)$ modifies $\mathit{vars}(t)$ when it succeeds. An instruction $\mathsf{new}\ \mathsf{C}(\bar{\mathsf{t}})$ in ABS is represented in the IR by $\mathsf{new}\ C$ followed by a call to the class constructor with the corresponding parameters $\bar{\mathsf{t}}$.

A program consists of classes, functions, and a main method from which the execution starts. A class C consists of a set of methods and a set of fields $\bar{f}_C$. A method $C.m$ is defined by a set of rules such that there is a single rule named $C.m$ (the method entry). The other rules are intermediate procedures that are

used only within the method, using the `block` call. A function is a (global) set of rules that is accessible from any method (using the `func` call), and therefore it cannot access nor modify fields. The main method does not belong to any class.

3.1.4 Operational Semantics

An execution state (or *configuration* $\mathcal{S}$) has the form $\{\!\{a_1, \ldots, a_n\}\!\}$, where an a_i can be either an *object*, a *future* event, or a method invocation. An object is of the form $\mathtt{ob}(o, C, h, \langle tv; s\rangle, \mathcal{Q})$, where o is the *object identifier*, C is its class name, h is its *heap*, tv a *table of local variables*, s is a sequence of instructions to be executed by the current task, and $\mathcal{Q}$ is the set of pending tasks. A heap h maps *field names* (declared in C) to $\mathbb{V} = \mathbb{Z} \cup \{\mathsf{null}\} \cup \mathit{Objects} \cup \mathit{Terms}$, where *Objects* (resp., *Terms*) denotes the set of object identifiers (resp., functional terms). A table of variables tv maps local variables to $\mathbb{V}$. It contains the special entry `destiny` to associate the return variable of a method to the corresponding future variable.

Future events have the form $\mathtt{fut}(\mathtt{fn}, v)$ where $v \in \mathbb{V} \cup \bot$. The symbol $\bot$ indicates that `fn` does not have a value yet. A method invocation is of the form $\mathtt{invoke}(m(o, \bar{x}), tv, \mathtt{fn})$, where `fn` is the future variable in which the return value should be stored. For simplicity, we assume that all methods return a value. The *operational semantics* is given in a rewriting-based style, where, at each step, a subset of the state is rewritten according to the following rules:

$$(1)\ \frac{v = \mathit{eval}(\mathit{exp}, h, tv, \mathit{Obj}),\ x \in \mathit{dom}(tv)}{\{\!\{\mathtt{ob}(o, C, h, \langle tv; x := \mathit{exp} \cdot s\rangle, \mathcal{Q}) | \mathit{Obj}\}\!\} \rightsquigarrow \{\!\{\mathtt{ob}(o, C, h, \langle tv[x \mapsto v]; s\rangle, \mathcal{Q}) | \mathit{Obj}\}\!\}}$$

$$(2)\ \frac{v = \mathit{eval}(\mathit{exp}, h, tv, \mathit{Obj})}{\{\!\{\mathtt{ob}(o, C, h, \langle tv; \mathit{this}.f := \mathit{exp} \cdot s\rangle, \mathcal{Q}) | \mathit{Obj}\}\!\} \rightsquigarrow \{\!\{\mathtt{ob}(o, C, h[f \mapsto v], \langle tv; s\rangle, \mathcal{Q}) | \mathit{Obj}\}\!\}}$$

$$(3)\ \frac{o' \text{ is fresh, } h' \text{ is an empty mapping}}{\begin{array}{c}\{\!\{\mathtt{ob}(o, C, h, \langle tv; x := \mathsf{new}\ D \cdot s\rangle, \mathcal{Q}) | \mathit{Obj}\}\!\} \rightsquigarrow \\ \{\!\{\mathtt{ob}(o, C, h, \langle tv, x := o'; s\rangle, \mathcal{Q}), \mathtt{ob}(o', D, h', \epsilon, \emptyset) | \mathit{Obj}\}\!\}\end{array}}$$

$$(4)\ \frac{\begin{array}{c}\mathit{callType} \in \{\mathtt{block}, \mathtt{func}\}, r \equiv p(o, \bar{x}, \bar{y}) \leftarrow g, b_1, \ldots, b_n \ll_{tv} P, \\ tv' \text{ is a default mapping over } \mathit{vars}(r) \setminus (\bar{x} \cup \bar{y}),\ \mathit{eval}(g, h, tv \cup tv', \mathit{Obj}) = \mathit{true}\end{array}}{\begin{array}{c}\{\!\{\mathtt{ob}(o, C, h, \langle tv; \mathtt{call}(\mathit{callType}, p(o, \bar{x}', \bar{y})) \cdot s\rangle, \mathcal{Q}) | \mathit{Obj}\}\!\} \rightsquigarrow \\ \{\!\{\mathtt{ob}(o, C, h, \langle tv \cup tv'; b_1, \ldots, b_n \cdot s\rangle, \mathcal{Q}) | \mathit{Obj}\}\!\}\end{array}}$$

$$(5)\ \frac{o'' = \mathit{eval}(o', h, tv, \mathit{Obj}), \bar{v} = \mathit{eval}(\bar{x}, h, tv, \mathit{Obj}), \mathtt{fn} \text{ is a fresh future name}}{\begin{array}{c}\{\!\{\mathtt{ob}(o, C, h, \langle tv; \mathtt{call}(\mathtt{method}, m(o', \bar{x}, y)) \cdot s\rangle, \mathcal{Q}) | \mathit{Obj}\}\!\} \rightsquigarrow \\ \{\!\{\mathtt{ob}(o, C, h, \langle tv; y := \mathtt{fn} \cdot s\rangle, \mathcal{Q}), \mathtt{invoke}(m(o'', \bar{v}), \mathtt{fn}), \mathtt{fut}(\mathtt{fn}, \bot) | \mathit{Obj}\}\!\}\end{array}}$$

$$(6)\ \frac{\begin{array}{c}r \equiv m(\mathit{this}, \bar{x}, y) \leftarrow g, b_1, \ldots, b_n \ll P,\ tv' \text{ is a default mapping over } \mathit{vars}(r), \\ tv'' = tv'[\mathit{this} \mapsto o, \bar{x} \mapsto \bar{v}, \mathtt{destiny} \mapsto (y, \mathtt{fn})], \mathit{eval}(g, h, tv'', \mathit{Obj}) = \mathit{true}\end{array}}{\begin{array}{c}\{\!\{\mathtt{ob}(o, C, h, \langle tv; s\rangle, \mathcal{Q}), \mathtt{invoke}(m(o, \bar{v}), \mathtt{fn}) | \mathit{Obj}\}\!\} \rightsquigarrow \\ \{\!\{\mathtt{ob}(o, C, a, \langle tv; s\rangle, \{\langle tv''; b_1, \ldots, b_n\rangle | \mathcal{Q}\}) | \mathit{Obj}\}\!\}\end{array}}$$

$$(7)\ \frac{(y, \mathtt{fn}) = tv(\mathtt{destiny}),\ v = tv(y)}{\{\!\{\mathtt{ob}(o, C, h, \langle tv; \epsilon\rangle, \mathcal{Q}), \mathtt{fut}(\mathtt{fn}, \bot) | \mathit{Obj}\}\!\} \rightsquigarrow \{\!\{\mathtt{ob}(o, C, h, \epsilon, \mathcal{Q}), \mathtt{fut}(\mathtt{fn}, v) | \mathit{Obj}\}\!\}}$$

$$(8)\ \frac{\mathtt{fn} = tv(y),\ v \neq \bot}{\{\!\{\mathtt{ob}(o, C, h, \langle tv; x := y.\mathtt{get} \cdot s\rangle, \mathcal{Q}), \mathtt{fut}(\mathtt{fn}, v) | \mathit{Obj}\}\!\} \rightsquigarrow \{\!\{\mathtt{ob}(o, C, h, \langle tv; x := v \cdot s\rangle, \mathcal{Q}) | \mathit{Obj}\}\!\}}$$

$$(9)\ \frac{\mathit{eval}(g, h, tv, \mathit{Obj}) = \mathit{true}}{\{\!\{\mathtt{ob}(o, C, h, \langle tv; \mathtt{await}\ g \cdot s\rangle, \mathcal{Q}) | \mathit{Obj}\}\!\} \rightsquigarrow \{\!\{\mathtt{ob}(o, C, h, \langle tv; s\rangle, \mathcal{Q}) | \mathit{Obj}\}\!\}}$$

$$(10)\ \frac{\mathit{eval}(g, h, tv, \mathit{Obj}) = \mathit{false}}{\{\!\{\mathtt{ob}(o, C, h, \langle tv; \mathtt{await}\ g \cdot s\rangle, q) | \mathit{Obj}\}\!\} \rightsquigarrow \{\!\{\mathtt{ob}(o, C, h, \langle tv; \mathtt{release} \cdot \mathtt{await}\ g \cdot s\rangle, \mathcal{Q}) | \mathit{Obj}\}\!\}}$$

$$(11)\ \frac{}{\{\!\{\mathtt{ob}(o, C, h, \langle tv; \mathtt{release} \cdot s\rangle, \mathcal{Q}) | \mathit{Obj}\}\!\} \rightsquigarrow \{\!\{\mathtt{ob}(o, C, h, \mathtt{idle}, \{\!\{\langle tv; s\rangle | \mathcal{Q}\}\!\}) | \mathit{Obj}\}\!\}}$$

$$(12)\ \frac{b = \epsilon \ or\ b = \texttt{idle}}{\{\!|\texttt{ob}(o, C, h, b, \mathcal{Q})|Obj|\!\} \rightsquigarrow \{\!|\texttt{ob}(o, C, h, s, \mathcal{Q} - \{\!|s|\!\})|Obj|\!\}}$$

Let us intuitively explain the semantics. Function $eval(exp, h, tv, Obj)$ evaluates an expression exp using the heap h and the variables table tv in the standard way. If exp is a future, then it uses Obj to see if it has a value. If exp is a *match* instruction, *eval* modifies the matched variables in the corresponding variables table. The notation $tv[x \mapsto v]$ (resp. $h[f \mapsto v]$) in the first two rules stands for storing v in the local variable x (resp. field f). In rule 3, it can be observed that an instruction new creates a new object initially empty. We assume that, after creating the object, the scheduler always picks the task associated to the constructor before selecting any other task. In rule 4, a call to a block or a function is resolved by finding a matching rule and add (a renamed apart version of) its body to the sequence of instructions to be executed.

The most important points in the semantics are (a) the treatment of method invocations and future variables (rules 5-7) and (b) synchronization operations (rules 8-12). As regards (a), when we find an asynchronous call in rule 5, we create a (fresh) future variable fn on which the result will be returned and establish the link between the return variable y of the method and fn by means of an assignment instruction. The fact that the asynchronous call reaches the object occurs in rule 6, where the asynchronous call is dequeued for execution, and the destiny future variable is stored in the local variables table. Then in rule 7, when the corresponding method finishes execution, the future variable is updated with the returned value. As regards (b), the instruction get blocks execution until the future variable has a value in rule 8. If the evaluation of the guard in an await instruction succeeds, execution continues in rule 9. If it fails, the processor is released (rule 10) to allow another task to become active. This can be seen in rule 11 in which the task becomes idle. In rule 12 another task is dequeued (because the current one terminated or released the processor).

Executions start from a main method. *Initial configurations* have the form $\{\!|\texttt{ob}(\texttt{main}, \bot, \bot, \langle tv; \bar{b}\rangle, \emptyset)|\!\}$ where local variables in tv are initialized to default values. The execution ends in a *final configuration* $\mathcal{S}$ in which all events are either future events or objects of the form $\texttt{ob}(o, C, h, \epsilon, \emptyset)$. Execution proceeds non-deterministically by applying the above execution steps. When there is no rule to apply the execution stops. Executions can be regarded as *traces* of the form $\mathcal{S}_0 \rightsquigarrow \mathcal{S}_1 \rightsquigarrow \cdots \rightsquigarrow \mathcal{S}_n$ where $\mathcal{S}_n$ is a final configuration.

Example 3.1.2. *Let us show two states of the execution of the* main *method in Figure 3.3. After executing the constructors we reach a configuration with seven objects:*

$$\{\!|\ \texttt{ob}(\texttt{main}, \bot, \bot, \langle tv_{\texttt{main}}; b_1 \cdots b_5\rangle, \emptyset), \texttt{ob}(o_{db_1}, DB, h_{db_1}, \epsilon, \emptyset), \texttt{ob}(o_{db_2}, DB, h_{db_2}, \epsilon, \emptyset),$$
$$\texttt{ob}(o_{n_1}, Node, h_{n_1}, \epsilon, \emptyset), \texttt{ob}(o_{n_2}, Node, h_{n_2}, \epsilon, \emptyset), \texttt{ob}(o_{n_3}, Node, h_{n_3}, \epsilon, \emptyset), \texttt{ob}(\texttt{o}_{admin}, \ldots)|\!\}$$

After starting the execution of searchFiles *in o_{n_1} and o_{n_2}, we might reach this configuration which contains* invoke *events and* fut *events associated to the execution of the asynchronous call to getNeighbors in the corresponding objects:*

$$\{\!|\ \texttt{invoke}(getNeighbors(o_{admin}, o_{n_1}, f_1), \texttt{fn}_1),\ \texttt{invoke}(getNeighbors(o_{admin}, o_{n_2}, f_2), \texttt{fn}_2),$$
$$\texttt{fut}(\texttt{fn}_1, \bot), \texttt{fut}(\texttt{fn}_2, \bot), \texttt{ob}(o_{admin}, NetWork, h_{admin}, \epsilon, \emptyset), \ldots |\!\}$$

The tables of variables of both objects must contain references to the future variables in which the results will be returned $tv_{n_1}(f_1) = \texttt{fn}_1$ and $tv_{n_2}(f_2) = \texttt{fn}_2$.

3.2 Cost and Cost Models for Concurrent Programs

In this section, we define the notion of cost that we aim at approximating by static analysis. In the execution of sequential programs, the *cost of an execution trace* is obtained by applying a certain *cost model* to each step of the trace. A cost model assigns a cost to the different instructions, and the cost of a trace is defined as the sum of the costs of its execution steps. In our setting, this simple notion of cost has to be extended because, rather than considering a single machine in which all steps are performed, we have

a potentially distributed setting, with multiple objects possibly running concurrently on different central processing units (CPUs). Thus, rather than aggregating the cost of all executing steps, it is more useful to treat execution steps which occur on different computing infrastructures separately. This is important since different infrastructures might have different configurations, such CPU, memory, etc. Therefore, inferring the resource consumption for each infrastructure separately helps in identifying those that might exceed the resource limit (e.g., run out of memory). This information is not deducible from the total resource consumption. With this aim, we adopt the notion of *cost centers* [57] proposed for profiling functional programs. Since the concurrency unit of our language is the object, cost centers are used to charge the cost of each step to the cost center associated to the object where the step is performed.

In what follows, we consider annotated traces of the form $t = \mathcal{S}_0 \rightsquigarrow_{o_1} \ldots \rightsquigarrow_{o_n} \mathcal{S}_n$ where the annotation o_i refers to the object identifier that has been selected (from $\mathcal{S}_{i-1}$) for that execution step. For a given set of objects O, we let $t|_O = \{\mathcal{S}_i \rightsquigarrow_{o_i} \mathcal{S}_{i+1} \mid \mathcal{S}_i \rightsquigarrow_{o_i} \mathcal{S}_{i+1} \in t,\ o_i \in O\}$, i.e., the set of execution steps that are performed on objects from O. A cost model $\mathcal{M}$ is a function $\mathcal{M} : \mathit{Ins} \mapsto \mathbb{R}$, which maps instructions built using the grammar in Section 3.1.3 to real numbers. We assume that cost models map those instructions introduced by the semantics to zero, i.e., `idle`, `release` and `await` which are not part of the program are not counted. This means that we consider the cost associated to executing the instructions of the program but not implementation aspects of concurrency, like how many times an `await` is checked until it succeeds. The cost of an execution step $\mathcal{S}_i \rightsquigarrow_{o_i} \mathcal{S}_{i+1}$ w.r.t. $\mathcal{M}$, denoted $\mathcal{M}(\mathcal{S}_i \rightsquigarrow_{o_i} \mathcal{S}_{i+1})$, is defined as $\mathcal{M}(b)$ where $\texttt{ob}(o_i, C, h, \langle tv; b \cdot s\rangle, \mathcal{Q}) \in \mathcal{S}_i$. Cost centers can be defined at the level of objects, or sets of objects. It is common that a group of objects becomes a concurrency unit (see [59]). The cost of t w.r.t. a cost model $\mathcal{M}$ and a set of cost centers O (i.e., set of objects) is defined as $\mathcal{C}(t, O, \mathcal{M}) = \sum_{e \in t|_O} \mathcal{M}(e)$. Observe that in this setting, it is also possible to apply a different cost model to the different cost centers, e.g., if we want to take into account the particular features of the machine on which the component will be deployed.

3.2.1 Cost Models

In the concurrent setting, in addition to traditional cost models used in sequential computation (e.g., number of instructions or memory allocated), we consider cost models specifically tailored to the case of concurrent applications. The cost models we consider are platform independent (e.g., worst-case execution time is left out of study). Nonetheless, information inferred by our framework (e.g., size relations) can be useful also for platform-dependent cost models.

3.2.1.1 Termination.

We consider termination as the simplest cost model, $\mathcal{M}_{termin}$, which is used to infer if the resource consumption is bounded, but does not provide actual bounds. Formally, we define it as $\mathcal{M}_{termin}(b) = 0$ for all instructions b. As the accumulated cost is zero, the only thing that cost analysis must do is to infer bounds on the number of iterations of loops (thus prove loops terminating). Hence, in this cost model, inferring cost boils down to prove termination.

3.2.1.2 Number of Instructions.

The most traditional model, denoted $\mathcal{M}_{ins}$, assigns cost 1 to all instructions, except to calls to blocks (as they do not appear in the original program):

$$\mathcal{M}_{ins}(b) \quad = \quad \begin{cases} 0 & b \equiv \texttt{call(block,\_)} \\ \mathcal{M}_{ins}(g_1) + \mathcal{M}_{ins}(g_2) & b \equiv g_1 \wedge g_2 \\ 1 & \mathit{otherwise} \end{cases}$$

3.2.1.3 Memory Consumption.

Since objects are meant to be the concurrency units while data structures will be constructed using terms, the cost model that estimates the amount of memory, $\mathcal{M}_{mem}$, measures the size of functional data:

$$\mathcal{M}_{mem}(b) \quad = \quad \begin{cases} |Co(p_1,\ldots,p_n)| & b \equiv x := Co(p_1,\ldots,p_n), \\ 0 & otherwise \end{cases}$$

where $|x| = 0$ if x is a variable, and $|Co(p_1,\ldots,p_n)| = size(Co) + |p_1| + \ldots + |p_n|$ such that $size(Co)$ denotes the memory required by the data constructor Co.

3.2.1.4 Concurrent Objects.

Since objects represent the concurrency units, an interesting cost model is counting the total number of objects created along the execution (note that objects can be created inside loops). This provides a useful indication of the amount of parallelism that might be achieved. This cost model is defined as $\mathcal{M}_{obj}(b) = 1$ if $b \equiv \mathsf{new}\ c$ and $\mathcal{M}_{obj}(b) = 0$ otherwise.

3.2.1.5 Spawned Processes & Remote Requests.

By counting `call(method, _)`, we can infer the *task-level* of the program [3], i.e., the number of tasks that are spawned along an execution. Knowledge on the task-level of the program is useful for both understanding and debugging parallel programs. Our analysis can infer a task-level which is larger (or smaller) than the programmer's expectations. This can help find bugs in the concurrent program, e.g., the analysis results would be "unbounded" when an instruction which spawns new tasks is (wrongly) placed within a loop which does not terminate. The task-level is also useful for optimizing and finding optimal deployment configurations, e.g., when parallelizing the program, it is not profitable to have more processors than the inferred task-level. This cost model can be specialized to count the number of calls to specific methods or objects. For instance, if we count `call(method, _(o, _, _))`, we obtain UBs on the number of requests to a remote component o. This can be useful to detect that some components are receiving a large amount of remote requests while other siblings are idle most of the time.

3.3 The Basic Cost Analysis Framework

There has been a significant development in cost analysis for sequential Java-like languages during the last years. Before starting HATS, the main focus had been primarily on automatically producing (cost) RRs from realistic imperative programs. This step is non-trivial since programs typically contain different forms of iterations (recursion, do-while, while, for, etc.), different types of variables and data, dynamic dispatch, etc., which must be converted into mathematical relations. The IR can be seen as a normal form representation of the program that makes this step feasible (and that we directly borrow from [10]). During the first year of HATS, we have focused on making cost analysis *field-sensitive* [2], i.e., being able to approximate data stored in the heap such that when the resource consumption of the program depends on such data, upper/lower bounds can still be generated. Therefore, the starting point of our work is a powerful field-sensitive cost analysis framework for sequential OO programs. When lifting such a framework to the concurrent setting, the main two difficulties and novelties are:

1. It is widely recognized that, due to the possible interleaving between tasks, tracking values of data stored in the heap is challenging. In Sec. 3.3.1, we present the basic, novel, *field-sensitive* size analysis for the concurrent setting, and in Sec. 3.4 we discuss how to further improve its precision.

2. Standard RRs (in the sequential setting) assume a single cost center which accumulates the cost of the whole execution. In Sec. 3.3.2, we propose a novel form of RRs which are parametric on the *cost centers* to which cost must be assigned.

3.3.1 Field-Sensitive Size Analysis for Concurrent OO Programs

The objective of the size analysis component is to infer precise *size relations* which allow us to reason on how the size of data changes along a program's execution. When the program contains loops, size relations are fundamental to bound the number of iterations that loops perform. For instance, if a loop traverses a non-cyclic list, we aim at automatically inferring a size relation that ensures that the length of the list decreases at each loop iteration. This in turn will allow us to bound the number of iterations by the length of the list. In what follows, we present the size analysis component in three steps:

- *Size measures.* We first recall the notion of size measure that maps data structures to numerical values that represent their sizes.
- *Abstract compilation.* We present a sound abstraction which compiles each program instruction into corresponding size constraints (w.r.t. the chosen size measures). Our main challenge is to keep as much information on global data (i.e., fields) as possible while still being sound in the concurrent execution.
- *Input-Output relations.* We infer input-output (IO) size relations for increasing precision.

3.3.1.1 Size measures.

When a program manipulates numerical data, its cost often depends on the initial (integer) values of the corresponding variables, and when it manipulates terms (i.e., data-structures), its cost usually depends on the *size* of the concrete input. For example, the cost of traversing a list depends on the length of the list. Therefore, we first need to define the meaning of *size of a term*, which is also known as *norm* [21]. A norm is a function that maps terms to numerical values that represent their sizes. For example, the *term-size* norm calculates the number of type constructors in a given term. It can be defined recursively as follows: $\left| Co(p_1, \ldots, p_n) \right|_{ts} = 1 + \Sigma_{i=1}^{n} \left| p_i \right|_{ts}$ and $\left| x \right|_{ts} = x$. Another common norm is the *term-depth* which calculates the length of the longest path in the term: $\left| Co(p_1, \ldots, p_n) \right|_{td} = 1 + max(\left| p_1 \right|_{td}, \ldots, \left| p_n \right|_{td})$ and $\left| x \right|_{td} = x$. Any norm can be used in the analysis, depending on the nature of data structures used in the program. In what follows, w.l.o.g., we use the term-size norm.

Example 3.3.1. *Given* $t \equiv Cons(node_3, Nil)$*, we have* $\left| t \right|_{ts} = 2 + node_3$ *and* $\left| t \right|_{td} = 1 + \mathsf{max}\{node_3, 1\}$*. For the term stored in variable* db *in Figure 3.3, we have* $\left| \mathsf{db} \right|_{ts} = 19$ *and* $\left| \mathsf{db} \right|_{td} = 6$*, assuming that the size of a string is* 1 *in both cases.*

In addition to terms and numerical values, our language includes reference variables that point to objects created using new c (and future variables). In OO, objects represent data structures and, therefore, their sizes should be tracked as we do for terms since they might directly affect the number of iterations a loop can make. This is usually done using the path-length abstraction [60] (which is the OO version of term-depth). However, in our context, objects are intended to simulate concurrent computing entities and not data structures. It is hence not common that they directly affect loop iterations. Therefore, ignoring their sizes is sound and precise enough in most cases. A slightly more precise abstraction can distinguish between the case in which a reference variable points to an object (size 1) or to null (size 0). The size of a future variable is like the size of the value it holds. This is sound because future variables cannot be used directly, but rather only through the get instruction (which blocks until it gets a value).

3.3.1.2 Abstract Compilation.

One of the main challenges when (statically) analyzing OO programs is to model the shared memory or heap. Our starting point is the field-sensitive size analysis for Java programs developed during the first year of HATS [2], where we propose to model fields as local variables whenever doing so is sound. Thus, one can then rely on a fully field-insensitive analysis to infer information on the fields. Soundness requires that the field to be tracked meets two conditions: (1) its memory location does not change (this property is

known as reference constancy [2, 1]) and (2) the field is accessed always through the same reference (i.e., not through aliases). Both conditions can often be proven statically and the transformation of fields into local variables can then be successfully applied for many fragments of the program. If we ignore concurrency, this approach could be directly adopted for our language. Even more, since different objects do not share heaps in ABS, and terms (i.e., data structures) are immutable, the transformation is simpler than in Java and can be applied locally to classes.

Example 3.3.2. *Consider the loop in the* reqFile *method. By ignoring the* `await` *instruction, it is clear that the two soundness conditions for the field* size *will hold, i.e.,* size *refers to the same memory location in all iterations and there are no aliases to it. Hence, for the sake of analysis, we can track* size *as if it were a local variable. However, concurrency introduces new challenges. Unlike local variables, different tasks, say A and B, running on the same object o share the values of variables stored in the heap. Thus, while A is executing, B may modify the values stored in o's heap, and therefore affect A's behavior. Therefore, A cannot assume that the values (fields) stored in o's heap are preserved in each execution step. In order to address this problem, it is essential to identify those program points in which other tasks might modify the shared memory, and understand how the shared memory is modified. In the context of the considered concurrent objects model, this happens in the following situations:*

1. *The execution of a task A is interleaved with the execution of another task B (of the same object) only when A explicitly executes* `release` *or* `await`*. Let us add (only for the sake of this example) the following method to class* Node*:* void p() {size = size − 2;}*. The interleaved execution of* reqFile *and* p *might introduce non-termination to the loop of* reqFile*.*

2. *Since method invocations are asynchronous, the actual execution of the invoked method does not necessarily start at the point in time when the invocation is made. For instance, given method* p *above, the invocation of* getPacks *in the loop could start after the value of* size *is decreased by* p*. Depending on the pending requests and the scheduler, the invocation can start executing at a later point in time, in a different configuration. The values of the local variables are preserved, but not those stored in the heap.*

The first point suggests that, if a sequence of instructions does not include `release` or `await`, then the shared memory can be tracked locally, since no other tasks can modify it. Namely, fields behave as local variables. The second point suggests that, when starting the execution of a method, we cannot assume that the initial state of the shared memory is identical to that when the method has been invoked. As a first attempt, we present a safe abstraction for each instruction that loses all information about fields when they cannot be tracked locally: (1) at `release` or `await` points and (2) at method entries.

An abstract state, at a program point, maps each program variable and field to a set of integers that represent possible sizes for their concrete values at that program point. Abstract states are often represented by sets of linear constraints whose solutions define the above mappings. This representation allows describing relations such as (1) the size of x is always smaller than the size of y; and (2) the size of x decreases by 1 in two consecutive states. These relations are essential for infering cost and proving termination. When representing a state using constraints, the basic building blocks are (simpler) constraints that describe the effect of each instruction b on a given state. We refer to such constraints as the *abstraction* of b. Figure 3.5 depicts the abstraction of all instructions.

In order to abstract an instruction b (the first column), we assume a mapping ρ from variables and field names to constraint variables that represent their sizes in the state before executing b. The result of abstracting b w.r.t ρ is the set of constraints $\alpha_\rho(b)$ (second column), and a new mapping ρ' (third column) from variables and field names to constraints variables that represent their sizes in the state after executing b. This is required since we might need to describe a relation like: *the size of x after executing b is equal to its size before the execution plus 1.* Therefore, we need one constraint variable (the size of x) for the state before and one for the state after. ρ' is obtained from ρ by introducing new constraint variables for those variables and fields that might change their value when executing b. This is equivalent to a single static assignment transformation.

	b	$\alpha_\rho(b)$	ρ'
1	$exp\ op\ exp$ $op \in \{>, \geq, <, \leq, =, +, -\}$	$\alpha_\rho(exp)\ op\ \alpha_\rho(exp)$	$\rho'{=}\rho$
2	$exp\ op\ exp$ $op \in \{\neq\}$	$true$	$\rho'{=}\rho$
3	$exp\ op\ exp$ $op \in \{*, /\}$	_	$\rho'{=}\rho$
4	t	$\lvert t[x{\mapsto}\rho(x)] \rvert_{ts}$	$\rho'{=}\rho$
5	`null`	0	$\rho'{=}\rho$
6	x	$\rho(x)$	$\rho'{=}\rho$
7	$this.f$	$\rho(f)$	$\rho'{=}\rho$
8	$x?$	$true$	$\rho'{=}\rho$
9	$match(x,t)$	$\rho(x){=}\alpha_\rho(t)$	$\rho'{=}\rho$
10	$nonmatch(x,t)$	$true$	$\rho'{=}\rho$
11	$g_1 \wedge g_2$	$\alpha_\rho(g_1) \wedge \alpha_\rho(g_2)$	$\rho'{=}\rho$
12	$g_1 \vee g_2$	$true$	$\rho'{=}\rho$
13	`release`	$true$	$\rho'{=}\rho[\bar{f}_C{\mapsto}\rho(\bar{f}_C)']$
14	`await` g	$\alpha_{\rho'}(g)$	$\rho'{=}\rho[\bar{f}_C{\mapsto}\rho(\bar{f}_C)']$
15	$x := y$.`get`	$\rho'(x){=}\rho(y)$	$\rho'{=}\rho[x{\mapsto}\rho(x)']$
16	$x := exp$	$\rho'(x){=}\alpha_\rho(exp)$	$\rho'{=}\rho[x{\mapsto}\rho(x)']$
17	$this.f := exp$	$\rho'(f){=}\alpha_\rho(exp)$	$\rho'{=}\rho[f{\mapsto}\rho(f)']$
18	$x :=$ `new` C	$\rho(x){=}1$	$\rho'{=}\rho[x{\mapsto}\rho(x)']$
19	`call(block,`$q(receiver, \bar{x}, \bar{y})$`)`	$q(\rho(receiver), \rho(\bar{x}{\cdot}\bar{f}_C), \rho'(\bar{y}{\cdot}\bar{f}_C))$	$\rho'{=}\rho[\bar{y}{\cdot}\bar{f}_C{\mapsto}\rho(\bar{y}{\cdot}\bar{f}_C)']$
20	`call(`id$, q(receiver, \bar{x}, \bar{y})$`)` id $\in$ {`method`, `func`}	$q(\rho(receiver), \rho(\bar{x}), \rho'(\bar{y}))$	$\rho'{=}\rho[\bar{y}{\mapsto}\rho(\bar{y})']$

Figure 3.5: Abstract compilation. $\texttt{ABST}(b_{k:i}, \rho){=}\langle\alpha_\rho(b_{k:i}), \rho'\rangle$

Let us describe the abstraction of a few selected representative instructions. In line 16, the instruction $x := exp$ is abstracted into the equality $\rho'(x) = \alpha_\rho(exp)$ where $\alpha_\rho(exp)$ is the size of exp w.r.t. ρ. For example, if $exp \equiv Cons(x, y)$, then line 4 abstracts exp to $1 + \rho(x) + \rho(y)$. The left hand side of the equality uses $\rho'(x)$ and not $\rho(x)$ because it refers to the size of x *after* executing the instruction. Note that the constraint variable to which x is mapped in ρ' is different from that of ρ. The abstraction of `release` at line 13 basically "forgets" sizes of the fields $\bar{f}_C$ of the corresponding class. This is because they might be updated by other methods that take the control when the current task suspends. The abstraction of `await` at line 14 is similar to that of `release`, though we can add to our abstract state the information that the guard g is satisfied upon completion of `await` g. Note that we use ρ' in order to abstract g, since the execution might suspend if g is evaluated to false. When abstracting a call to a block in line 19, the class fields are added as arguments in order to track their values. However, when abstracting calls to methods and functions the fields are not added. For methods, they are not added because their values at call time might not be the same as when the method actually starts to execute. For functions, they are not added since functions are not class members and cannot access fields. Since we use linear constraints only, non-linear arithmetic expressions (line 3) are abstracted to a fresh constraint variable "_" that represents any value. Similarly, non-linear conditions (line 2) are abstracted to *true* in order to lose their effect.

By using the abstraction in Figure 3.5, we now transform a given program P into an abstract program P^α that approximates its behavior w.r.t. the given size measure by applying the abstraction to all instructions in all program rules.

Definition 3.3.3 (abstract compilation)**.** *Given a rule* $r \equiv m(this, \bar{x}, \bar{y}) \leftarrow g, b_1, \ldots, b_n \in P$*, then its abstract compilation is* $r^\alpha \equiv m(this, \bar{I}, \rho_{n+1}(\bar{O})) \leftarrow g^\alpha, b_1^\alpha, \ldots, b_n^\alpha$*, where:*

- $\langle g^\alpha, \rho_1\rangle = \texttt{ABST}(g, \rho_0)$ *and* $\langle b_i^\alpha, \rho_{i+1}\rangle = \texttt{ABST}(b_i, \rho_i)$ *and* ρ_0 *is an identity map over* $vars(r_k) \cup \bar{f}_C$*;*

$reqFile(this, \langle s, fn\rangle, \langle\rangle) \leftarrow$	ρ_0
$f' = 1,$	$\rho_1 = \rho_0[f \mapsto f']$
$ps' = 1,$	$\rho_2 = \rho_1[ps \mapsto ps']$
$i' = 0,$	$\rho_3 = \rho_2[i \mapsto i']$
$incr' = 0$	$\rho_4 = \rho_3[incr \mapsto incr']$
③ $lengthNode(s, \langle fn\rangle, \langle l_1'\rangle),$	$\rho_5 = \rho_4[l_1 \mapsto l_1']$
① $true,$	$\rho_6 = \rho_5[\overline{F} \mapsto \overline{F'}]$
$i'' = l_1',$	$\rho_7 = \rho_6[i \mapsto i'']$
② $loop(this, \langle s, fn, f', ps', i'', incr', l_2, \overline{F}'\rangle,$	$\rho_8 = \rho_7[f \mapsto f'', ps \mapsto ps'', l_2 \mapsto l_2',$
$\langle f'', ps'', l_2', i''', incr'', \overline{F}''\rangle),$	$i \mapsto i''', incr \mapsto incr'', \overline{F} \mapsto \overline{F}'']$
$thisDB' = db'',$	$\rho_9 = \rho_8[thisDB \mapsto thisDB']$
④ $storeFile(thisDB', \langle fn, f''\rangle, \langle\rangle).$	$\rho_{10} = \rho_9$

Figure 3.6: Abstract compilation of instructions of rule *reqFile* (left) and renamings (right).

- $\bar{I} = \bar{x}$ *and* $\bar{O} = \rho_{n+1}(\bar{y})$ *if* m *is a method or a function; and* $\bar{I} = \bar{x}{\cdot}\bar{f}_C$ *and* $\bar{O} = \rho_{n+1}(\bar{y}{\cdot}\bar{f}_C)$ *if* m *is a block,*

The abstract compilation of all rules in P *is denoted by* P^α.

Example 3.3.4. *Figure 3.6 shows the abstract compilation of rule reqFile in Ex. 3.1.1 (*ρ_0 *stands for the identity mapping).* $\overline{F}$ *denotes the sequence of fields db, file, catalog, myNeighbors, admin, size declared in class* Node. *The most relevant points are marked as:* ① *the abstraction of the* `await` *instruction returns true and at this point the information on the fields is lost,* ② *when abstracting the call to the loop block in the first rule, we have added the fields of the class in order to keep track of this information. However, in* ③ *and* ④*, since we are calling methods lengthNode and storeFile, the abstraction has to "forget" this information, i.e., we cannot pass it in the call.*

3.3.1.3 IO Relations.

The constraints in each abstract rule describe the size relations induced by the rule's instructions. In addition, there are some relations that cannot be observed by looking locally at each rule. For example, in *reqFile*, the relation between the output variable l_1' of *lengthNode* and the variables before calling *lengthNode* is not explicitly there, as it depends on the functionality of *lengthNode*. These relations are called *IO (size) relations*, and they describe post-conditions that hold (upon return) between the sizes of the input and output variables of a given rule. These relations are essential for cost analysis, as they also describe how values change when moving from the program point before the call to the one after the call. The abstract program P^α can be used to infer such relations. Since P^α does not contain any concurrency construct that appears in P, techniques developed for sequential programs (see [19]) can be safely applied to the concurrent setting. In what follows, we assume that $\mathcal{I}_P$ is the set of such relations for all procedures. The elements of $\mathcal{I}_P$ are of the form $\langle m(this, \bar{x}, \bar{y}), \psi\rangle$ where ψ is a conjunction of (linear) constraints over the (sizes of) the variables $\bar{x} \cup \bar{y}$. As in size relations of sequential programs, if executing m on input of size $\bar{v}_1$ results in output of size $\bar{v}_2$, then $\bar{x} = \bar{v}_1 \wedge \bar{y} = \bar{v}_2 \models \psi$. In addition, for the concurrent setting, we require that ψ does not restrict the input variables $\bar{x}$, i.e., $\bar{x} = \bar{v} \wedge \psi$ is satisfiable for any valuation $\bar{v}$. The latter condition guarantees that we have IO relations for each non-terminating input since, for such input, ψ is *true*.

Example 3.3.5. *In method* lengthDB*, we can observe that the output of lookup is an input to length. Thus, in order to infer the cost of* lengthDB *(and thus the cost of all methods that invoke it), we need to have IO*

relations for lookup. The IR (left) and abstract compilation (right) of lookup are:

$lookup(this, \langle ms, k\rangle, \langle lp\rangle) \rightarrow$ $\quad match(ms, InsAss(Pair(k, y), z)),$ $\quad lp := y.$	$lookup(this, \langle ms, k\rangle, \langle lp'\rangle) \rightarrow$ $\quad ms' = 2 + k + y + z,$ $\quad lp' = y.$
$lookup(this, \langle ms, k\rangle, \langle lp\rangle) \rightarrow$ $\quad nonmatch(ms, InsAss(Pair(k, y), z)),$ $\quad match(ms, InsAss(w, tm)),$ $\quad \texttt{call}(\texttt{func}, lookup(this, \langle tm, k\rangle, \langle lp\rangle)).$	$lookup(this, \langle ms, k\rangle, \langle lp'\rangle) \rightarrow$ $\quad true,$ $\quad ms' = 1 + w + tm,$ $\quad lookup(this, \langle tm, k\rangle, \langle lp'\rangle).$

Using the techniques of [19] we infer $lookup(\langle this, ms, k\rangle, \langle lp\rangle), \{2 + lp + k \leq ms\}$*, which expresses, for example, that the size of lp is smaller than the size of ms. This piece of information is crucial when inferring the cost of the call to length.*

3.3.2 Cost Relations Based on Cost Centers

After having designed the size abstraction, the next step is to generate a CRS (Cost Relation System, See Chapter 2) which defines the cost of executing each method in the program as a function of its input variables and the initial state of the heap when executing the method. As its main novelty, the CRSs used here include cost centers in order to keep the resource usage assigned to the different components separate. Given a finite set of cost centers $c_0, \ldots, c_n$, where c_0 denotes the cost center of the main method, we assume the existence of a function $\texttt{CC}(o)$ which returns statically a set of possible cost centers of object o at a given program point. This will allow us to instantiate our analysis with different deployment strategies. In particular, it can happen that a group of objects share the processor, as in full ABS, i.e., several objects belong to the same cost center. In the examples, we consider an instance of this model in which all objects of the same class are supposed to share the processor and hence belong to the same cost center, i.e., $\texttt{CC}(o)$ returns the class of o. For this case, CC is computed automatically using class analysis.

Definition 3.3.6 (CRS with cost centers)**.** *Consider a cost model* $\mathcal{M}$*. Given a rule* $r \equiv m(this, \bar{x}, \bar{y}) \leftarrow g, b_1, \ldots, b_n \in P$ *and its abstract compilation* $r^\alpha \equiv m(this, \bar{I}, \bar{O}) \leftarrow g^\alpha, b_1^\alpha, \ldots, b_n^\alpha \in P^\alpha$*, its corresponding cost equation w.r.t.* $\mathcal{M}$ *is* $m(D_0, this, \bar{I}) = e + q_1(D_1, \bar{x}_1) + \ldots + q_m(D_m, \bar{x}_m), \varphi$ *where:*

1. $e = \texttt{c}(D_0) * \mathcal{M}(g) + \Sigma_{i=1}^{n} \texttt{c}(D_0) * \mathcal{M}(b_i)$*;*
2. $\varphi = g^\alpha \wedge \varphi_1 \wedge \cdots \wedge \varphi_n$ *where* $\varphi_i \equiv b_i^\alpha$ *if* b_i *is not a call, and* $\varphi_i = \psi$ *if* $b_i^\alpha \equiv q(receiver, \bar{x}, \bar{y})$ *and* $\langle q(receiver, \bar{x}, \bar{y}), \psi\rangle \in \mathcal{I}_P$*;*
3. *each* $q_i(receiver, \bar{x}_i, \bar{y}_i)$ *in* r_k^α *defines* $q_i(D_i, \bar{x}_i)$ *where* $D_i = D_0$ *if* $receiver \equiv this$*, otherwise* $D_i = \texttt{CC}(receiver)$*;*

The CRS of P, denoted P^{crs}*, is the set of cost equations of its rules.*

The above *CRs* are like standard *RRs* for sequential programs in that: (i) they do not have output arguments, as the cost is a function of the input; (ii) given a rule being analyzed, its cost equation is obtained by applying the cost model $\mathcal{M}$ to each of the instructions in the body (item 1 in Def. 3.3.6); (iii) the size relations are gathered together to define the applicability constraints φ for the equations (item 2); (iv) a call in the program is substituted by a call to its corresponding cost equation (item 3).

The main differences to standard relations are: (a) the equations contain as first parameter a variable D_0 that represents the set of cost centers to which their cost will be attributed. The cost of calls (item 3) is assigned to the cost centers of the object on which the call is performed; (b) the cost expressions we accumulate (item 1) are multiplied by a symbolic expression $\texttt{c}(D)$. Such symbolic expressions allow us to obtain the cost for a particular cost center c_i by setting up $\texttt{c}(D)$ to 1 if $c_i \in D$, and to 0 otherwise. In what follows, given a cost expression e, we let $e|_{D'}$ denote the cost expression that results from replacing $\texttt{c}(D)$ by 1 if $D \cap D' \neq \emptyset$ and 0 otherwise.

$$
\begin{array}{rcl}
\mathit{reqFile}(\mathsf{Node}, s, \mathit{fn}) & = & \mathsf{c}(\mathsf{Node}) * 15 + ①\mathit{lengthNode}(\mathsf{Node}, \mathit{fn}) + \\
 & & \mathit{loop}(\mathsf{Node}, \mathit{fn}, i'', \mathit{size}) + ①\mathit{storeFile}(\mathsf{DB}) \quad \{i'' \geq 0\} \\
\mathit{loop}(\mathsf{Node}, \mathit{fn}, i, \mathit{size}) & = & \mathsf{c}(\mathsf{Node}) * 3 \quad \{i \leq 0\} \\
\mathit{loop}(\mathsf{Node}, \mathit{fn}, i, \mathit{size}) & = & \mathsf{c}(\mathsf{Node}) * 4 + \mathit{if}(\mathsf{Node}, \mathit{fn}, i, \mathit{size}) \quad \{i > 0\} \\
\mathit{if}(\mathsf{Node}, \mathit{fn}, i, \mathit{size}) & = & \mathsf{c}(\mathsf{Node}) * 3 + \mathit{if}^c(\mathsf{Node}, \mathit{fn}, i, \mathit{incr}, \mathit{size}) \ \{i < \mathit{size}, \mathit{incr} = i\} \\
\mathit{if}(\mathsf{Node}, \mathit{fn}, i, \mathit{size}) & = & \mathsf{c}(\mathsf{Node}) * 3 + \mathit{if}^c(\mathsf{Node}, \mathit{fn}, i, \mathit{incr}, \mathit{size}) \ \{i \geq \mathit{size}, \mathit{incr} = \mathit{size}\} \\
\mathit{if}^c(\mathsf{Node}, \mathit{fn}, i, \mathit{incr}, \mathit{size}) & = & \mathsf{c}(\mathsf{Node}) * 10 + ①\mathit{getPacks}(\mathsf{Node}, \mathit{fn}, \mathit{incr}, i) + \\
 & & ①\mathit{app}(\mathsf{Node}, \mathit{ps}, f) + \mathit{loop}(\mathsf{Node}, \mathit{fn}, i', \mathit{size}') \\
 & & \{\ \mathit{ps} + f - 1 = f', i' = i - \mathit{incr}\} \\
\mathit{lengthNode}(\mathsf{Node}, \mathit{fn}) & = & \mathsf{c}(\mathsf{Node}) * 4 + ①\mathit{lengthDB}(\mathsf{DB}, \mathit{db}, \mathit{fn}) \\
\mathit{getPacks}(\mathsf{Node}, \mathit{fn}, \mathit{ps}, n) & = & \mathsf{c}(\mathsf{Node}) * 10 + ①\mathit{getFile}(\mathsf{DB}, \mathit{db}, \mathit{fn}) + \mathit{loop}_1(\mathsf{Node}, \mathit{ps}, f, n) \\
\mathit{loop}_1(\mathsf{Node}, \mathit{ps}, f, n) & = & \mathsf{c}(\mathsf{Node}) * 3 \ \{\mathit{ps} \leq 0\} \\
\mathit{loop}_1(\mathsf{Node}, \mathit{ps}, f, n) & = & \mathsf{c}(\mathsf{Node}) * 13 + \mathit{nth}(\mathsf{Node}, f, k) + \mathit{loop}_1(\mathsf{Node}, \mathit{ps}', f, n) \\
 & & \{\mathit{ps} \geq 1, k = n + \mathit{ps} - 1, \mathit{ps}' = \mathit{ps} - 1\} \\
\mathit{storeFile}(\mathsf{DB}) & = & \mathsf{c}(\mathsf{DB}) * 3 \\
②\mathit{lengthDB}(\mathsf{DB}, \mathit{fn}) & = & \mathsf{c}(\mathsf{DB}) * 2 + \mathit{lookup}(\mathsf{DB}, \mathit{dbf}, \mathit{fn}) + \mathit{length}(\mathsf{DB}, \mathit{lp}) \\
 & & \{\mathit{lp} \geq 1, 3 + \mathit{lp} + \mathit{fn} \leq \mathit{dbf}\} \\
③\mathit{getFile}(\mathsf{DB}, \mathit{fn}) & = & \mathsf{c}(\mathsf{DB}) * 1 + \mathit{lookup}(\mathsf{DB}, \mathit{dbf}, \mathit{fn})
\end{array}
$$

Figure 3.7: Cost Relation System for Selected Methods in classes Node and DB.

A fundamental feature of the generated CRS is that, since the concurrency constructs have been removed, they can be solved to closed-form upper/lower bounds using the techniques in Section 4.4. It should be noted also that, unlike other approaches to cost analysis (e.g., type-based systems [43]), CRS are powerful tools able to infer all kinds of elementary complexity classes, including exponential, polynomial and logarithmic bounds.

Example 3.3.7. *Let us consider the cost model $\mathcal{M}_{ins}$ in Section 3.2.1. Assume that* CC(o) *returns as cost center the class of the object o. Figure 3.7 shows the CRS resulting from applying Def. 3.3.6 to reqFile and to all methods its cost depends upon. For readability, we have removed all arguments and intermediate constraints that do not affect the cost. Moreover, we assume the implicit constraint $x \geq 1$ for any variable of type Term (i.e., the size of a data structure is always positive). The first parameter in the equations is instantiated to the cost center to which the cost will be assigned. When a method call on an object is performed (annotated as ①), the cost center gets instantiated with the actual class. Let us explain the first* 5 *equations, which correspond to the* `reqFile` *method. The first equation defines the cost reqFile in terms of the cost of lengthNode, loop and storeFile. The expression* $\mathsf{c}(\mathsf{Node}) * 15$ *corresponds to the cost of the instructions in* `reqFile` *which are not inside the loop. Note than when calling loop we start with $i'' \geq 0$, since it is the only information we can infer about i''. The second and third equations correspond to the* `while` *loop condition. The first is for the case when $i \leq 0$, in which the loop is not executed, and the second for the case $i > 0$ in which we continue to the equation if in order to accumulate the loop's cost. The equations of if correspond to the* `then` *($i <$ size) and* `else` *($i \leq$ size) branches where incr is assigned to i or size and it continues to if^c. The equation if^c corresponds to the loop's body, where it accumulates the cost of getPacks and app, and recursively calls loop for the next iteration. Note that i is decremented by incr units, and that the value of the field size is lost (we use size$'$) due to the* `await` *. While a CRS can always be set up, the accuracy of the analysis shows up when trying to solve the CRS into a closed-form upper/lower bound. None of the equations in Figure 3.7 is solvable. Indeed, using the solver described in Section 4.4, we can only obtain the following closed-form UBs for the functions:*

$$\begin{array}{rcl} nth(D_0, list, n) & = & \mathsf{c}(D_0) * 5 + \mathsf{nat}(n) * \mathsf{c}(D_0) * 9 \\ length(D_0, list) & = & \mathsf{c}(D_0) * 3 + \mathsf{nat}(\frac{list-1}{2}) * \mathsf{c}(D_0) * 6 \\ lookup(D_0, ms, k) & = & \mathsf{c}(D_0) * 2 + \mathsf{nat}(\frac{ms-1}{2}) * \mathsf{c}(D_0) * 4 \\ app(D_0, l_1, l_2) & = & \mathsf{c}(D_0) * 2 + \mathsf{nat}(\frac{l_1-1}{2}) * \mathsf{c}(D_0) * 5 \end{array}$$

The reason why it is not possible to solve the equations for any method is related to the loss of information in the abstract compilation. Let us focus on lengthDB*. After replacing the cost of* lookup *and* length *by the above UBs in equation* (2)*, we get the equation (and the size relation that appears in Figure 3.7):*

$$lengthDB(\mathsf{DB}, fn) \quad = \quad \mathsf{c}(\mathsf{DB}) * 7 + \mathsf{nat}(\tfrac{dbf-1}{2}) * \mathsf{c}(\mathsf{DB}) * 4 + \mathsf{nat}(\tfrac{l-1}{2}) * \mathsf{c}(\mathsf{DB}) * 6$$

The problem is that we do not have an UB on the size of dbf, since we have lost this information in the abstraction of the call. Therefore, we cannot find the maximal cost of lengthDB. The same happens in getFile and, since the other equations depend on them, none is solvable.

The following theorem states the soundness of our analysis by ensuring that the cost of any trace can be reproduced in the corresponding CRS. Thus, an upper (resp., lower) bound of the CRS is a correct upper (resp., lower) bound on the actual cost.

Theorem 3.3.8 (soundness)**.** *Given a program P, a cost model $\mathcal{M}$, and a set of cost centers D. Then for any trace t, that starts from an initial configuration, there exists $e \in Ans(\mathtt{main}(\{c_0\}))$ such that $e|_D = \mathcal{C}(t, D, \mathcal{M})$.*

3.4 Class Invariants in Cost Analysis

In this section, we propose a generalization of *class invariants* (see, e.g., [52]) which allows to greatly improve the accuracy of cost analysis of concurrent programs. As discussed in Section 3.3, information about shared variables is problematic because between the point in time when a method is asynchronously called and when it is actually executed, and between the moment when a task releases the CPU and it becomes active again, other task(s) may modify the values of shared variables. However, it is often possible to gather some useful information about shared variables, in the form of class invariants, which must hold in any sensible execution of the program. In sequential programs, class invariants have to be established by constructors and must hold on termination of all (public) methods of the class. They can be assumed at (public) method entry but may not hold temporarily at intermediate states not visible outside the object. In our context, we need that such invariants hold on method termination and also that they hold at all release points (if any) of all methods. This way, since tasks can only start or resume execution after a method terminates or another release point is executed, we can assume that such invariants hold on task start and resumption.

Class invariants have been used for program verification, since they greatly simplify the task of providing a formal specification. Then, a formal verification tool can be used for establishing that the formal specification consisting of class invariants and pre and post conditions for methods is satisfied by the program. Interestingly, class invariants might provide fundamental information to resource analysis of concurrent programs, as we will see in the examples below. We now formalize the process of incorporating class invariants during abstract compilation. Given a program P, we use $\mathcal{CI}_P$ to denote the set of class invariants for all classes in P. The elements of $\mathcal{CI}_P$ are pairs of the form $\langle C, \varphi \rangle$ and should be interpreted as φ is the class invariant for class C. As in Section 3.3, we assume that φ is a set of linear constraints that involve the class fields.

Definition 3.4.1 (abstract compilation with class invariants)**.** *Let P be a program, and $r \equiv m(this, \bar{x}, \bar{y}) \leftarrow g, b_1, \ldots, b_n$ a rule in P. Let $m(this, \bar{I}, \bar{O}) \leftarrow \psi$ be the result of abstract compilation of r according to Def. 3.3.3. Given a set of class invariants $\mathcal{CI}_P$ for P, the* abstract compilation *of r w.r.t. $\mathcal{CI}_P$ is $m(this, \bar{I}, \bar{O}) \leftarrow inv, \psi$ and is obtained as follows:*

- *inv=true if m is block or a function, and inv=φ if m is a method in class C and $\langle C, \varphi \rangle \in \mathcal{CI}_P$;*

$$
\begin{array}{l}
\mathit{lengthDB}(\mathsf{DB}, fn) = \mathsf{c}(\mathsf{DB}) * (7 + \mathsf{nat}(\frac{dbf_{max}-1}{2}) * 4 + \mathsf{nat}(\frac{dbf_{max}-fn}{2}-2) * 6) \\
\mathit{getFile}(\mathsf{DB}, fn) = \mathsf{c}(\mathsf{DB}) * (3 + \mathsf{nat}(\frac{dbf_{max}-1}{2}) * 4) \\
\mathit{lengthNode}(\mathsf{Node}, fn) = \mathsf{c}(\mathsf{Node}) * 4 + \mathsf{c}(\mathsf{DB}) * (7 + \mathsf{nat}(\frac{dbf_{max}-1}{2}) * 4 + \mathsf{nat}(\frac{dbf_{max}-fn}{2}-2) * 6) \\
\mathit{getPacks}(\mathsf{Node}, fn, ps, n) = \mathsf{c}(\mathsf{Node}) * 13 + \\
\quad \mathsf{c}(\mathsf{DB}) * (3 + \mathsf{nat}(\frac{dbf_{max}-1}{2}) * 4) + \mathsf{c}(\mathsf{Node}) * \mathsf{nat}(ps) * (18 + \mathsf{nat}(ps+n-1) * 9) \\
\mathit{reqFile}(\mathsf{Node}, s, fn) = \ \textcircled{a}\ \mathsf{c}(\mathsf{Node}) * 15 + \\
\textcircled{b}\quad \mathsf{c}(\mathsf{Node}) * 4 + \mathsf{c}(\mathsf{DB}) * (7 + \mathsf{nat}(\frac{dbf_{max}-1}{2}) * 4 + \mathsf{nat}(\frac{dbf_{max}-fn}{2}-2) * 6) + \\
\textcircled{c}\quad \mathsf{nat}(\frac{dbf_{max}-fn}{2} - 2) * (\\
\textcircled{d}\ 17 * \mathsf{c}(\mathsf{Node}) + \\
\textcircled{e}\quad \left\{ \begin{array}{l} \mathsf{c}(\mathsf{Node}) * 13 + \mathsf{c}(\mathsf{DB}) * (3 + \mathsf{nat}(\frac{dbf_{max}-1}{2}) * 4) + \\ \mathsf{c}(\mathsf{Node}) * \mathsf{nat}(size_{init}) * (18 + \mathsf{nat}(\frac{dbf_{max}-fn}{2} - 3)) * 9 + \end{array} \right. \\
\textcircled{f}\quad\ \mathsf{c}(\mathsf{Node}) * (2 + 5 * \mathsf{nat}(\frac{dbf_{max}-1}{2}))) + \ \textcircled{g}\ \mathsf{c}(\mathsf{DB}) * 3
\end{array}
$$

Figure 3.8: UBs for Selected Methods using Class Invariants.

- *The abstract compilation of all instructions remains the same as in Figure 3.5, except for* `release` *and* `await`*, which are now as follows:*

$13'$	`release`	$\varphi[\bar{f}_C \mapsto \rho'(\bar{f}_C)]$	$\rho' = \rho[\bar{f}_C \mapsto \rho(\bar{f}_C)']$
$14'$	`await` g	$\alpha_\rho(g) \wedge \varphi[\bar{f}_C \mapsto \rho'(\bar{f}_C)]$	$\rho' = \rho[\bar{f}_C \mapsto \rho(\bar{f}_C)']$

It can be observed in the above definition that the invariants are considered in calls to methods and at possible release points ($13'$ and $14'$).

Example 3.4.2. *The following invariants are required to solve the equations in Figure 3.7:*

1. *In class* DB*, we need an invariant* $0 \leq dbf \leq dbf_{max}$*, where* dbf_{max} *is a constant symbol which bounds the value of* dbf*.*
2. *In class* Node*, we need an invariant which establishes that* $size = size_{init}$*, i.e., field* size *is initialized in the constructor and it is never modified.*

By applying Def. 3.4.1 using the first invariant, the equations for getFile and lengthDB are like those annotated as ② *and* ③ *in Figure 3.7 but include the additional constraint* $\{0 \leq dbf \leq dbf_{max}\}$*. From them, we can obtain the UBs in Figure 3.8 for both methods, as well as for* lengthNode *and* getPacks*, whose costs depend on them. The second invariant is essential in order to obtain an UB for* reqFile*. In particular, it is needed in the equation if*^{c} *in Figure 3.7, which corresponds to the cost of the block that contains the* `await` *instruction (and that introduced inaccuracy in the analysis). By applying Def. 3.4.1 using such an invariant, the solver [9] obtains the UB for reqFile in Figure 3.8. Let us explain the different parts of this UB:* ⓐ *is the cost of the instructions of reqFile excluding those of the loop;* ⓑ *is the cost introduced by lengthNode;* ⓒ *is the number of iterations of the loop;* ⓓ − ⓕ *is the cost of each iteration of the loop, where* ⓓ *is the cost of the loop's instructions,* ⓔ *is the cost of calls to getPacks, and* ⓕ *is the cost of calls to app; and finally* ⓖ *is the cost of storeFile. As expected, the number of executed instructions has an asymptotic bound* $O(dbf_{max}^2 * size_{init})$*. The cost on the cost center* Node *is* $O(dbf_{max}^2 * size_{init})$ *while that on* DB *is* $O(dbf_{max}^2)$*. The distribution of the total cost over different cost centers occurs also in the other UBs. For example, in getPacks the cost on the cost center* DB *is* $O(dbf_{max})$ *while on* Node *it is* $O(ps^2)$*.*

Our analyzer adds symbolic bounds (like in invariant 1 above) for all fields and in all classes. The second type of invariants are also automatically generated by syntactically identifying fields which are written only once in the constructor. Importantly, the UB for a method obtained using invariants may be given, in addition to its input parameters, in terms of the symbolic bound variables introduced when expressing such class invariants, as shown above.

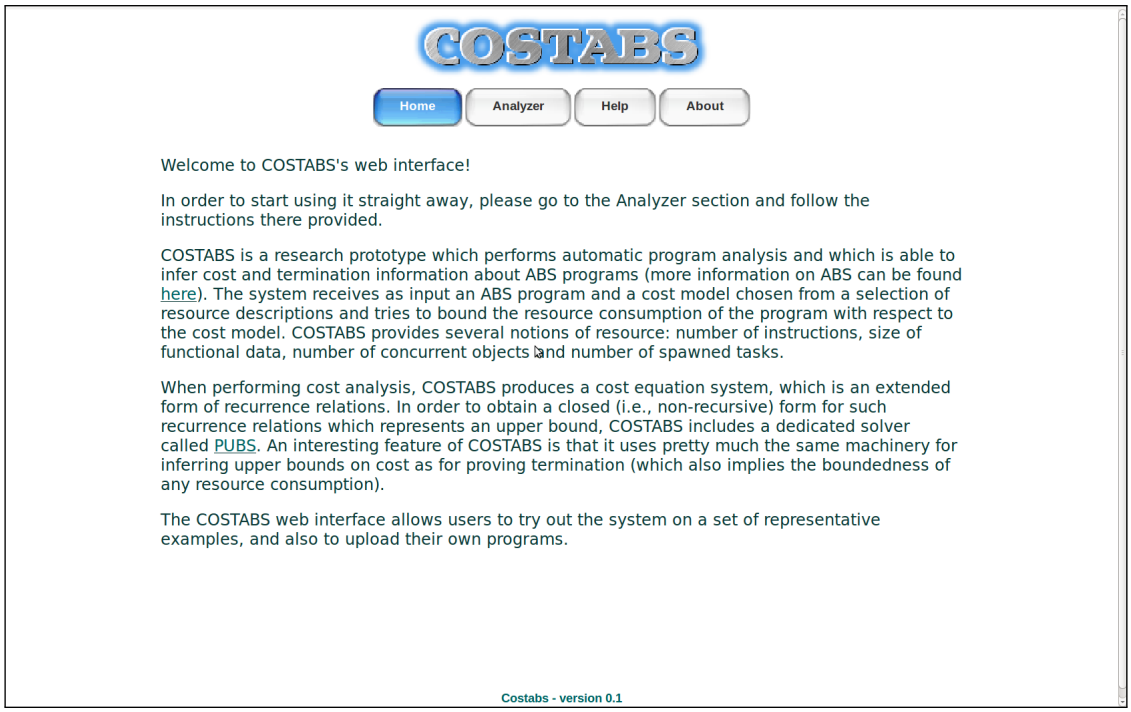


Figure 3.9: COSTABS web interface: Home page

3.5 Implementation and Experimental Evaluation

We have developed an extension for COSTA, named COSTABS, that supports the ABS language. The system uses as external components: the ABS compiler frontend, which parses and compiles ABS source files producing an abstract syntax tree which is then compiled into our rule-based form, and, the Parma Polyhedra Library [16], for manipulating linear constraints. The CRS solver of COSTA is used for finding upper/lower bounds and proving termination. COSTABS works with all the cost models described in Section 3.2.1 except for remote requests, currently under development. The system has two interfaces: a command-line interface and a web interface. Additionally, we are now in the process of developing an *Eclipse* plugin which, in the near future, will be integrated within the ABS Eclipse plugin together with the rest of the tools for ABS.

In the command-line interface the user has to call the "*costabs*" command followed by the name of the file with the ABS source code, the list of methods/functions for which the user wants to get an UB, and, an optional set of parameters, including the cost model, the size abstraction, etc. The COSTABS web interface can be accessed at `http://costa.ls.fi.upm.es/costabs` and allows users to try out the tool without having to install it.

Figure 3.9 shows a screenshot of the home page. This page just shows some information about the tool and its underlying technology. On the top of the page there is a menu with the following four links: *Home*, that shows this home page; *Analyzer*, which goes to the actual COSTABS web-interface; *Help*, that provides some links which can be useful to learn more about the ABS language and the underlying technology of the system; and, *About*, which shows information about the authors of the tool, associated publications, the supporting projects, etc.

Let us now go to *Analyzer*. Figure 3.10 shows the corresponding screenshot. The first step is to provide the ABS source code, which can be either uploaded in the form of a *.abs* file, written in the provided text area, or selected from the set of available examples. This page also allows the user to decide whether the ABS standard library should be loaded together with the source code or not. Let us select the *MyPeerToPeer* file, the "*no*" option, and press "*Continue*". The web interface now brings the user to a second page where: (1) the set of functions and methods for which the user wants to get an UB has to be selected (by clicking in the corresponding check-boxes), and, (2) the system default options can be modified. These options include:

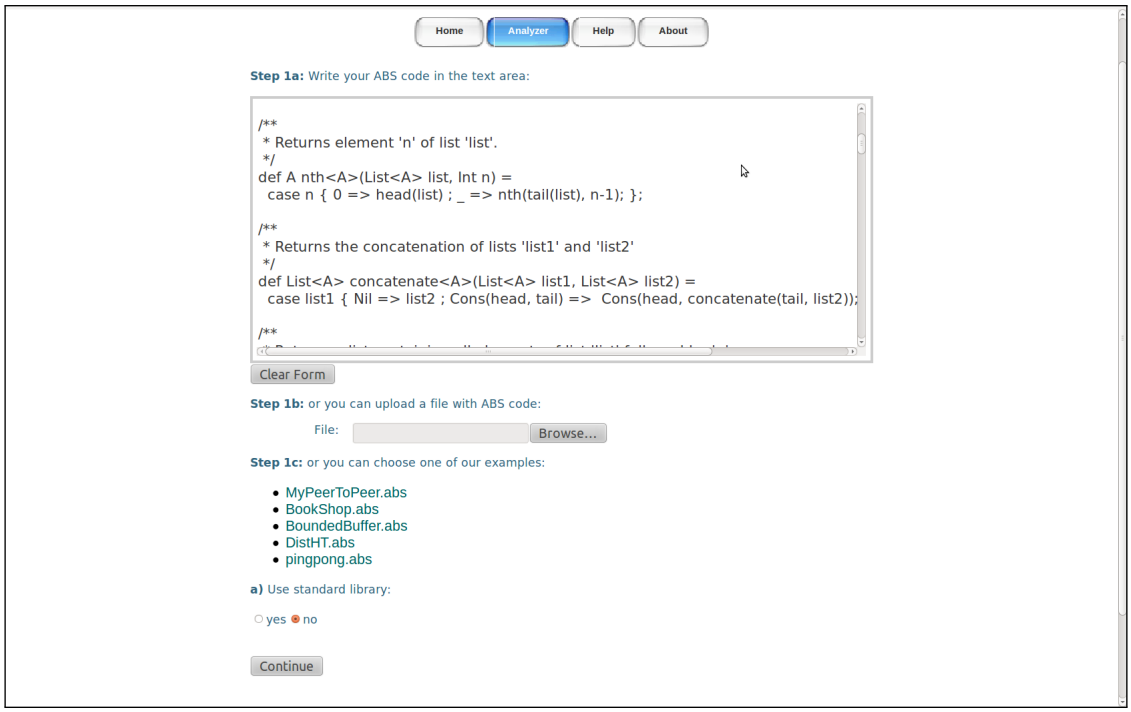


Figure 3.10: COSTABS web interface: Analyzer

Bench	**#F**	**#M**	**#M**$_t$	**#M**$_{ub}$	**#I**$_t$	**#I**$_{ub}$	**T**$_{ir}$	**T**$_{ac}$	**T**$_{io}$	**T**$_{term}$	**T**$_{ub}$
P2P	11	17	14	6	1	2	9	17	69	1494	1886
BookShop	28	11	11	6	0	3	11	18	66	802	1520
BoundedBuffer	4	8	8	4	0	1	1	2	7	70	86
DistHT	7	8	8	1	0	2	3	5	12	82	100
PingPong	0	6	6	6	0	0	3	3	7	22	22

Table 3.1: Statistics about the Analysis Process

the cost model, the possibility of enabling/disabling the cost-centers calculation, the size abstraction used, and the verbosity level. Figure 3.11 shows the corresponding screenshot. Let us select functions *nth* and *concatenate*, and methods *getLength* and *getPacks* (and let the default options). Finally, clicking the *Analyze* button makes the web interface gather all the information provided, and build the corresponding call to the command-line interface. The web interface then jumps directly to another page where the output of the generated command is shown. Figure 3.12 shows the corresponding screenshot. COSTABS shows the times taken by the different steps carried out, and the obtained UBs for the selected functions and methods.

3.5.1 Experimental Evaluation

An experimental evaluation has been carried out using as benchmarks the applications developed in HATS (available at the COSTABS website): PeerToPeer, our running example; BookShop, which implements a web shop client-server application; BBuffer, that implements the classical producer-consumer (or bounded-buffer) problem; DistHT, which implements a distributed hash-table data structure, and PingPong, that implements a simple communication protocol.

Table 3.1 shows some statistics about the analysis process. Each program is analyzed twice. Once for proving termination, i.e., using the $\mathcal{M}_{termin}$ cost model, and once for obtaining an UB on the number of instructions, i.e., the cost model $\mathcal{M}_{ins}$. For each benchmark, columns (**#F**) and (**#M**) show, resp., the number of functions and methods that have been analyzed. Regarding functions, we have proved termination

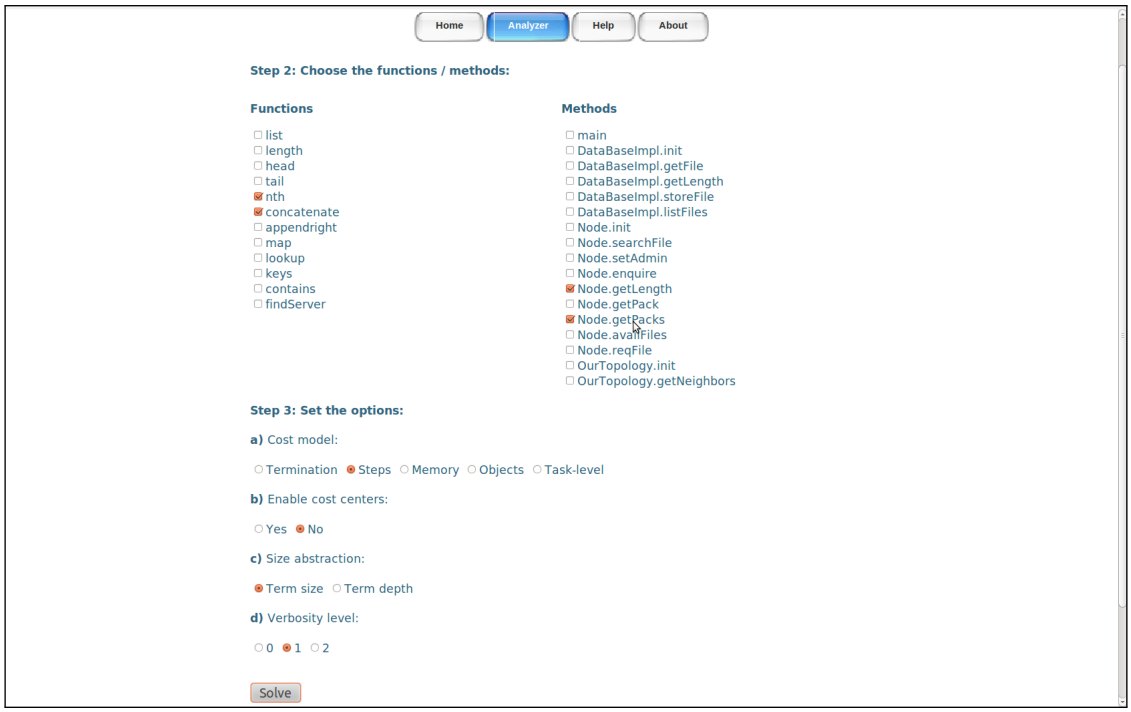


Figure 3.11: COSTABS web interface: Steps 2 and 3

and found UBs for all of them, without using class invariants. Regarding methods, columns $\#\mathbf{M}_t$ and $\#\mathbf{M}_{ub}$ show, resp., the number of methods for which COSTABS proves termination, resp., finds UBs, without using invariants. Column ($\#\mathbf{I}_t$) shows the number of class invariants needed to prove termination. Let us observe that only one invariant is required (the second invariant of Ex. 3.4.2). Similarly, ($\#\mathbf{I}_{ub}$) is the number of class invariants needed to find UBs, they are all similar to the first invariant of Ex. 3.4.2. Note that the whole process is fully automatic: the system generates all invariants and, by using them during abstract compilation, we prove termination and infer UBs for all methods.

The next five columns show the time taken by the different steps of the analysis. Times are in milliseconds, and have been computed as the average of 5 runs. Experiments have been performed on an Intel Core i5 at 3.2GHz with 3.1GB of RAM, running Linux 2.6.32. Columns $\mathbf{T}_{ir}$, $\mathbf{T}_{ac}$, and $\mathbf{T}_{io}$ show, resp., the times taken to build the IR, perform abstract compilation and infer IO relations. These times are the same for $\mathcal{M}_{termin}$ and $\mathcal{M}_{ins}$, since these steps are the same in both cases. The times to infer the invariants are negligible and hence are not shown. As expected, $\mathbf{T}_{io}$ is the most expensive step, as it involves a fixed point computation. Finally, columns $\mathbf{T}_{term}$ and $\mathbf{T}_{ub}$ show, resp., the times taken by [9] to solve the CRS for proving termination and, resp., for obtaining an UB w.r.t. $\mathcal{M}_{ins}$. Since proving termination requires strictly less work than finding UBs, times in $\mathbf{T}_{term}$ are smaller than in $\mathbf{T}_{ub}$.

Overall, we argue that, although our implementation is still prototypical, the experiments show that our approach is promising and UBs for concurrent programs can be, for the first time, inferred in a fully automatic way.

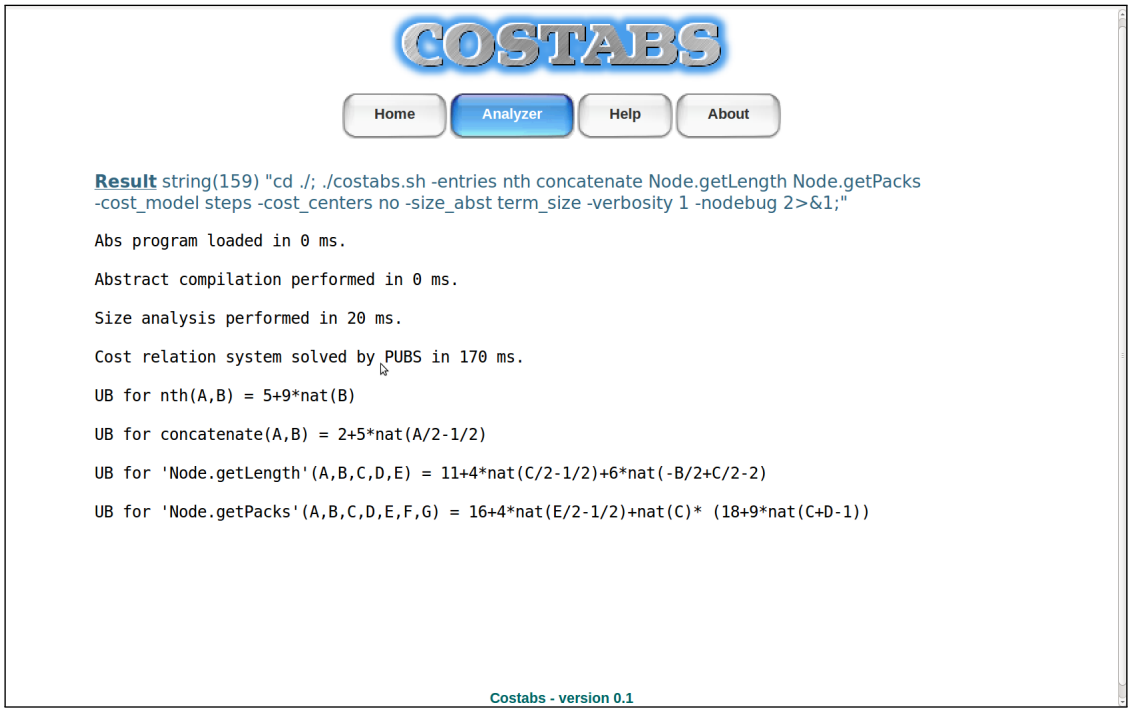


Figure 3.12: COSTABS web interface: Output

Chapter 4

Advanced Issues in Cost Analysis

In this chapter, we present a series of advanced issues in cost analysis which are of general interest in the context of cost analysis of any language, and can be applied in particular to the ABS cost analysis framework presented in the previous section:

- *Component-based approach.* First, we present in Section 4.1 the modular extension of the cost analysis framework where the objective is that different components can be analyzed independently and the results then composed together.
- *Asymptotic UBs.* The results of cost analysis are usually precise, non-asymptotic UBs. For most applications in HATS, their asymptotic variants are of more interest, because they are simpler, allow to ignore implementation details, and are more efficient to obtain. We have designed an automatic transformation of non-asymptotic UBs into asymptotic form, summarized in Section 4.2.
- *Checking against specifications.* The next step is to define a technique, in Section 4.3, to compare the UBs automatically generated by a cost analyzer against resource specifications provided by the user (or the system vendor). This will allow us to verify that the software will safely run on the actual configuration.
- *Accurate upper and lower bounds.* Clearly, improving the accuracy of the results is crucial for all applications of cost analysis (verification, certification, optimization, etc). We will summarize in Section 4.4 a novel technique to infer more precise UBs than [9] and which, furthermore, can be dually applied to infer LBs.

4.1 Component-Based Approach

Typically, cost analysis requires a global analysis of the program in the sense that all the reachable code has to be considered. This is known to work well up to medium-sized programs. However, global analyses can get into scalability problems when trying to analyze larger programs. It is thus required to reach some degree of compositionality which allows decomposing the analysis of large programs into the analysis of smaller parts. This section overviews the compositional approach for cost analysis presented in [58]. We refer to the approach as modular in the sense that it allows reasoning on a method at a time. The approach provides several advantages: first, it allows the analysis of larger programs, since the analyzer does not need to have the complete code of the program nor the intermediate results of the analysis in memory. Second, methods are often called by several other methods. The analysis result of such a shared method can then be reused. The approach presented is flexible with respect to granularity: it can be used in a component-based system at the level of components. A specification can be generated for a component C by analyzing its code, and it can be deployed together with the component and used afterwards for analyzing other components that depend on this one. When analyzing a component-based application that uses C, the code of C does not need to be available at analysis time, since the specification generated can be used

instead. In order to evaluate the effectiveness of the approach, we have extended the COSTA analyzer to be able to perform modular cost analysis and we have applied the improved system to the analysis of the phoneME implementation of the core libraries of JavaME. The results are presented in [58].

4.1.1 Abstract Interpretation Fundamentals

Before describing the modular analysis framework, a brief description to abstract interpretation is included. *Abstract interpretation* [30] is a technique for static program analysis in which execution of the program is simulated on a description (or abstract) domain (D) which is simpler than the actual (or concrete) domain (C). Values in the description domain and sets of values in the actual domain are related via a pair of monotonic mappings $\langle \alpha, \gamma \rangle$: *abstraction* $\alpha : 2^C \to D$ and *concretization* $\gamma : D \to 2^C$ which form a Galois connection, i.e.

$$\forall x \in 2^C : \ \gamma(\alpha(x)) \supseteq x \quad \text{and} \quad \forall \lambda \in D : \ \alpha(\gamma(\lambda)) = \lambda$$

The set of all possible descriptions represents a description domain D which is usually a complete lattice for which all ascending chains are finite. Note that in general $\sqsubseteq$ is induced by $\subseteq$ and α (in such a way that $\forall \lambda, \lambda' \in D : \ \lambda \sqsubseteq \lambda' \Leftrightarrow \gamma(\lambda) \subseteq \gamma(\lambda')$). Similarly, the operations of *least UB* ($\sqcup$) and *greatest LB* ($\sqcap$) mimic those of 2^C in some precise sense that depends on the particular abstract domain. A description $\lambda \in D$ *approximates* a set of concrete values $x \in 2^C$ if $\alpha(x) \sqsubseteq \lambda$. Correctness of abstract interpretation guarantees that the descriptions computed approximate all of the actual values which occur during the execution of the program.

In our cost analysis framework, abstract interpretation is usually performed on an IR, such as the one introduced in Section 3.1.3. Let us introduce some notation. CP and AP stand for descriptions in the abstract domain. The expression $P{:}CP$ denotes a *call pattern*. This consists of a procedure P together with an entry pattern for that procedure. Similarly, $P \mapsto AP$ denotes an answer pattern, though it will be referred to as AP when it is associated to a call pattern $P{:}CP$ for the same procedure. Since a method is represented in the IR as a set of interconnected procedures that start from a single particular procedure, the same notation will be used for methods: $m{:}CP$ denotes a call pattern that corresponds to an invocation to method m (i.e., the entry procedure for method m), and $m \mapsto AP$ denotes the answer pattern obtained after analyzing method m.

Context-sensitive abstract interpretation takes as input a program R and an initial call pattern $P{:}CP$, where P is a procedure and CP is a restriction of the values of the arguments of P expressed as a description in the abstract domain D, and computes a set of triples[1], denoted $analysis(R, P{:}CP) = \{P_1{:}CP_1 \mapsto AP_1, \ldots, P_n{:}CP_n \mapsto AP_n\}$. In each element $P_i{:}CP_i \mapsto AP_i$, P_i is a procedure and CP_i and AP_i are, respectively, the abstract call and answer patterns.

An analysis is said to be *polyvariant* if more than one triple $P{:}CP_1 \mapsto AP_1, \ldots, P{:}CP_n \mapsto AP_n$, $n \geq 0$ with $CP_i \neq CP_j$ for some i, j may be computed for the same procedure P, while a *monovariant* analysis computes (at most) a single triple $P{:}CP \mapsto AP$ for each procedure (with a call pattern CP general enough to cover all possible patterns that appear during the analysis of the program for P).

Although in general context-sensitive, polyvariant analysis algorithms are more precise than those obtained with context-insensitive or monovariant analyses, monovariant algorithms are simpler and have smaller memory requirements. Context-insensitive analysis does not consider call pattern information, and therefore obtains as result of the analysis a set of pairs $\{P_1 \mapsto AP_1, \ldots, P_n \mapsto AP_n\}$, valid for any call pattern.

In order to infer the resource usage, a number of pre-analyses are required. In particular COSTA includes the following abstract interpretation-based analyses:

- *Nullity and sign analysis.* Both of them are context-sensitive and monovariant. The first one detects, at concrete program points, those objects for which it is ensured to be null or non-null. This allows to reduce the IR by removing those branches depending on conditions related to the nullity of an object.

[1] $P{:}CP \mapsto AP$ can be written as $\langle P, CP, AP \rangle$.

Sign analysis is similar, but in its case the information inferred is related to the sign of numeric variables.

- *Size analysis.* It is context-insensitive. This analysis infers, for each rule in the IR, size relations among the input variables to the rule and the variables in all calls in the rule.
- *Heap properties analysis* [39]. It is context-sensitive and polyvariant. This analysis captures the reachability information among program variables and the acyclicity of data structures.

4.1.2 Modular Cost Analysis

In our framework, the cost analysis we perform is in fact a combination of different processes and analyses that receive as input a complete program and eventually produce an upper/lower bound. Our goal now is to obtain a *modular* analysis framework which is able to produce upper/lower bounds by analyzing programs one method at a time. I.e., in order to analyze a method m, we analyze the code of m only and (re-)use the analysis results previously produced for the methods invoked by m.

The communication mechanism used for this work is based on *assertions*, which store the analysis results for those methods which have already been analyzed. Assertions are stored in a file per class basis and they keep information regarding the different analyses performed: nullity, sign, size, heap properties, termination and UBs.

Same as analysis results, assertions are of the form $m{:}Pre \mapsto Post$, where Pre is the *precondition* of the assertion and $Post$ is the *postcondition*. The precondition states for which call pattern the method has been analyzed. It includes information regarding all domains previously mentioned except size, which is context-insensitive. Pre_D (resp., $Post_D$) denotes the information of the precondition (resp., postcondition) related to the analysis domain D. For example, $Pre_{nullity}$ corresponds to the information related to nullity in the precondition Pre. The postcondition of an assertion contains the analysis results for all domains produced after analyzing method m. Furthermore, the assertion also states if an upper/lower bound has been computed for that method.

In addition to assertions inferred by the analysis, our cost analysis has been extended to handle assertions written by the user, namely *assumed* assertions. These assertions are relevant for the cases in which analysis is not able to infer some information of interest that we know is correct. This can happen either because the analyzer is not precise enough or because the code of the method is not available to the analyzer, as happens with *native* methods, i.e., those implemented at low-level and for which no bytecode is available. The user can add assumed assertions with information for any domain. In assumed assertions where only information about bounds is available, abstract interpretation-based analyses take $\top$ as the postcondition for the corresponding methods.

4.1.2.1 Modular Bottom-up Analysis

The analysis of a Java program using the modular analysis framework consists in analyzing each of the methods in the program, and eventually computing an UB for a given call pattern. Analyzing a method separately presents the difficulty that, from the analysis point of view, the code to be analyzed is *incomplete* in the sense that the code for methods invoked is not available. More precisely, during analysis of a method m there may be calls $m'{:}CP$ and the code for m' is not available. Following the terminology in [36], we refer to determining the value of AP to be used for $m'{:}CP \mapsto AP$ as the *answer patterns problem*.

Several analysis domains existing in our cost analysis framework are context-sensitive, and all of them, except heap properties analysis, are monovariant. For simplicity, the modular analysis framework we present is monovariant as well. That means that at most one assertion $m{:}Pre \mapsto Post$ is stored for each method m. If there is an analysis result for m', $m'{:}Pre \mapsto Post$, such that CP is applicable, that is, $CP \sqsubseteq Pre_D$ in the domain D of interest, then $Post_D$ can be used as answer pattern for the call to method m' in m.

For applying this schema, it is necessary that all methods invoked by m have been analyzed already when analyzing method m. Therefore, the analysis must perform a bottom-up traversal of the call graph of

the program. In order to obtain analysis information for m' which is applicable during the analysis of m, it is necessary to use a call pattern for m' in its precondition such that it is equal or more general than the pattern actually inferred during the analysis of m. We refer to this as the *call patterns problem.*

Solving the *call* and *answer patterns problems.* A possibility for solving the *call patterns problem* would be to make the modular analysis framework polyvariant: store all possible call patterns to methods in the program and then analyze those methods for each call pattern. This approach has two main disadvantages: on the one hand, it is rather complex and inefficient, because all call patterns are stored and every method must be analyzed for all call patterns that appear in the program. On the other hand, it requires performing a fixpoint computation through the methods in the program instead of a single traversal of the call graph, since different call patterns for a method may generate new call patterns for other methods. Another alternative is a context-insensitive analysis. All methods are analyzed using $\top$ as call pattern for all domains. In this approach, all assertions are therefore applicable, although in a number of cases $\top$ is too general as call pattern for some domains, and the information obtained is too imprecise.

The solution used in COSTA tries to find a balance between both approaches. A monovariant modular analysis framework simplifies a great deal the behavior of the modular analysis, since a single traversal of the call graph is required. In contrast, it is context-sensitive: instead of $\top$, a default call pattern is used, and the result of the analysis is obtained based on this pattern. This framework uses different values as call patterns, depending on the particular analysis being performed. The default call pattern for nullity and sign is $\top$. For the heap properties analysis, in cyclicity it is the pattern that indicates that no argument of the method is cyclic. For variable sharing, it is the one that states that no arguments share.

The default call patterns used for analyzing methods are general enough to be applicable to most invocations used in the libraries and in user programs, solving the *call patterns problem.* However, there can be cases in which the call pattern of an invocation from other method is not included in the default pattern, i. e., $CP \not\sqsubseteq Pre_D$. If the code of the invoked method is available, we will reanalyze it with respect to $CP \sqcup Pre_D$, even though it has been analyzed before for the default pattern. If the code is not available, $\top$ is used as answer pattern.

A potential disadvantage of this approach is that all methods are analyzed with respect to a default call pattern, instead of the specific call pattern produced by the analysis. This means that the analyses could produce more precise results when applied non modularly, even though they are monovariant, and it represents a possible loss of precision in the modular analysis framework.

Cycles in the call graph. Analyzing just a method at a time and (re-)using analysis information while performing a bottom-up traversal of the call graph only works under the assumption that there are no cyclic dependencies among methods. In the case where there are strongly connected components (SCCs for short) consisting of more than one method, we analyze all the methods in the corresponding SCC simultaneously. This presents no technical difficulties, since we can analyze multiple methods at the same time and is the only situation in which multiple methods need to be analyzed together in order to obtain correct results. Therefore, we perform a SCC study first to decide whether there are sets of methods which need to be handled as a unit.

4.2 Asymptotic UBs

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.* 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* UBs, 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 LBs ("Big Omega" notation) and asymptotically tight estimates, when the asymptotic upper and LBs 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 main techniques presented in [5] are applicable to obtain asymptotic versions of the cost functions produced by any cost analysis, including lower, upper and average cost analyses. The main contributions are:

1. A new notion of *asymptotic complexity* to cover the analysis of realistic programs whose limiting behavior is determined by the limiting behavior of its loops.

2. A novel transformation from *non-asymptotic cost functions* into asymptotic form. After some syntactic simplifications, the transformation detects and eliminates subterms which are *asymptotically subsumed* by others while preserving the complexity order.

4.2.1 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 4.2.1 (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 UBs and the big Θ 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 ∞ when the input is ∞. This assumption does not necessarily hold when the UBs are obtained from realistic programs. For instance, consider the loop in Figure 2.1. Clearly, the execution cost of the program increases by increasing the number of iterations of the loop, i.e., $n - i$. Therefore, in order to observe the limiting behavior of the program we should study the case when $\mathsf{nat}(n - i)$ goes to ∞, i.e., when, for example, n goes to ∞ and i stays constant, but not when both n and i go to ∞. In order to capture this asymptotic behavior, 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 4.2.2 (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 4.2.3. *Given the following cost function:*

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

Its nat*-free representation is* $5{+}7{*}A{*}\mathsf{max}(\{100{*}A^2{*}B^4, 11{*}3^B{*}A^2\}){+}2{*}\log(A){*}2^B{*}\log(B){*}C$*, where A corresponds to* $\mathsf{nat}(x)$*, B to* $\mathsf{nat}(y)$ *and C to* $\mathsf{nat}(x{-}y)$*.*

Definition 4.2.4. *Given two cost expressions e_1, e_2 and its* 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 Definition 4.2.1 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 nat-free expressions. Note that by obtaining their 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 a set as *context constraint*. For example, the cost expression of Example 4.2.3 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 φ associated to the cost expression can sometimes be used to check whether some 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 nat-expressions (represented by their corresponding nat-variables A and B), we say that $\varphi{\models}A \succeq B$ if A asymptotically subsumes B when both go to ∞.

4.2.2 Asymptotic Orders of Cost Expressions

As it is well-known, by using Θ 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 goals stated in the beginning of this subsection 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 Example 4.2.3, 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 parentheses, 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 possibly 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 a non-redundant, asymptotically equivalent one and, in particular, it can be applied to the UBs obtained in Chapter 3. 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 nat-expressions, in order to obtain an asymptotic order of e, as defined in Definition 4.2.4.

4.2.2.1 Syntactic Simplification of Cost Expressions

First, we perform some syntactic simplifications to enable the subsequent steps of the transformation. Given a nat-free cost expression $\tilde{e}$, we describe how to simplify it and obtain another 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 4.2.5. *Given a* 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 4.2.6. *Applying the above transformation on the* nat*-free cost expression of Example 4.2.3 results in: $\tau(\tilde{e}) = A*(A^2*B^4+3^B*A^2)+\log(A)*2^B*\log(B)*C$.*

Once the τ 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 the term *basic* 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 4.2.7 (normalized nat-free cost expression)**.** *A normalized* 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* 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 multiset of its elements (i.e., basic nat-free cost expressions). We use the letter M to denote such a multiset. Also, since $M_1 + M_2$ and $M_2 + M_1$ are equal, it is convenient to view the sum as the multiset of its elements, i.e., products (represented as multisets). Therefore, a normalized cost expression is a multiset of multisets of basic cost expressions. In order to normalize a nat-free cost expression $\tau(\tilde{e})$ we apply repeatedly the distributive property of multiplication over addition in order to get rid of all parenthesis in the expression.

Example 4.2.8. *The normalized expression for $\tau(\tilde{e})$ of Example 4.2.6 is:*

$$A^3*B^4+2^{\log(3)*B}*A^3+ \log(A)*2^B * \log(B) * C$$

and its multiset representation is $\{\{A^3, B^4\}, \{2^{\log(3)*B}, A^3\}, \{\log(A), 2^B, \log(B), C\}\}$.

4.2.2.2 Asymptotic Subsumption

Given a normalized nat-free cost expression $\tilde{e} = \{M_1, \ldots, M_n\}$ and a context constraint φ, 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 remainder of this section defines a decision procedure for deciding whether $M_i \in O(M_j)$. 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 expressions b with a single 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 4.2.9 (asymptotic subsumption)**.** *Let b be a basic* nat*-free cost expression, $M = \{b_1, \cdots, b_m\}$ a product, φ 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 $1 \leq i \leq m$ 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) $r \geq \Sigma_{i=1}^{m} \mathsf{pow}(b_i)$ *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* $\log(A_i)$ *and* $r \geq \Sigma_{i=1}^{m} \mathsf{deg}(b_i)$; *or*
   (b) *there is at least one* b_i *of the form* $\log(A_i)$ *and* $r \geq 1 + \Sigma_{i=1}^{m} \mathsf{deg}(b_i)$
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 UB. Note that an exponential would introduce an ∞ degree. In 2b, we express that there can be many logarithms and still the maximal polynomial is the UB, e.g., A^2 subsumes $A * \log(A) * \log(A) * \ldots * \log(A)$. In 3, a logarithm only subsumes another logarithm.

Example 4.2.10. *Let* $b = A^3$, $M = \{\log(A), \log(B), C\}$, *where* A, B *and* C *correspond to* $\mathsf{nat}(x)$, $\mathsf{nat}(y)$ *and* $\mathsf{nat}(x-y)$ *respectively. Let us assume that the context constraint is* $\varphi = \{x \geq y, x \geq 0, y \geq 0\}$. M *is asymptotically subsumed by* b *since* $\varphi \models (A \succeq B) \wedge (A \succeq C)$, *and condition 2b in Lemma 4.2.9 holds.*

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 nat-free cost expression in M_1. Note that M_1 can contain additional basic nat-free cost expressions which are not needed for subsuming M_2.

Example 4.2.11. *Let* $M_1 = \{2^{\log(3)*B}, A^3\}$ *and* $M_2 = \{\log(A), 2^B, \log(B), C\}$, *with the context constraint* φ *as defined in Example 4.2.10. 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,* $M_2 \in O(M_1)$. *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* $M_2' \in O(M_1)$.

4.3 Checking Against Specifications

In all applications of resource analysis, such as resource usage verification, program synthesis and optimization, etc., it is necessary to compare cost functions. This allows choosing an implementation with smaller cost or to guarantee that the given resource usage bounds are preserved. Essentially, given a method m, a cost function f_m and a context (set of linear constraints) ϕ_m which impose size restrictions (e.g., that a variable in m is larger than a certain value or that the size of an array is non zero, etc.), we aim at comparing it with another cost function bound $\mathtt{b}$ and corresponding size constraints $\phi_{\mathtt{b}}$. Such cost functions can be automatically inferred by a resource analyzer (e.g., if we want to choose between 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 non-negative, e.g., $\mathtt{b} - f_m \geq 0$ in the context $\phi_{\mathtt{b}} \wedge \phi_m$. This is undecidable and also non-trivial to approximate, as cost functions involve non-linear subexpressions (e.g., exponential, polynomial and logarithmic subexpressions) and they can contain multiple variables possibly related by means of constraints in $\phi_{\mathtt{b}}$ and ϕ_m. In order to develop a practical approach to the comparison of cost functions, we take advantage of the form that cost functions originating from the analysis of programs have and of the fact that they evaluate to non-negative values. Essentially, the technique presented in [6] consists in the following steps:

1. Normalizing cost functions to a form which make them amenable to be syntactically compared, e.g., this step includes transforming them to sums of products of basic cost expressions.

2. Defining a series of comparison rules for basic cost expressions and their (approximated) differences, which then allow us to compare two products.

3. Providing sufficient conditions for comparing two sums of products by relying on the product comparison, and enhancing it with a *composite* comparison schema which establishes when a product is larger than a sum of products.

We have implemented our technique in the COSTA system [11]. Our experimental results demonstrate that our approach works well in practice, it can deal with cost functions obtained from realistic programs and verifies user-provided UBs efficiently.

4.3.1 Context Constraints

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

Example 4.3.1. *Figure 4.1 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* COSTA*, 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 work, we will skip explanations about the constants in the UB function and refer to [10] for details.*

Given the context constraint $\{n > 0\}$*,* COSTA *outputs the UB cost function for method* m *which is shown at the bottom of the figure. Since* m *contains two recursive calls, the complexity is exponential on* n*, namely we have a factor* $2^{\mathsf{nat}(n)}$*. At each recursive call, the method* f *is invoked and its cost (plus a constant value) is multiplied by* $2^{\mathsf{nat}(n)}$*. In the code of* f*, we can observe that the* while *loop has a logarithmic complexity because the loop counter is divided by 2 at each iteration. This cost is accumulated with the cost of the second nested loop, which has a quadratic complexity. Finally, the cost introduced by the base cases of* m *is exponential since, due to the double recursion, there is an exponential number of computations which correspond to base cases. Each such computation requires a maximum of 3 instructions.*

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

4.3.2 Comparison of Cost Functions

In this section, we state the problem of comparing two cost functions represented as cost expressions. As we have seen in Example 4.3.1, a cost function $\langle e, \varphi \rangle$ for a method m is a single cost expression which approximates the cost of any possible execution of m which is consistent with the context constraints φ. This can be done by means of nat subexpressions which encapsulate conditions on the input data sizes in a single cost expression. Besides, cost functions often contain max subexpressions, e.g., $\langle \mathsf{max}(\{\mathsf{nat}(x) *$

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

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

UB Cost Function

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

Figure 4.1: UB obtained by COSTA on the number of executed bytecode instructions.

$\mathsf{nat}(z), \mathsf{nat}(y) * \mathsf{nat}(z)\}), \mathit{true}\rangle$ which represent the cost of disjunctive branches in the program (e.g., the first subexpression might correspond to the cost of a then-branch and the second one the cost of the else-branch of a conditional statement).

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

Definition 4.3.2 (max-free operator)**.** *Let $\langle e, \varphi\rangle$ be a cost function. We define the* max*-free operator $\tau_{\mathsf{max}} : 2^{\mathcal{CF}} \mapsto 2^{\mathcal{CF}}$ as follows: $\tau_{\mathsf{max}}(M) = (M - \{\langle e, \varphi\rangle\}) \cup \{\langle e[\mathsf{max}(S) \mapsto e'], \varphi\rangle, \langle e[\mathsf{max}(S) \mapsto \mathsf{max}(S'), \varphi\rangle\}$, where $\langle e, \varphi\rangle \in M$ contains a subexpression of the form $\mathsf{max}(S)$, $e' \in S$ and $S' = S - \{e'\}$.*

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

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

Definition 4.3.3 (nat-free operator)**.** *Let $\langle e, \varphi\rangle$ be a* max*-free cost function. We define the* nat*-free operator $\tau_{\mathsf{nat}} : 2^{\mathcal{CF}} \mapsto 2^{\mathcal{CF}}$ as follows: $\tau_{\mathsf{nat}}(M) = (M - \{\langle e, \varphi\rangle\}) \cup \{\langle e_i, \varphi_i\rangle \mid \varphi \wedge \varphi_i$ is satisfiable $, 1 \leq i \leq 2\}$, where $\langle e, \varphi\rangle \in M$ contains a subexpression $\mathsf{nat}(l)$, $\varphi_1 = \varphi \cup \{A = l, A > 0\}$, $\varphi_2 = \varphi \cup \{l \leq 0\}$, with A a fresh variable, and $e_1 = e[\mathsf{nat}(l) \mapsto A]$, $e_2 = e[\mathsf{nat}(l) \mapsto 0]$.*

In contrast to the max elimination transformation, the elimination of nat subexpressions modifies the set of linear constraints by adding the new assignments of fresh variables to linear expressions and the fact that the subexpression is greater than zero or when it becomes zero. The above operator τ_{nat} is applied

iteratively until there are no new terms to transform. The result of this operation is a nat-free cost function, denoted by $fp_{\mathsf{nat}}(M)$. For instance, for the cost function $\langle \mathsf{nat}(x) * \mathsf{nat}(z-1), \{x > 0\}\rangle$, fp_{nat} returns the set composed of the following nat-free cost functions:

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

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

Example 4.3.4. *Let us consider the cost function in Example 4.3.1. Since this cost function contains the context constraint $n > 0$, the subexpressions $\mathsf{nat}(n)$ and $\mathsf{nat}(2 * n-1)$ are always positive. By assuming that fp_{nat} replaces $\mathsf{nat}(n)$ by A and $\mathsf{nat}(2 * n-1)$ by B, only those linear constraints containing $\varphi = \{n > 0, A = n, A > 0, B = 2 * n-1, B > 0\}$ are satisfiable (the remaining cases are hence not considered). We obtain the following set of flat functions:*

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

In order to compare cost functions, we start by comparing two flat cost functions in Definition 4.3.5 below. Then, in Definition 4.3.6 we compare a flat function against a general, i.e., non-flat, one. Finally, Definition 4.3.7 allows comparing two general functions.

Definition 4.3.5 (smaller flat cost function in context)**.** *Given two flat cost functions $\langle e_1, \varphi_1\rangle$ and $\langle e_2, \varphi_2\rangle$, we say that $\langle e_1, \varphi_1\rangle$ is smaller than or equal to $\langle e_2, \varphi_2\rangle$ in the context of φ_2, written $\langle e_1, \varphi_1\rangle \trianglelefteq \langle e_2, \varphi_2\rangle$, if for all assignments σ such that $\sigma \models \varphi_1 \cup \varphi_2$ it holds that $\sigma(e_1) \leq \sigma(e_2)$.*

Observe that the assignments in the above definition must satisfy the conjunction of the constraints in φ_1 and in φ_2. Hence, it discards the values for which the constraints become incompatible. An important point is that Definition 4.3.5 allows comparing pairs of flat functions. However, the result of such a comparison is weak in the sense that the comparison is only valid in the context of φ_2. In order to determine that a flat function is smaller than a general function for any context we need to introduce Definition 4.3.6 below.

Definition 4.3.6 (smaller flat cost function)**.** *Given a flat cost function $\langle e_1, \varphi_1\rangle$ and a (possibly non-flat) cost function $\langle e_2, \varphi_2\rangle$, we say that $\langle e_1, \varphi_1\rangle$ is smaller than or equal to $\langle e_2, \varphi_2\rangle$, written $\langle e_1, \varphi_1\rangle \preceq \langle e_2, \varphi_2\rangle$, if $\varphi_1 \models \varphi_2$ and for all $\langle e_i, \varphi_i\rangle \in \tau(\langle e_2, \varphi_2\rangle)$ it holds that $\langle e_1, \varphi_1\rangle \trianglelefteq \langle e_i, \varphi_i\rangle$.*

Note that Definition 4.3.6 above is only valid when the context constraint φ_2 is more general, i.e., less restrictive than φ_1. This is required because in order to prove that a function is smaller than another one it must be so for all assignments which are satisfiable according to φ_1. If the context constraint φ_2 is more restrictive than φ_1 then there are valid input values for $\langle e_1, \varphi_1\rangle$ which are undefined for $\langle e_2, \varphi_2\rangle$. For example, if we want to check whether the flat cost function (1) in Example 4.3.4 is smaller than another f which has the context constraint $\{n > 4\}$, the comparison will fail. This is because function f is undefined for the input values $0 < n \leq 4$. This condition is also required in Definition 4.3.7 below, which can be used on two general cost functions.

Definition 4.3.7 (smaller cost function)**.** *Consider two cost functions $\langle e_1, \varphi_1\rangle$ and $\langle e_2, \varphi_2\rangle$ such that $\varphi_1 \models \varphi_2$. We say that $\langle e_1, \varphi_1\rangle$ is smaller than or equal to $\langle e_2, \varphi_2\rangle$ iff for all $\langle e'_1, \varphi'_1\rangle \in \tau(\langle e_1, \varphi_1\rangle)$ it holds that $\langle e'_1, \varphi'_1\rangle \preceq \langle e_2, \varphi_2\rangle$.*

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

4.3.3 Inclusion of Cost Functions

It is clearly not possible to try all assignments of input variables in order to prove that the comparison holds as required by Definition 4.3.5 (and transitively by Definitions 4.3.6 and 4.3.7). In this section, we aim at defining a practical technique to syntactically check that one flat function is smaller or equal than another one for all valid assignments, i.e., the relation $\trianglelefteq$ of Definition 4.3.5. The whole approach is defined over flat cost functions since from it one can use Definitions 4.3.6 and 4.3.7 to apply our techniques on two general functions.

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

In the sequel, we use the term *basic cost expression* to refer to expressions of the form n, $\log_a(A+1)$, A^n, a^l. Furthermore, we use the letter b, possibly subscripted, to refer to such cost expressions. *Normalized cost expressions* are defined similarly as done in Section 4.2.2.1. The following example recalls this notion and introduces normalized cost expressions used in the rest of the section.

Example 4.3.8. *Let us consider the cost functions in Example 4.3.4. Normalization results in the following cost functions:*

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

$(2)_n$ $\langle 34 * 2^A + 8 * \log_2(1+B) * 2^A, \varphi_2 = \{A = n, A > 0, B = 2 * n - 1, B > 0, a \leq 0, D = b, D > 0\}\rangle$

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

$(4)_n$ $\langle 34 * 2^A + 8 * \log_2(1+B) * 2^A, \varphi_4 = \{A = n, A > 0, B = 2 * n - 1, B > 0, a \leq 0, b \leq 0\}\rangle$

As a set representation we obtain:

$(1)_s$ $\langle\{\{34, 2^A\}, \{8, \log_2(1+B), 2^A\}, \{10, C, 2^A\}, \{6, C, D, 2^A\}\}, \varphi_1\rangle$
$(2)_s$ $\langle\{\{34, 2^A\}, \{8, \log_2(1+B), 2^A\}\}, \varphi_2\rangle$
$(3)_s$ $\langle\{\{34, 2^A\}, \{8, \log_2(1+B), 2^A\}, \{10, C, 2^A\}\}, \varphi_3\rangle$
$(4)_s$ $\langle\{\{34, 2^A\}, \{8, \log_2(1+B), 2^A\}\}, \varphi_4\rangle$

4.3.3.1 Product Comparison

We start by providing sufficient conditions which allow proving the $\trianglelefteq$ relation on the basic cost expressions that will be used later to compare products of basic cost expressions. Given two basic cost expressions e_1 and e_2, the third column in Table 4.1 specifies sufficient, linear conditions under which e_1 is smaller or equal than e_2 in the context of φ (denoted as $e_1 \leq_\varphi e_2$). Since the conditions under which $\leq_\varphi$ holds are over linear expressions, we can rely on existing linear constraint solving techniques to automatically prove them. Let us explain some of entries in the table (recall that n stands for a non-negative integer and l for a linear expression). E.g., verifying that $A^n \leq m^l$ is equivalent to verifying $\log_m(A^n) \leq \log_m(m^l)$, which in turn is

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

Table 4.1: Comparison of basic expressions $e_1 \leq_\varphi e_2$

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

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

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

Definition 4.3.9 (product comparison operator)**.** *Given $\langle \mathcal{M}_1, \varphi_1 \rangle, \langle \mathcal{M}_2, \varphi_2 \rangle$ in $\mathcal{P}_b$, we define the product comparison operator $\tau_* : (\mathcal{P}_b, \mathcal{P}_b) \mapsto (\mathcal{P}_b, \mathcal{P}_b)$ as follows: $\tau_*(\mathcal{M}_1, \mathcal{M}_2) = (\mathcal{M}_1 - \{e_1\}, \mathcal{M}_2 - \{e_2\} \cup \{\mathsf{adiff}(e_1, e_2)\})$ where $e_1 \in \mathcal{M}_1$, $e_2 \in \mathcal{M}_2$, and $e_1 \leq_{\varphi_1 \wedge \varphi_2} e_2$.*

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

Example 4.3.10. *Let us consider the product $\{8, \log_2(1+B), 2^A\}$ which is part of $(1)_s$ in Example 4.3.8. We want to prove that this product is smaller or equal than the following one $\{7, 2^{3*B}\}$ in the context $\varphi = \{A \leq B-1, B \geq 10\}$. This can be done by applying the τ_* operator three times. In the first iteration, since we know by Table 4.1 that $\log_2(1+B) \leq_\varphi 2^{3*B}$ and the adiff is 2^{2*B-1}, we obtain the new sets $\{8, 2^A\}$ and $\{7, 2^{2*B-1}\}$. In the second iteration, we can prove that $2^A \leq_\varphi 2^{2*B-1}$, and add as adiff $2^{2*B-A-1}$. Finally, it remains to be checked that $8 \leq_\varphi 2^{2*B-A-1}$. This problem is reduced to checking that $\varphi \models 8 \leq 2*B-A-1$, which is trivially true.*

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

Lemma 4.3.11. *Given* $\langle \mathcal{M}_1, \varphi_1 \rangle, \langle \mathcal{M}_2, \varphi_2 \rangle$ *where* $\mathcal{M}_1, \mathcal{M}_2 \in \mathcal{P}_b$ *and for all* $e \in \mathcal{M}_1$ *it holds that* $\varphi_1 \models e \geq 1$. *If* $fp_*(\mathcal{M}_1, \mathcal{M}_2) = (\emptyset, \_)$ *then* $\langle \mathcal{M}_1, \varphi_1 \rangle \trianglelefteq \langle \mathcal{M}_2, \varphi_2 \rangle$.

Note that the above operator is non-deterministic due to the (non-deterministic) choice of e_1 and e_2 in Definition 4.3.9. Thus, the computation of $fp_*(\mathcal{M}_1, \mathcal{M}_2)$ might not lead directly to $(\emptyset, \_)$. In this case we can backtrack in order to explore other choices and, in the limit, all of them can be explored until we find one for which the comparison succeeds.

4.3.3.2 Comparison of Sums of Products

We now aim at comparing two sums of products by relying on the product comparison of Section 4.3.3.1. As for the case of basic cost expressions, we are interested in having a notion of approximated adiff when comparing products. The idea is that when we want to prove $k_1 * A \leq k_2 * B$ and $A \leq B$ and k_1 and k_2 are constant factors, we can leave as approximated difference of the product comparison the product $(k_2 - k_1) * B$, provided $k_2 - k_1$ is greater or equal than zero. As notation, given a product $\mathcal{M}$, we use $\mathsf{constant}(\mathcal{M})$ to denote the constant factor in $\mathcal{M}$, which is 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 4.3.12 (sum comparison operator)**.** *Given* $\langle \mathcal{S}_1, \varphi_1 \rangle$ *and* $\langle \mathcal{S}_2, \varphi_2 \rangle$, *where* $\mathcal{S}_1, \mathcal{S}_2 \in \mathcal{P}_\mathcal{M}$, *we define the* sum comparison operator $\tau_+ : (\mathcal{P}_\mathcal{M}, \mathcal{P}_\mathcal{M}) \mapsto (\mathcal{P}_\mathcal{M}, \mathcal{P}_\mathcal{M})$ *as follows:* $\tau_+(\mathcal{S}_1, \mathcal{S}_2) = (\mathcal{S}_1 - \{\mathcal{M}_1\}, (\mathcal{S}_2 - \{\mathcal{M}_2\}) \cup \mathcal{A})$ *iff* $fp_*(\mathcal{M}_1, \mathcal{M}_2) = (\emptyset, \_)$ *where:*

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

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

Example 4.3.13. *Let us consider the sum of products* $(3)_s$ *in Example 4.3.8 together with* $\mathcal{S} = \{\{50, C, 2^B\}, \{9, D^2, 2^B\}\}$ *and the context constraint* $\varphi = \{1 + B \leq D\}$. *We can prove that* $(3)_s \trianglelefteq \mathcal{S}$ *by applying* τ_+ *three times as follows:*

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

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

Theorem 4.3.14. *Let* $\langle \mathcal{S}_1, \varphi_1 \rangle$, $\langle \mathcal{S}_2, \varphi_2 \rangle$ *be two sums of products such that for all* $\mathcal{M} \in \mathcal{S}_1$, $e \in \mathcal{M}$ *it holds that* $\varphi_1 \models e \geq 1$. *If* $fp_+(\mathcal{S}_1, \mathcal{S}_2) = (\emptyset, \_)$ *then* $\langle \mathcal{S}_1, \varphi_1 \rangle \trianglelefteq \langle \mathcal{S}_2, \varphi_2 \rangle$.

Example 4.3.15. *For the sum of products in Example 4.3.13, we get* $fp_+((3)_s, \mathcal{S}) = (\emptyset, \mathcal{S}''')$. *Thus, according to the above theorem, it holds that* $\langle (3)_s, \varphi_3 \rangle \trianglelefteq \langle \mathcal{S}, \varphi \rangle$.

We have implemented our technique and it can be used as a back-end of existing non-asymptotic cost analyzers for average, lower and upper bounds (e.g., [41, 10, 53, 22, 26]), and regardless of whether it is based on the approach to cost analysis of [63] or any other. Currently, it is integrated within COSTA, and it can be tried out through its web interface which is available from `http://costa.ls.fi.upm.es`.

```
void f(int n) {
 List l = null;
 int i=0;
 while (i<n) {
  int j=0;
  while ( j<i ) {
   for(int k=0;k<n+j;k++)
    l=new List(i*k*j,l);
   j=j+random()?1:3;
  }
  i=i+random()?2:4;
 }
}
```

(a) Running Example

$$
\begin{array}{lll}
F(n) & = A(0,n) & \{\} \\
 & & \\
A(i,n) & = 0 & \{i{\geq}n\} \\
A(i,n) & = B(0,i,n){+}A(i',n) & \{i{+}1{\leq}n, i{+}2{\leq}i'{\leq}i{+}4\} \\
 & & \\
B(j,i,n) & = 0 & \{j{\geq}i\} \\
B(j,i,n) & = C(0,j,n){+}B(j',i,n) & \{j{+}1{\leq}i, j{+}1{\leq}j'{\leq}j{+}3\} \\
 & & \\
C(k,j,n) & = 0 & \{k{\geq}n{+}j\} \\
C(k,j,n) & = 1{+}C(k',j,n) & \{k'=k{+}1, k{+}1{\leq}n{+}j\}
\end{array}
$$

(b) *CRs* for Memory Consumption

Figure 4.2: Running Example and its Cost Relation System

4.4 Accurate Upper and Lower Bounds

This section summarizes the work published in [9] and [14] for solving *CRSs* into closed-form UBs and LBs. We motivate our work on a contrived example depicted in Figure 4.2a. The example is sufficiently simple to explain the main technical parts of the techniques, but still interesting to understand the challenges and precision gains. For this program and the *memory consumption* cost model, COSTA generates the *CR* shown in Figure 4.2b. This cost model estimates the number of objects allocated in the memory. Observe that the structure of the Java program and its corresponding *CR* match. The equations for C correspond to the `for` loop, those of B to the inner `while` loop and those of A to the outer `while` loop. The recursive equation for C states that the memory consumption of executing the inner loop with $\langle k, j, n\rangle$ such that $k + 1 \leq n + j$ is 1 (one object) plus that of executing the loop with $\langle k', j, n\rangle$ where $k' = k + 1$. The recursive equation for B states that executing the loop with $\langle j, i, n\rangle$ costs as executing $C(0, j, n)$ plus executing the same loop with $\langle j', i, n\rangle$ where $j + 1 \leq j' \leq j + 3$. While, in the Java program, j' can be either $j + 1$ or $j + 3$, due to the static analysis, the case for $j + 2$ is added in order to have a convex shape [32].

Since *CRs* are syntactically quite close to Recurrence Relations (*RRs*), in most cost analysis frameworks, it has been assumed that cost relations can be easily converted into *RRs*. This has led to the belief that it is possible to use existing Computer Algebra Systems (*CAS*) for finding closed-forms UBs and LBs in cost analysis. However, there are fundamental differences between *CRs* and *RRs* which make using *CAS* for solving *CRs* impractical. The following features have been identified in [9] as main differences, which in turn justify the need to develop specific solvers for solving *CRs*:

1. *CRs* often have *multiple arguments* that increase or decrease over the relation (e.g., in A of Figure 4.2b variable i' increases). The number of evaluation steps (i.e., recursive calls performed) is often a function of several such arguments.

2. *CRs* often contain *inexact size relations*, e.g., variables range over an interval $[a, b]$ (e.g., variable j' in B). Thus, given a valuation, we might have several solutions which perform a different number of evaluation steps.

3. Even if the original programs are deterministic, due to the loss of precision in the first stage of the static analysis, *CRs* often involve several *non-deterministic equations* (see Figure 2.1).

As a consequence of items 2 and 3, an exact solution often does not exist and hence *CAS* just cannot be used in such cases. And even if an exact solution exists, due to the three additional features, *CAS* do not accept *CRs* as a valid input.

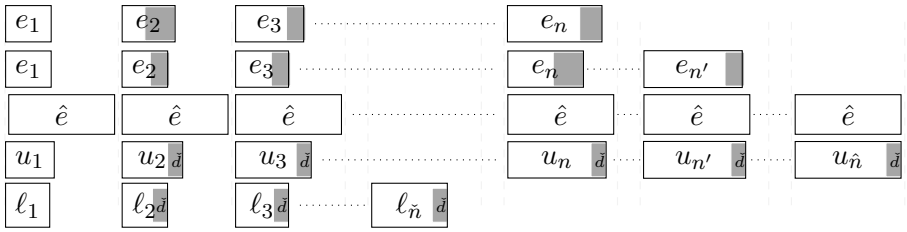


Figure 4.3: Concrete evaluations for one entry (rows 1, 2). UB approximation of [9] (row 3). Upper and lower bounds of [14] (rows 4 and 5).

In [9] we have developed a technique for solving *CRs* into closed-form UBs which relies on static program analysis instead of *CAS*. The main achievement of this work is its wide applicability when compared to *CAS*. Since precision is fundamental in some resource usage analysis, such as worst-case execution time (WCET), in [14] we have developed a technique which improves the precision of [9] and has similar applicability. Moreover, it is applicable also for inferring LBs. A main achievement of this work is the seamless integration of [9] and *CAS* that gets the best of both worlds: precision as *CAS*, whenever possible, while applicability as close to [9] as possible. To the best of our knowledge, this is the first general approach to inferring LBs from *CRs* and, as regards UBs, the one that achieves a better precision vs. applicability balance. All above approaches have been implemented in COSTA. The next section informally describes these approaches.

4.4.1 An Informal Account of Our Approach

Consider a *CR* in its simplest form with one base-case equation $\langle C(\bar{x})=0, \varphi_0\rangle$ and one recursive equation with a single recursive call $\langle C(\bar{x})=e+C(\bar{x}'), \varphi_1\rangle$. Given a concrete input valuation $\bar{v}$ for $\bar{x}$, the first two rows in Figure 4.3 represent two different possible results obtained for $C(\bar{v})$, depending on the assignments chosen in φ_1, where each e_i is the evaluation of e contributed by the i-th application of the recursive equation. In the first execution (first row), the recursive equation has been applied n times; while in the second execution (second row) we have applied it n' times. The shady part of each rectangle denotes the *progress* d_i between each two steps, i.e., $d_i{=}e_{i+1} - e_i$. Note that the progress is in general not constant. The distance might vary from one application of the equation to the next one and it does not necessarily increase.

The first two rows thus show two possible results of one particular valuation $C(\bar{v})$. Our challenge is to accurately estimate the cost of $C(\bar{x})$ for any input, i.e., we look for a UB (resp., LB) of $C(\bar{x})$ which is larger or equal (resp., smaller or equal) than all possible results of evaluating $C(\bar{x})$ for any valuation of $\bar{x}$. As we have explained above, *CAS* aims at obtaining an exact cost function, which is often not possible for *CRS* since, even for a single valuation, it might produce multiple solutions. Instead, we aim at inferring approximations and provide them as closed-form UBs/LBs for C.

Our first general approximation for UBs [9] has two dimensions: (1) *Number of applications of the recursive case:* The first dimension is to infer an UB on the number of times the recursive equations can be applied (which, for loops, corresponds to the number of loop iterations). This is done by inferring an UB $\hat{n}$ on the length of chains of calls; and (2) *Cost of applications:* The second dimension is to infer an UB $\hat{e}$ of the cost of each loop iteration, i.e., $\hat{e} \geq e_i$ for all i. Then, for the above *CR*, $\hat{n} * \hat{e}$ is guaranteed to be an UB for C. If the relation C had two recursive calls, the solution would be an exponential function of the form $2^{\hat{n}} * \hat{e}$. The third row of Figure 4.3 shows the result of this approximation for the simple *CR* we are considering. It can be observed that the cost of all iterations e_i is approximated by the same worst-case cost, which is $\hat{e}$. The number of iterations is approximated by $\hat{n}$ which is larger or equal than n, n', etc. In contrast to the first two rows, which are evaluations for concrete input data, the third row is a safe approximation of $C(\bar{x})$ for any input. We have proposed programming-languages techniques of wide applicability in order to solve those two dimensions:

Ranking functions. A ranking function is a function f such that for any recursive equation $\langle C(\bar{x})=e+C(\bar{x}_1)+\cdots+C(\bar{x}_k),\varphi\rangle$ in the *CR*, it holds that $\forall 1\leq i\leq k.\varphi \models f(\bar{x})>f(\bar{x}_i)\wedge f(\bar{x})>0$. This guarantees that when evaluating $C(\bar{v})$, the length of any chain of calls to C cannot exceed $f(\bar{v})$. Thus, f is used to bound the length of these chains [9, 18, 23].

Maximization. The second dimension is solved by first inferring an *invariant* $\langle C(\bar{x}_0)\rightsquigarrow C(\bar{x}),\Psi\rangle$, where Ψ is a set of linear constraints, which describes the relation between the values that $\bar{x}$ can take in any call to C and the initial values $\bar{x}_0$. Then, in order to generate $\hat{e}$, each $\mathsf{nat}(l)\in e$ is replaced by $\mathsf{nat}(\hat{l})$, where $\hat{l}$ is a linear expression (over $\bar{x}_0$) which is an UB for any valuation of l. Namely, inferring $\hat{l}$ is done by *syntactically* looking for a constraint $\xi\leq\hat{l}$ in $\bar{\exists}\bar{x}_0\cup\{\xi\}.\Psi\wedge\varphi_1\wedge\xi=l$ where ξ is a new variable. The advantage of this approach is that it can be implemented using any tool for manipulating linear constraints such as [15]. Alternatively, we can also use parametric integer programming [38] in order to maximize ξ w.r.t. the parameters $\bar{x}_0$.

Example 4.4.1. *Let us demonstrate this approach for CRs C and B of Figure 4.2b. A ranking function for C is $\mathsf{nat}(n_0+j_0-k_0)$, and thus $C^+(k_0,j_0,n_0)=\mathsf{nat}(n_0+j_0-k_0)*1$ is an UB for C. Replacing $C(0,j,n)$ by the UB $\mathsf{nat}(n+j)$ in the CR B results in $\langle B(j,i,n)=\mathsf{nat}(n+j)+B(j',i,n),\varphi_1=\{j<i,j+1\leq j'\leq j+3\}\rangle$. A ranking function for B is $\mathsf{nat}(i_0-j_0)$, and the maximization of $\mathsf{nat}(n+j)$ results in $\mathsf{nat}(n_0+i_0-j_0)$ since $n+j$ gets its maximum value when j is maximal (i.e. when $j=i_0-1$). Thus, $B^+(j_0,i_0,n_0)=\mathsf{nat}(i_0-j_0)*\mathsf{nat}(n_0+i_0-j_0)$ is an UB for B.*

Note that when trying to adapt this approach to inferring LBs, we only obtain trivial bounds. This is because the minimization of the cost expression accumulated along the execution is in most cases zero and, hence, by assuming it for all executions we would obtain a trivial (zero) LB.

Since precision is fundamental in some resource usage analysis, e.g., in WCET, in a recent work [14], we try to improve the precision of our first approach [9] in order make it comparable to that of using *CAS* while still keeping a similar applicability for UBs and, besides, be able to apply our approach to infer useful LBs. For UBs (resp., LBs), this approximation corresponds to the approximation shown in the fourth (resp., fifth) row of Figure 4.3. The fundamental idea is to generate a sequence of (different) elements $u_1,\ldots,u_{\hat{n}}$ such that for any concrete evaluation $e_1,\ldots,e_n$ of the equation, it holds that the last n elements we generate are larger or equal than the exact ones, i.e., $\forall 0\leq i\leq n-1.\ u_{\hat{n}-i}\geq e_{n-i}$. Then, clearly $u_1+\cdots+u_{\hat{n}}$ is an UB for $e_1+\cdots+e_n$. This UB is potentially more precise than $\hat{n}*\hat{e}$ (as shown in the third row), since each e_i is approximated more tightly by a corresponding u_j. Technically, we compute the approximation shown in the fourth row by transforming the *CR* into a (worst-case) *RR* whose closed-form solution is $u_1+\cdots+u_{\hat{n}}$. This *RR* has the following general from:

$$P(0)=\lambda$$
$$P(x)=E+P(x-1)+\cdots+P(x-1)$$

where E is a function on x (and might have constant symbols), and λ is a symbol representing the value of the base-case. The advantage of this *RR* is that it is simple enough to be solved by *CAS*.

When e is a simple linear expression such as $e\equiv\mathsf{nat}(l)$, the novel idea is to view $u_1,\cdots,u_{\hat{n}}$ as an arithmetic sequence that starts from $u_{\hat{n}}\equiv\hat{e}$ (last element of fourth row), and that each time decreases by $\check{d}$, where $\check{d}$ is an under-approximation of all $d_i=e_{i+1}-e_i$, i.e., $u_i=u_{i-1}+\check{d}$. Let us observe in the fourth row of the figure that such a distance is constant in our approximation. When e is a complex non-linear expression, e.g., $\mathsf{nat}(l)*\mathsf{nat}(l')$, it cannot be precisely approximated using sequences. For such cases, our novel contribution has been a method for approximating e by approximating its nat subexpressions (which are linear) separately.

Example 4.4.2. *Let us demonstrate this approach for CR B of Example 4.4.1 (after replacing the UB of C). The expression $\mathsf{nat}(n+j)$ increases at least by $\check{d}$. Using this $\check{d}$, the ranking function $\mathsf{nat}(i_0-j_0)$, and*

the maximization $\mathsf{nat}(n_0 + i_0 - 1)$ *of* $\mathsf{nat}(n_j)$*, we generate the following RR*

$$\begin{array}{l} P_B(0) = 0 \\ P_B(x) = \mathsf{nat}(n_0 + j_0{-}1) + (\mathsf{nat}(i_0{-}j_0){-}x{+}1){*}1 + P_B(x{-}1) \end{array}$$

which is solved by CAS to

$$P_B(x) = \mathsf{nat}(n_0 + j_0 - 1) * x + \mathsf{nat}(i_0 - j_0) * x + \frac{x}{2} - \frac{x^2}{2}$$

Note that variable x *refers to the number of applications of the recursive equations, thus, changing it by the ranking function* $\mathsf{nat}(i_0 - j_0)$ *results in the following UB for* B

$$B^+(j_0, i_0, n_0) = \mathsf{nat}(n_0 + j_0 - 1) * \mathsf{nat}(i_0 - j_0) + \frac{\mathsf{nat}(i_0 - j_0)}{2} * (\mathsf{nat}(i_0 - j_0) + 1)$$

which is more precise than the one of Example 4.4.1.

Interestingly, this technique can be applied to cost expressions with any progression behavior that can be modeled using sequences, and not only to those with a linear progression behavior.

An important advantage of this approach w.r.t. the previous one and other related approaches [41, 45], is that the problem of inferring LBs is dual. In particular, we can infer a LB $\check{n}$ on the length of chains of recursive calls, the minimum value $\check{e}$ to which e can be evaluated, and then sum the sequence $\ell_1, \ldots, \ell_{\check{n}}$ where $\ell_i = \ell_{i-1} + \check{d}$ and $\ell_1 = \check{e}$. The LB approximation is shown graphically in the fifth row of Figure 4.3. In practice these two subproblems are solved as follows:

LB on the length of chains of calls. Given a *CR* C, a LB on the length of any chain of calls during the evaluation of $C(\bar{x}_0)$ is computed as follows:

1. *Instrumentation*: Replace each head $C(\bar{x})$ by $C(\bar{x}, lb)$, each recursive call $C(\bar{x}_j)$ by $C(\bar{x}_j, lb')$, and add $\{lb' = lb + 1\}$ to each φ_1;
2. *Invariant*: Infer an invariant $\langle C(\bar{x}_0, 0) \rightsquigarrow C(\bar{x}, lb), \Psi\rangle$ for the new *CR*, such that the linear constraints Ψ hold between the (variables) of the initial call $C(\bar{x}_0, 0)$ and any recursive call $C(\bar{x}, lb)$; and
3. *Synthesis*: Syntactically look for $lb \geq l$ in $\bar{\exists}\bar{x}_0 \cup \{lb\}.\ \Psi \wedge \varphi_0$.

Then, $\mathsf{nat}(l)$ is a LB on the length of the chains of calls for C. Let us give an intuition on the different steps of the above definition: In step 1, the *CR* C is instrumented with an extra argument lb which practically computes the length of the corresponding chain of calls, when starting the evaluation from $C(\bar{x}_0, 0)$. This instrumentation reduces the problem of finding a LB on the length of any chain of calls to the problem of finding a (symbolic) minimum value for lb for which the base-case equation is applicable (i.e., the chain of calls terminates). This is exactly what steps 2 and 3 do. In 2, we infer an invariant Ψ on the arguments of any call $C(\bar{x}, lb)$ encountered during the evaluation of $C(\bar{x}_0, 0)$. This is done exactly as for the invariant when maximizing cost expressions. In 3, from all states described by Ψ, we are interested only in those in which the base-case equation is applicable, i.e., in $\Psi \wedge \varphi_0$. Then, within this set of states, we look for a symbolic expression $lb \geq l$ where l is an expression over $\bar{x}_0$. Such an l is the LB we are interested in. Note that instead of syntactically looking for $lb \geq l$, we can also use parametric integer programming [38] in order to minimize lb w.r.t. the parameters $\bar{x}_0$.

Example 4.4.3. *Adding the loop counter* lb *to the CR* B *of Example 4.4.1 results in*

$$\begin{array}{lcll} \langle B(j,i,n,lb) & = & 0 & \{j \geq i\}\rangle \\ \langle B(j,i,n,lb) & = & \mathsf{nat}(n+j) + B(j',i,n,lb') & \{j < i, j+1 \leq j' \leq j+3, lb' = lb+1\}\rangle \end{array}$$

The invariant for this CR is $\Psi = \{j - j_0 - lb \geq 0, j_0 + 3 * lb - j \geq 0, i = i_0, n = n_0\}$*. Projecting* $\Psi \wedge \{j \geq i\}$ *on* $\langle j_0, i_0, n_0, lb\rangle$ *results in* $\{j_0 + 3 * lb - i_0 \geq 0\}$ *which implies* $lb \geq \frac{(i_0 - j_0)}{3}$*, from which we can synthesize* $\check{f}_B(j_0, i_0, n_0) = \mathsf{nat}(\frac{i_0 - j_0}{3})$.

Minimization. Inferring the minimum value to which $\mathsf{nat}(l) \in e$ can be evaluated is done in a dual way to that of inferring the maximum value to which it can be evaluated. Namely, using the above invariant Ψ, we syntactically look for an expression $\xi \geq \check{l}$ in $\bar{\exists}\bar{x}_0 \cup \{\xi\}$. $\Psi \wedge \varphi_1 \wedge \xi = l$ where ξ is a new variable. As in the case of maximization, the advantage of this approach is that it can be implemented using any tool for manipulating linear constraints such as [15]. Alternatively, we can also use parametric integer programming [38] in order to minimize ξ w.r.t. the parameters $\bar{x}_0$.

Example 4.4.4. *The minimization of* $\mathsf{nat}(n + j)$ *of the CR B of Example 4.4.1 results in* $\mathsf{nat}(n_0 + j_0)$. *Using this expression, the LB on the length of the chains of calls inferred in Example 4.4.3, and* $\check{d} = 1$, *we generate the following (best-case) RR*

$$\begin{array}{l} P_B(0) = 0 \\ P_B(x) = \mathsf{nat}(n_0 + j_0) + (\mathsf{nat}(\frac{i_0 - j_0}{3}) - x) * 1 + P_B(x-1) \end{array}$$

which is solved by CAS to

$$P_B(x) = \mathsf{nat}(n_0 + j_0) * x + \mathsf{nat}(\frac{i_0 - j_0}{3}) * x - \frac{x^2}{2} - \frac{x}{2}$$

Then, replacing x *by the minimum length of the chains of calls results in the following LB for B*

$$B^{-}(j_0, i_0, n_0) = \frac{1}{2} * \mathsf{nat}(\frac{i_0 - j_0}{3}) * (\mathsf{nat}(\frac{i_0 - j_0}{3}) + 2 * \mathsf{nat}(n_0 + j_0 - \frac{1}{2}))$$

Chapter 5

Verification of Resource Guarantees using KeY + COSTA

Resource guarantees allow being certain that programs will run within the indicated amount of resources, which may refer to memory consumption, number of instructions executed, etc. This information can be very useful, especially in real-time and safety-critical applications. Nowadays, a number of automatic tools exist, often based on type systems or static analysis, which produce such resource guarantees. For example, the COSTA tool described in the previous chapters. In spite of being based on theoretically sound techniques, the implemented tools may contain bugs which render the resource guarantees thus obtained not completely trustworthy. Performing full-blown verification of such tools is a daunting task, since they are large and complex. In this work we investigate an alternative approach whereby, instead of the *tools*, we formally verify the *results* of the tools. We have implemented this idea using COSTA for producing resource guarantees and KeY, a state-of-the-art verification tool, for formally verifying the correctness of such resource guarantees. Our preliminary results show that the proposed tool cooperation can be used for automatically producing verified resource guarantees.

5.1 Introduction

There is a growing awareness, both in industry and academia, of the crucial role of formally proving the correctness of systems. Verifying the correctness of modern static analyzers is rather challenging, among other things, because of the sophisticated algorithms used in them, their evolution over time, and, possibly, proprietary considerations. A simpler alternative is to construct a validating tool [55] which, after every run of the analyzer, formally confirms that the results are correct and, optionally, generates correctness proofs. Such proofs could then be translated to *resource certificates*, in the proof-carrying code style [33, 54].

In this work, we are interested in *resource guarantees* obtained by static analysis. An essential aspect of programs is that resources be used effectively. This is especially true in the context of software families, which provide us with mechanisms for code reuse by means of components and services: not only functionality, but also resource consumption (or *cost*) must be taken into consideration.

The analysis algorithms used in COSTA for inferring the three main components of the UB generation (i.e., ranking functions, loop invariants, and size relations) were proven correct at a theoretical level. However, there is no guarantee that correctness is preserved in the actual implementation, which is rather involved and includes translation to an intermediate language.

KeY [17] is a state-of-the-art source code verification tool for the Java programming language. Its coverage of Java is comparable to that of COSTA (nearly full sequential Java, plus a simplified concurrency model). KeY implements a logic-based setting of symbolic execution that allows deep integration with aggressive first-order simplification. While the degree of automation of KeY is very high on loop- and recursion-free programs, the user must in general supply suitable invariants to deal with loops and recursion. In general, invariants that are sufficient to prove complex functional properties cannot be inferred automatically. How-

ever, simpler invariants that are sufficient to establish UBs *can* be automatically derived in many cases and this is exactly COSTA's forte. Our work is based on the insight that the static analysis tool COSTA and the formal verification tool KeY have complementary strengths: COSTA is able to derive UBs of Java programs including the invariants needed to obtain them. This information is enough for KeY to *prove* the validity of the bounds and provide a certificate. The main contribution of this work is to show that, using KeY, it is possible to formally and automatically verify the correctness of the UBs obtained by COSTA. Realizing the cooperation between COSTA and KeY required a number of non-trivial extensions of both systems, which are described in more detail below.

5.2 Inference of UBs in COSTA

In this section, we briefly recall the techniques used in COSTA for automatically inferring UBs, and we identify the proof obligations that need to be verified using KeY.

5.2.1 Main Components of an UB

Consider the following (JML annotated) program that implements the insert sort algorithm.

```
1  public class NestedLoops {
2    static void insert_sort(int A[]) {
3      int i, j, v;
4      //@ ghost int i0=i; int j0=j; int a0=a;
5      i=A.length-2;
6      //@ assert (i=i0−2 ∧ j=j0 ∧ a=a0)
7      //@ ghost int i1=i; int j1=j; int a1=a;
8      //@ loop_invariant i≤i1
9      //@ decreases i>0 ? i : 0
10     while ( i>=0 ) {
11       //@ ghost int i2=i; int j2=j; int a2=a;
12       j=i+1;
13       v=A[i];
14       //@ assert j=i2+1 ∧ i2 ≥ 0
15       //@ ghost int i3=i; int j3=j; int a3=a;
16       //@ loop_invariant j≤a3
17       //@ decreases a − j>0 ? a − j : 0
18       while ( j<A.length && A[j]<v) {
19         A[j-1]=A[j];
20         j++; }
21       A[j-1]=v;
22       i--;
23     }
24   }
25 }
```

COSTA receives a non-annotated version of the above program and, for the cost model that counts the number of executed bytecode instructions, produces the (asymptotic) UB `insert_sort(`a`)=`a^2, where a refers to `A.length`. The underlying analysis used in COSTA infers UBs for each iterative and recursive constructs (loops) and then composes the results in order to obtain an UB for the method of interest. Intuitively, in order to infer an UB for a single loop, it first infers an UB A on the cost of a single execution of its body, an UB I on the number of iterations that it can make, and then $A * I$ is an UB for the loop. To infer A and I COSTA relies on several program analysis components that provide essential information as

has been explained in the previous chapters. We summarize the essentials here for the convenience of those who want to read the present chapter stand-alone.

Ranking functions. For each loop, COSTA infers a linear function from the loop variables to $\mathbb{N}$ which is decreasing at each iteration. For example, for the loop at line 18, it infers function $f(a,j) = \mathsf{nat}(a-j)$. This function can be safely used to bound the number of iterations. In the example, if a_3 and j_3 are the initial values of a and j, then it is guaranteed that $f(a_3, j_3)$ is an UB on the number of iterations of the loop.

Loop invariants. For each loop in the program, COSTA infers an invariant that involves the loop's variables and their initial values (i.e., their values before entering the loop). Let us denote by i_1 the initial value of i when entering the loop at line 10. COSTA infers the invariant $i \leq i_1$, which states that i is always smaller than or equal to its initial value when the program reaches the loop condition. This information, together with the size relations below, is needed to compute the worst-case cost of executing one loop iteration.

Size relations. Given a fragment of code or a scope (details below), COSTA infers relations between the values of the program variables at a certain program point of interest within the scope and their initial values when entering the scope. For example, at line 14, it infers that $j = i_2 + 1$, where i_2 is the value of i when entering the scope that contains line 14 (i.e., the scope here is the loop body). In this case the relation is a simple consequence of the instruction at line 12. In general, however, it may not be trivial to infer such relations nor to prove that they are correct.

UBs. Once the above information has been inferred, it is straightforward to compute an UB for the method. Let us show this process on the running example:

Inner loop. The process starts from the innermost loops. Thus, we start with the loop at line 18. Assuming that executing the condition costs (at most) c_1 instructions, and that the cost of each iteration (i.e., the loop body) is c_2 instructions, then it is clear that $\mathsf{nat}(a_3 - j_3) * (c_1 + c_2) + c_1$ is an UB on the cost of this loop (because c_1 and c_2 are constant).

Outer loop. Next, we move to the outer loop at line 10. Let us assume that the cost of the comparison is c_3 instructions, the code at lines $12 - {-}13$ are c_4 instructions, and the code at lines $21 - {-}22$ are c_5 instructions. Then, the cost of each iteration of this loop is $c_3 + c_4 + \underline{\mathsf{nat}(a_3 - j_3) * (c_1 + c_2) + c_1} + c_5$, where the highlighted subexpression corresponds to the cost of the inner loop computed above. Note that in this case, each iteration might have a different cost, since $a_3 - j_3$ is not the same for all iterations. As detailed in the previous nchapter, simply multiplying the number of iterations $\mathsf{nat}(i_1)$ by such a cost is unsound. The solution is to find an expression U in terms of the initial values of a_1, i_1, j_1 which does not change during the loop such that $U \geq a_3 - j_3$ in all iterations. Then, $\mathsf{nat}(i_1) * [c_3 + c_4 + \underline{\mathsf{nat}(U) * (c_1 + c_2) + c_1} + c_5] + c_3$ is an UB for the loop. In order to find such a U, COSTA uses the loop invariant (line 8) and the size relations (line 14) as follows: it solves the parametric integer programming (PIP) problem of maximizing the objective function $a_3 - j_3$ w.r.t. the loop invariant and the size relations where i_1, a_1, j_1 are the parameters. This produces an expression in terms of i_1, a_1, j_1 which is greater than or equal to $a_3 - j_3$ in all iterations of the loop. In our example, it is $U = a_1 - 1$.

Method. We finally can compute the cost of the `insert_sort` method. Assume that the cost of line 5 is c_6, then the cost of the method is $c_6 + \underline{\mathsf{nat}(i_1) * [c_3 + c_4 + \mathsf{nat}(a_1 - 1) * (c_1 + c_2) + c_1 + c_5]} + c_3$. We need to express this UB in terms of the input parameter a. For this, COSTA maximizes (using PIP) i_1 and $a_1 - 1$ w.r.t. the size relation at line 6 and, respectively, obtains $a - 2$ and $a - 1$. Therefore, $c_6 + \mathsf{nat}(a-2) * [c_3 + c_4 + \underline{\mathsf{nat}(a-1) * (c_1 + c_2) + c_1} + c_5] + c_3$ is the UB for `insert_sort`

5.2.2 COSTA Claims as JML Annotations

To justify that the UBs obtained by COSTA are correct, we need to provide formal correctness proofs for all the claims above. This includes the ranking functions, invariants, size relations, the cost model that

provides all c_i, and the underlying PIP solver.

Correctness of the cost model is trivial as it is a simple mapping from each instruction to a number. Correctness of the underlying PIP solver is also straightforward if we use the maximization procedure defined in [9], which is based only on the Gaussian elimination algorithm. Therefore, we concentrate on verifying the correctness of the ranking functions, size relations and invariants. They are inferred by large software components whose correctness has not been verified. We now briefly describe the translation of the different pieces of information generated by COSTA to JML annotations on the Java program, which will allow their verification in KeY.

Ranking functions. For a given loop, when COSTA infers a ranking function of the form $\mathsf{nat}(\ell)$, we translate it to the JML annotation "`//@ decreases` $\ell > 0 \ ? \ \ell : 0$", since $\mathsf{nat}(\ell)$ can be defined as an if-then-else. COSTA might provide also ranking functions of the form $\log(\mathsf{nat}(\ell) + 1)$, which are handled similarly.

Invariants. COSTA infers an invariant φ for each loop. This invariant involves the loop variables $\bar{v}$ and auxiliary variables $\bar{w}$ such that each w_i represents the initial value of v_i. The JML annotation for this invariant consists of one line defining all $\bar{w}$ as ghost variables ("`//@ ghost int` $w_1 = v_1$`;...; int` $w_n = v_n$") and one line for declaring the loop invariant ("`//@ loop_invariant` φ").

Size relations. Size relations are linear constraints between the values of a set of variables of interest between two program points. As we have seen, this allows composing the cost of the different program fragments. For each loop (or method call), COSTA infers the relation φ between the values before the loop entry (or the call) and the entry of its parent scope. Suppose that the loop (or the call) is at line L_l, its parent scope starts at line L_p, and that $\bar{v}$ are the variables of interest at L_l and $\bar{w}$ represent their values at L_p. Then we add the JML annotation "`//@ ghost int` $w_1 = v_1$`;...; int` $w_n = v_n$`;`" immediately after line L_p to capture the values of $\bar{v}$ at line L_p, and the JML annotation "`//@ assert` φ" immediately before line L_l to state that the relation φ must hold at the program point. Additional size relations inferred by COSTA are IO size relations. These are linear constraints that relate the return value of a given method to its input values. For example, suppose that we replace "`i--`" in line 22 of the `insert_sort` program by "`i=decrement(i)`" where `decrement` is defined by "`int decrement(int x) {return x-1;}`". Then COSTA infers the relation "$\varphi \equiv$`\result=x-1`" which is used to bound the number of iterations of that loop. In order to verify this relation in KeY we add the JML annotation "`//@ ensures` φ" to the contract of `decrement`.

5.3 Verification of UBs using KeY

We now describe the verification techniques used in KeY to prove program correctness, focusing on those relevant to UB verification.

5.3.1 Verification by Symbolic Execution

The program logic used by KeY is *JavaCard Dynamic Logic* (JavaDL) [17], a first-order dynamic logic with arithmetic. Programs are first-class citizens similar to Hoare logics but, in dynamic logic, correctness assertions can appear arbitrarily nested. JavaDL extends sorted first-order logic by a program modality $\langle\cdot\rangle\cdot$ (read "diamond"). Let p denote a sequence of executable Java statements and ϕ an arbitrary JavaDL formula, then $\langle\mathsf{p}\rangle\phi$ is a JavaDL formula which states that program p terminates and in its final state ϕ holds. A typical formula in JavaDL looks like

$$\mathsf{i} \doteq i0 \wedge \mathsf{j} \doteq j0 \ \texttt{->} \ \langle\overbrace{\texttt{i=j-i;j=j-i;i=i+j;}}^{\mathsf{p}}\rangle(\mathsf{i} \doteq j0 \wedge \mathsf{j} \doteq i0)$$

where i, j are program variables represented as *non-rigid* constants. Non-rigid constants and functions are state-dependent: their value can be changed by programs. The *rigid* constants $i0, j0$ are state-independent: their value cannot be changed. The formula above says that if program p is executed in a state where i and j have values $i0$, $j0$, then p terminates *and* in its final state the values of the variables are swapped. To reason about JavaDL formulas, KeY employs a sequent calculus whose rules perform *symbolic execution* of the programs in the modalities. Here is a typical rule:

$$\text{ifSplit}\ \frac{\Gamma, b \Longrightarrow \langle \{\mathsf{p}\}\mathsf{rest}\rangle\phi, \Delta \quad \Gamma, \neg b \Longrightarrow \langle \{\mathsf{q}\}\mathsf{rest}\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \mathsf{if\ (b)\ \{p\}\ else\ \{q\}\ rest}\rangle\phi, \Delta}$$

As values are symbolic, it is in general necessary to split the proof whenever an implicit or explicit case distinction is executed. It is also necessary to represent the *symbolic* values of variables throughout execution. This becomes apparent when statements with side effects are executed, notably assignments. The assignment rule in JavaDL looks as follows:

$$\text{assign}\ \frac{\Gamma \Longrightarrow \{\mathsf{x} := \mathsf{val}\}\langle \mathsf{rest}\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \mathsf{x = val;\ rest}\rangle\phi, \Delta}$$

The expression in curly braces in the premise is called *update* and is used in KeY to represent symbolic state changes. An *elementary* update $loc := val$ is a pair of a location (program variable, field, array) and a value. The meaning of updates is the same as that of an assignment, but they can be composed in different ways to represent complex state changes. Updates u_1, u_2 can be composed into *parallel updates* $u_1 \| u_2$. In case of clashes (updates u_1, u_2 assign different values to the same location) a last-wins semantics resolves the conflict. This reflects left-to-right sequential execution. Apart from that, parallel updates are applied simultaneously, i.e., they do not depend on each other. Update application to a formula/term e is denoted by $\{u\}e$ and forms itself a formula/term. Application of updates is similar to explicit substitutions, but is aware of aliasing.

Loops and recursive method calls give rise to infinitely long symbolic executions. Invariants are used in order to deal with unbounded program structures (an example is given below). Exhaustive application of symbolic execution and invariant rules results in formulas of the form $\{u\}\langle\rangle\phi$ where the program in the modality has been fully executed. At this stage, symbolic updates are applied to the postcondition ϕ resulting in a first-order formula that represents the weakest precondition of the executed program w.r.t ϕ.

5.3.2 Proof-Obligation for Verifying UBs

To verify UBs in KeY the annotated source code files provided by COSTA are loaded. For methods where COSTA did not generate a contract, KeY provides the following default JML contract:

```
/*@ public behavior
  @  requires true;
  @  ensures  true;
  @  signals_only Exception;
  @  signals (Exception) true; @*/
```

This contract requires to prove termination for any input and ensures that all possible execution paths are analyzed. Abrupt termination by uncaught exceptions is allowed (signals clauses). To prove that a method m satisfies its contract, a JavaDL formula is constructed which is valid iff m satisfies its contract. Slightly simplified, for insert_sort from Section 5.2.1 this formula (using the default contract) is:

$$\forall o; \forall a0; \{\mathsf{a} := a0 \parallel \mathsf{self} := o\}(\neg(\mathsf{a} \doteq \mathsf{null}) \land \neg(\mathsf{self} \doteq \mathsf{null}) \rightarrow$$
$$\langle\ \mathsf{try\ \{\ self.insert\_sort(a)@NestedLoops;\ \}}$$
$$\mathsf{catch(Exception\ e)\{\ exc=e;\ \}}\rangle(\ \mathsf{exc} \doteq \mathsf{null} \lor; \mathrm{instance}_{\mathsf{Exception}}(\mathsf{exc}))$$

The above formula states that for any possibly value o of self and any value $a0$ of the argument a which satisfy the implicit JML preconditions (self and a are not null), the method invocation self.insert_sort(a) *terminates* (required by the use of the diamond modality) and in its final state no exception has been thrown or any thrown exception must be of type Exception.

5.3.3 Verification of Proof-Obligations

The proof obligation formula must be proven valid by executing the method insert_sort symbolically starting with the execution of the variable declarations. Ghost variable declarations and assignments to ghost variables (`//@ set var=val;`) are symbolically executed just like Java assignments.

Verifying Size Relations. If a JML assertion assert φ; is encountered during symbolic execution, the proof is split: the first branch must prove that the assertion formula φ holds in the current symbolic state; the second branch continues symbolic execution. In the insert_sort example, a proof split occurs exactly before entering each loop. This verifies the size relations among variables as derived by COSTA and encoded in terms of JML assertion statements (see Section 5.2.2). IO size relations encoded in terms of method contracts are proven correct as outlined in Section 5.3.2.

Verifying Invariants and Ranking Functions. Verification of the loop invariants and ranking functions obtained from COSTA is achieved with a tailored loop invariant rule that has a variant term to ensure termination:

$$\text{loopInv}\ \frac{\begin{array}{ll}(i) & \Gamma \Longrightarrow Inv \wedge dec \geq 0, \Delta \\ (ii) & \Gamma, \{\mathcal{U}_A\}(b \wedge Inv \wedge dec \doteq d0) \Rightarrow \\ & \quad \{\mathcal{U}_A\}\langle \mathsf{body}\rangle(Inv \wedge dec < d0 \wedge dec \geq 0), \Delta \\ (iii) & \Gamma, \{\mathcal{U}_A\}(\neg b \wedge Inv) \Longrightarrow \{\mathcal{U}_A\}\langle \mathsf{rest}\rangle\phi, \Delta \end{array}}{\Gamma \Longrightarrow \langle \mathsf{while\ (b)\ \{\ body\ \}\ rest}\rangle\phi, \Delta}$$

Inv and *dec* are obtained, respectively, from the loop_invariant and decreases JML annotations generated by COSTA. Premise (i) ensures that invariant *Inv* is valid just before entering the loop and that the variant *dec* is non-negative. Premise (ii) ensures that *Inv* is preserved by the loop body and that the variant term decreases strictly monotonic while remaining non-negative. Premise (iii) continues symbolic execution upon loop exit. The integer-typed variant term ensures loop termination as it has a LB (0) and is decreased by each loop iteration. Using COSTA's derived ranking function as variant term obviously verifies that the ranking function is correct. The update $\mathcal{U}_A$ assigns to all locations whose values are potentially changed by the loop a fixed, but unknown value. This allows using the values of locations that are unchanged in the loop during symbolic execution of its body.

Generated Proofs. A single proof for each method is sufficient to verify the correctness of the derived loop invariants, ranking functions and size relations. The reason is that the contracts capturing the IO size relations are not more restrictive w.r.t. the precondition than the default contracts are. Hence, with the verification of the IO size relation contracts, we analyze all feasible execution paths and prove correctness of all loop invariants, ranking functions and JML assertion annotations.

5.4 Implementation and Experiments

The implementation of our approach has required the following non-trivial extensions to COSTA and KeY (note that COSTA works on Java bytecode, and KeY on Java source): (1) output the proof obligations using the original variable names (at the bytecode level, operand stack variables are often used); (2) place the obligations in the Java source at the precise program points where they must be verified (entry points of loops); (3) finding a suitable JML format for representing proof obligations on UBs has required a considerable number of iterations (defining ghost variables, introducing `assert` constructs, etc.); (4) implement the JML `assert` construct in KeY which was not supported hitherto. To express assertions which have to hold before a method call but after parameter binding support for a second assertion construct `invocAssert` has been added. Eclipse plugins for both the extended COSTA and KeY systems are available from `http://pepm2011.hats-project.eu`.

Bench	COSTA					KeY			**Total**
	$\mathbf{T}_{size}$	$\mathbf{T}_{inv}$	$\mathbf{T}_{rf}$	$\mathbf{T}_{ana}$	$\mathbf{T}_{jml}$	**Nodes**	**Branches**	$\mathbf{T}_{ver}$	
slm	22	20	26	112	4	3641	36	6700	6816
nlf	30	16	24	106	6	5665	37	2800	2912
bubsort	38	24	144	296	14	14890	230	57800	58110
inssort	30	12	46	142	6	9875	167	29300	29448
selsort	40	20	112	232	8	12564	209	40700	40940
pastri	66	38	138	394	14	29723	337	110100	110508

Table 5.1: Statistics about the Analysis and Verification Process

Table 5.1 shows some preliminary experiments using a set of representative programs, available from the above website, which include sorting algorithms, namely bubble sort (*bubsort*), insert sort (*inssort*), and selection sort (*selsort*); a method to generate a Pascal Triangle (*pastri*); simple (*slm*) and nested loops (*nlf*). Columns $\mathbf{T}_{size}$, $\mathbf{T}_{inv}$, $\mathbf{T}_{rf}$, $\mathbf{T}_{ana}$ and $\mathbf{T}_{jml}$ show, respectively, the times taken by COSTA to obtain the size relations, loop invariants, ranking functions, the whole analysis (which includes the previous times) and generate the JML annotations. Column $\mathbf{T}_{ver}$ shows the time taken by KeY in order to verify the JML annotations generated by COSTA. As time measurements for Java are imprecise we state in addition the number of nodes and branches of the generated proof to provide some insight on the proof complexity. Column **Total** shows the time taken by the whole process. All times are measured in ms and were obtained using an Intel Core2 Duo P8700 at 2.53GHz with 4Gb of RAM running a Linux 2.6.32 (Ubuntu Desktop). A notable result of our experiments is that KeY was able to spot a bug in COSTA, as it failed to prove correct one invariant which was in fact incorrect. In addition, KeY could provide a concrete counterexample that helped understand, locate and fix the bug, which was related to a recently added feature of COSTA.

Chapter 6

Related Work

The concurrency primitives in the X10 language [25] are similar to those of ABS, namely asynchronous calls, future variables and await instructions have the same behavior. This is why the task-level analysis in [3] has been presented in the context of X10, but it is applicable to ABS programs as well. The RRs in [3] are specifically tailored for capturing information on the (maximum) number of tasks that can run in parallel along a program's execution but cannot be used for capturing the cost of executing each task, as the general cost analysis framework of Chapter 3 does. Furthermore, the size analysis in [3] is field-insensitive, does not use class invariants, and the equations do not specify cost centers. Therefore, the whole development in Chapter 3 is in general more powerful. On the other hand, [3] is not directly an instance of Chapter 3 because the maximum task-level is not accumulative. This requires generating a special form of RRs. We believe such equations could be generated within our framework in a similar way. It is worth noticing also that the may-happen-in-parallel analysis [50] is a complementary line of research to ours, in the sense that we can use their results to obtain tighter class invariants.

Our work is closely related to resource usage analysis frameworks [41, 43]. All such frameworks assume a sequential execution model. Therefore, they do not deal with the main challenges addressed in Chapter 3 to define the cost analysis framework. A notable exception is [48] that proposes a live heap space analysis for a concurrent language. The problem in their case is simpler since they do not have shared memory like we do. Besides, they consider a particular type of resource (memory) while our approach is developed for a generic type of resource which can then be instantiated to accumulative resources. The use of cost centers has been proposed in the context of profiling, but to our knowledge, its use in the context of static analysis is new. Termination of multi-threaded programs has been studied before in [28]. Unlike ours, their solution is based on inferring conditions that are required by a given thread from its environment (i.e., other threads) in order to guarantee its termination.

Modular analysis has received considerable attention in different programming paradigms, ranging from, e.g., logic programming [36, 29, 27] to object-oriented programming [56, 20, 51]. A general theoretical framework for modular abstract interpretation analysis was defined in [31], but most of the existing works regarding modular analysis have focused on specific analyses with particular properties and using more or less ad-hoc techniques.

Previous work from some of the authors of this report presents and empirically tests a modular analysis framework for logic programs [36, 29]. There are important differences with the present report: in addition to the programming paradigm, the framework of [36] is designed to handle one abstract domain, while the framework presented in this paper handles several domains at the same time, and the previous work is based on CIAOPP, a polyvariant context-sensitive analyzer in which an intermodular fixpoint algorithm was performed.

In [56] a control-flow analysis-based technique is proposed for call graph construction in the context of OO languages. Although there have been other works in this area, the novelty of this approach is that it is context-sensitive. Also, [20] shows a way to perform modular class analysis by translating the OO program into *open* DATALOG programs.

In [51] an abstract interpretation based approach to the analysis of class-based, OO languages is presented. The analysis is split in two separate semantic functions, one for the analysis of an object and another one for the analysis of the context that uses that object. The interdependence between context and object is expressed by two mutually recursive equations. In addition, it is context-sensitive and polyvariant.

Regarding the problem of comparing cost functions described in Section 4.3, in [40], an approach for inferring non-linear invariants using a linear constraints domain (such as polyhedra) has been introduced. The idea is based on a *saturation* operator, which lifts linear constraints to non-linear ones. For example, the constraint $\Sigma a_i x_i = a$ would impose the constraint $\Sigma a_i Z_{x_i u} = au$ for each variable u. Here $Z_{x_i u}$ is a new variable which corresponds to the multiplication of x_i by u. This technique can be used to compare cost functions, the idea is to start by saturating the constraints and, at the same time, converting the expressions to linear expressions until we can use a linear domain to perform the comparison. For example, when we introduce a variable $Z_{x_i u}$, all occurrences of $x_i u$ in the expressions are replaced by $Z_{x_i u}$. Let us see an example where, in the first step, we have the two cost functions to compare; in the second step, we replace the exponential with a fresh variable and add the corresponding constraints; in the third step, we replace the product by another fresh variable and saturate the constraints:

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

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

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

Another approach for this problem is based on numerical methods since our problem is analogous to proving whether $0 \preceq b - f_m$ in the context ϕ_b. There are at least two numerical approaches to this problem. The first one is to find the roots of $b - f_m$, and check whether those roots satisfy the constraints ϕ_b. If they do not, a single point check is enough to solve the problem. This is because, if the equation is verified at one point, the expressions are continuous, and there is no sign change since the roots are outside the region defined by ϕ_b, then we can ensure that the equation holds for all possible values satisfying ϕ_b. However, the problem of finding the roots with multiple variables is hard in general and often not solvable. The second approach is based on the observation that there is no need to compute the actual values of the roots. It is enough to know whether there are roots in the region defined by ϕ_b. This can be done by finding the minimum values of expression $b - f_m$, a problem that is more affordable using numerical methods [49]. If the minimum values in the region defined by ϕ_b are greater than zero, then there are no roots in that region. Even if those minimum values are out of the region defined by ϕ_b or smaller than zero, it is not necessary to continue trying to find their values. If the algorithm starts to converge to values out of the region of interest, the comparison can be proven to be false. One of the open issues about using numerical methods to solve our problem is whether or not they will be able to handle cost functions output from realistic programs and their performance. We have not explored these issues yet and they remain as subject of future work.

Interestingly, our technique is not restricted to any complexity class or cost analyzer: static cost analyzers should produce, for any program, cost functions which fall into the syntactic form we handle. On the other hand, our approach is clearly incomplete and may fail to prove that a cost function is smaller than another one in some cases where the relation actually holds. It remains as future work to try and combine our

approach with more heavyweight techniques, such as those based on numerical methods, in those cases where our approach is not sufficiently precise.

Finally, in Section 4.4 we proposed a novel approach to infer precise UBs and LBs of *CRs* which, as our experiments show, achieves a very good balance between the accuracy of our analysis and its applicability. Currently, we have applied it to *CRs* obtained from Java bytecode programs and also from X10 programs [3]. In the latter case, the language has concurrency primitives to spawn asynchronous tasks and to wait for termination of tasks. In spite of being a concurrent language, the first phase of cost analysis handles the concurrency primitives and the generated *CRs* can be solved directly using our approach.

The work that is most closely related to automatic inference of symbolic UBs includes [44, 43, 45] and [41]. The techniques of [44, 43, 45] are centered on static inference of resource usage of first-order functional programs. None of this work can handle programs whose resource usage depends on integer arithmetic operations, but only on the manipulation of data-structures such as lists and trees. While these techniques can be adapted to handle *CRs* with simple integer linear constraints (like those induced by manipulating data-structures), it is not clear how it can be extended to handle *CRs* with unrestricted form of integer linear arithmetic, in particular with negative coefficients. It is also important to note that these techniques cannot compute *logarithmic* or *exponential* UBs. Overall, we believe that our approach is more generic.

The work of [41] in the SPEED project computes *worst-case* symbolic bounds for C++ code containing loops and recursion. Since the underlying cost analysis framework is fundamentally different from ours, it is not possible to formally compare the resulting UBs in all cases. However, by looking at small examples, we can see why our approach can be more precise. For instance, in [41] the worst-case time usage $\sum_{i=1}^{n} i$ is over-approximated by n^2, while our approach is able to obtain the precise solution $\frac{n^2}{2} - \frac{n}{2}$.

Chapter 7

Conclusion

We have presented the first cost analysis framework for concurrent programs. To develop the analysis, we have considered ABS, an OO language based on the notion of concurrent objects which live in a distributed environment with asynchronous communication. Most of our techniques can be applied in a component-based fashion and are also applicable to other concurrent programming languages. In particular, the idea of having equations parametric on the cost centers is of general applicability. The size analysis is tailored for the concurrency primitives of our ABS language, but we believe that similar abstractions could be developed for other languages by finding correspondences between their concurrency primitives. Besides, the accuracy improvements for the field-sensitive extension could be directly applied to other languages.

We have also made important contributions in the field of cost analysis, which are not tied to ABS programs, but rather could be directly used by cost analyzers for other languages. Among them, we would like to highlight the inference of LBs on the resource consumption. Due in part to the difficulty of inferring under-approximations, a general framework for inferring LBs from *CRs* did not exist. Such LBs are typically useful in granularity analysis to decide whether tasks should be executed in parallel. This is because the parallel execution of a task incurs various overheads, and therefore the LB cost of the task can be useful to decide if it is worth executing it concurrently as a separate task. To the best of our knowledge, we have presented the first general approach to inferring LBs from *CRs*.

As another general contribution, 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 UBs of RRs. 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. We have also studied the verification problem within cost analysis in which one tries to verify (i.e., prove or disprove) *assertions* about the efficiency of the program w.r.t. the cost functions inferred by the analysis.

Finally, we have demonstrated that automatic verification of the UBs inferred by COSTA using KeY is feasible. This approach, though weaker in principle than verification of the analyzer has advantages in the context of mobile code. Following proof-carrying-code [54] principles, code originating from an untrusted *producer* can be bundled together with the proof generated by KeY for its declared resource consumption. This way, the code *consumer* can check locally and automatically using KeY whether the claimed resource guarantees are verified.

Bibliography

[1] A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and inferring local non-aliasing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, pages 129–140. ACM, 2003.

[2] E. Albert, P. Arenas, S. Genaim, Germán Puebla, and Diana Ramírez. From object fields to local variables: a practical approach to field-sensitive analysis. In *Static Analysis Symposium (SAS'10)*, volume 6337 of *Lecture Notes in Computer Science*, pages 100–116. Springer-Verlag, 2010.

[3] E. Albert, P. Arenas, S. Genaim, and D. Zanardini. Task-level analysis for a language with async-finish parallelism. In *Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES'11)*. ACM Press, 2011. To Appear.

[4] E. Albert, R. Bubel, S. Genaim, R. Hähnle, G. Puebla, and G. Román-Díez. Verified resource guarantees using COSTA and KeY. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM'11)*. ACM Press, 2011. To Appear.

[5] Elvira Albert, Diego Alonso, Puri Arenas, Samir Genaim, and Germán Puebla. Asymptotic resource usage bounds. In *Asian Programming Languages and Systems Symposium (APLAS'09)*, volume 5904 of *Lecture Notes in Computer Science*, pages 294–310. Springer-Verlag, 2009.

[6] Elvira Albert, Puri Arenas, Samir Genaim, Israel Herraiz, and Germán Puebla. Comparing cost functions in resource analysis. In *International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA'09)*, volume 6324 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, 2009.

[7] Elvira Albert, Puri Arenas, Samir Genaim, and German Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In *Static Analysis Symposium (SAS'08)*, volume 5079 of *Lecture Notes in Computer Science*, pages 221–237. Springer-Verlag, 2008.

[8] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Field-sensitive value analysis by field-insensitive analysis. In *Formal Methods (FM'09)*, volume 5850 of *Lecture Notes in Computer Science*, pages 370–386. Springer-Verlag, 2009.

[9] Elvira Albert, Puri Arenas, Samir Genaim, and German Puebla. Closed-form upper bounds in static cost analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011.

[10] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Cost analysis of Java bytecode. In *European Symposium on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer-Verlag, 2007.

[11] 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 *Formal Methods for Components and Objects (FMCO'07)*, volume 5382 of *Lecture Notes in Computer Science*, pages 113–132. Springer-Verlag, 2008.

[12] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Resource usage analysis and its application to resource certification. In *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-Verlag, 2009.

[13] Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. Live heap space analysis for languages with garbage collection. In *International Symposium on Memory Management (ISMM'09)*, pages 129–138. ACM Press, 2009.

[14] Elvira Albert, Samir Genaim, and Abu Naser Masud. More precise yet widely applicable cost analysis. In *Verification, Model Checking and Abstract Interpretation (VMCAI'11)*, Lecture Notes in Computer Science. Springer-Verlag, 2011. To Appear.

[15] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.

[16] R. Bagnara, E. Ricci, E. Zaffanella, and P.M. Hill. Possibly not closed convex polyhedra and the Parma polyhedra library. In *Static Analysis Symposium (SAS'02)*, volume 2477 of *Lecture Notes in Computer Science*, pages 213–229. Springer-Verlag, 2002.

[17] Bernhard Beckert, Reiner Hähnle, and Peter Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[18] Amir M. Ben-Amram. Size-change termination, monotonicity constraints and ranking functions. In *Conference on Computer Aided Verification (CAV'09)*, volume 5643 of *Lecture Notes in Computer Science*, pages 109–123. Springer-Verlag, 2009.

[19] F. Benoy and A. King. Inferring argument size relationships with CLP(R). In *Logic-Based Program Synthesis and Transformation (LOPSTR'97)*, volume 1207 of *Lecture Notes in Computer Science*, pages 204–223. Springer-Verlag, 1997.

[20] F. Besson and T. Jensen. Modular class analysis with datalog. In *Static Analysis Symposium (SAS'03)*, number 2694 in Lecture Notes in Computer Science, pages 19–36. Springer-Verlag, 2003.

[21] A. Bossi, N. Cocco, and M. Fabris. Proving Termination of Logic Programs by Exploiting Term Properties. In *Theory and Practice of Software Development (TAPSOFT'91)*, volume 494 of *Lecture Notes in Computer Science*, pages 153–180. Springer-Verlag, 1991.

[22] V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric Prediction of Heap Memory Requirements. In *International Symposium on Memory Management (ISMM'08)*, pages 141–150. ACM Press, 2008.

[23] P. Feautrier C. Alias, A. Darte and L. Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *Static Analysis Symposium (SAS'10)*, volume 6337 of *Lecture Notes in Computer Science*, pages 117–133. Springer-Verlag, 2010.

[24] D. Caromel, L. Henrio, and B. P. Serpette. Asynchronous and deterministic objects. In *Principles of Programming Languages (POPL'04)*, pages 123–134. Association of Computing Machinery, 2004.

[25] P. Charles, C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'05)*, pages 519–538. ACM, 2005.

[26] W-N. Chin, H.H. Nguyen, C. Popeea, and S. Qin. Analysing memory resource bounds for low-level programs. In *International Symposium on Memory Management (ISMM'08)*, pages 151–160. ACM Press, 2008.

[27] M. Codish, S. K. Debray, and R. Giacobazzi. Compositional analysis of modular logic programs. In *Principles of Programming Languages (POPL'93)*, pages 451–464. ACM, 1993.

[28] B. Cook, A. Podelski, and A. Rybalchenko. Proving thread termination. In *Programming Languages Design and Implementation (PLDI'07)*, pages 320–330. ACM, 2007.

[29] J. Correas, G. Puebla, M. Hermenegildo, and F. Bueno. Experiments in context-sensitive analysis of modular programs. In *Logic-based Program Synthesis and Transformation (LOPSTR'05)*, number 3901 in Lecture Notes in Computer Science, pages 163–178. Springer-Verlag, 2006.

[30] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL'77)*, pages 238–252. ACM Press, 1977.

[31] P. Cousot and R. Cousot. Modular static program analysis, invited paper. In *International Conference on Compiler Construction (CC'02)*, number 2304 in Lecture Notes in Computer Science, pages 159–178. Springer-Verlag, 2002.

[32] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Principles of Programming Languages (POPL'78)*, pages 84–97. ACM Press, 1978.

[33] K. Crary and S. Weirich. Resource Bound Certification. In *Principles of Programming Languages (POPL'00)*, pages 184–198. ACM Press, 2000.

[34] Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In *European Symposium on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer-Verlag, 2007.

[35] Report on the Core ABS Language and Methodology: Part A, March 2010. Part of Deliverable 1.1 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[36] G. Puebla et al. A generic framework for context-sensitive analysis of modular programs. In M. Bruynooghe and K. Lau, editors, *Program Development in Computational Logic, A Decade of Research Advances in Logic-Based Program Development*, number 3049 in Lecture Notes in Computer Science, pages 234–261. Springer-Verlag, 2004.

[37] M. Fähndrich. Static verification for code contracts. In *Static Analysis Symposium (SAS'10)*, volume 6337 of *Lecture Notes in Computer Science*, pages 2–5. Springer-Verlag, 2010.

[38] P. Feautrier. Parametric Integer Programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[39] S. Genaim and D. Zanardini. The acyclicity inference of COSTA. In *Workshop on Termination (WST'10)*, July 2010.

[40] B. S. Gulavani and S. Gulwani. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *Conference on Computer Aided Verification (CAV'08)*, volume 5123 of *Lecture Notes in Computer Science*, pages 370–384. Springer-Verlag, 2008.

[41] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: Precise and efficient static estimation of program computational complexity. In *Principles of Programming Languages (POPL'09)*, pages 127–139. ACM, 2009.

[42] Reiner Hähnle. HATS: highly adaptable and trustworthy software using formal models. In Tiziana Margeria and Bernhard Steffen, editors, *Proceedings 4th International Symposium On Leveraging Applications of Formal Methods (ISoLA), Part II, Verification and Validation, Heraclion, Crete*, volume 6416 of *Lecture Notes in Computer Science*, pages 2–7. Springer-Verlag, 2010.

[43] J. Hoffmann and M. Hofmann. Amortized resource analysis with polynomial potential. In *European Symposium on Programming (ESOP'10)*, volume 6012 of *Lecture Notes in Computer Science*, pages 287–306. Springer-Verlag, 2010.

[44] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Principles of Programming Languages (POPL'03)*, pages 185–197. ACM Press, 2003.

[45] Martin Hofmann Jan Hoffmann, Klaus Aehlig. Multivariate amortized resource analysis. In *Principles of Programming Languages (POPL'11)*, pages 357–370. ACM, 2011.

[46] Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. *Software and System Modeling*, 6(1):35–58, March 2007.

[47] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, New York, 1993.

[48] M. Kero, P. Pietrzak, and Nordlander J. Live heap space bounds for real-time systems. In *Asian Programming Languages and Systems Symposium (APLAS'10)*, volume 6461 of *Lecture Notes in Computer Science*, pages 287–303. Springer-Verlag, 2010.

[49] S. Kirkpatrick, Jr. C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[50] J. K. Lee and J. Palsberg. Featherweight X10: A core calculus for async-finish parallelism. In *ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'10)*, pages 25–36. ACM, 2010.

[51] Francesco Logozzo. Separate compositional analysis of class-based object-oriented languages. In *Algebraic Methodology and Software Technology (AMAST'04)*, volume 3116 of *Lecture Notes in Computer Science*, pages 332–346. Springer-Verlag, 2004.

[52] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2nd edition, 1997.

[53] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. User-Definable Resource Usage Bounds Analysis for Java Bytecode. In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'09)*, volume 253:5 of *Electronic Notes in Theoretical Computer Science*, pages 6–86. Elsevier - North Holland, March 2009.

[54] G. Necula. Proof-Carrying Code. In *Principles of Programming Languages (POPL'97)*, pages 106–119. ACM Press, 1997.

[55] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer-Verlag, 1998.

[56] Christian W. Probst. Modular control flow analysis for libraries. In *Static Analysis Symposium (SAS'02)*, volume 2477 of *Lecture Notes in Computer Science*, pages 165–179. Springer-Verlag, 2002.

[57] S. A. Jarvis R. G. Morgan. Profiling large-scale lazy functional programs. *Journal of Functional Programing*, 8(3):201–237, May 1998.

[58] Diana Ramírez-Deantes, Jesús Correas, and Germán Puebla. Modular termination analysis of Java bytecode and its application to phoneme core libraries. In *International Workshop on Formal Aspects of Component Software (FACS'10)*, Lecture Notes in Computer Science. Springer-Verlag, September 2010. To appear.

[59] Jan Schäfer and Arnd Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *European Conference on Object-Oriented Programming (ECOOP'10)*, volume 6183 of *Lecture Notes in Computer Science*, pages 275–299. Springer-Verlag, June 2010.

[60] F. Spoto, F. Mesnard, and É. Payet. A termination analyser for Java bytecode based on path-length. *ACM Transactions on Programming Languages and Systems*, 32(3), 2010.

[61] S. Srinivasan and A. Mycroft. Kilim: Isolation-Typed Actors for Java. In *European Conference on Object-Oriented Programming (ECOOP'08)*, volume 5142 of *Lecture Notes in Computer Science*, pages 104–128. Springer-Verlag, 2008.

[62] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *1999 conference of the Centre for Advanced Studies on Collaborative Research (CASCON'99)*, pages 125–135, 1999.

[63] Ben Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9), 1975.

Glossary

ABS Abstract Behavioral Specification language. An executable class-based, concurrent, object-oriented modeling language based on Creol, created for the HATS project.

CAS Computer Algebra System.

CFG Control Flow Graph.

COSTA Cost and Termination Analyzer.

COSTABS Cost and Termination Analyzer for ABS programs.

CPU Central Processing Unit.

CR Cost Relation.

CRs Cost Relations.

CRS Cost Relation System.

IO Input-Output.

IR Intermediate Representation.

JCoBox A Java extension with an actor-like concurrency model based on the notion of concurrently running object groups, so-called coboxes.

JML Java Modelling Language.

KeY The KeY System is a formal software development tool that aims to integrate design, implementation, formal specification, and formal verification of object-oriented software as seamlessly as possible.

LB Lower Bound.

OO Object Oriented.

PIP Parametric Integer Programming.

P2P Peer to Peer.

RR Recurrence Relation.

SCC Strongly Connected Component.

UB Upper Bound.

WCET Worst Cost Execution Time.

Appendix A

Task-Level Analysis for a Language with `async-finish` Parallelism

The paper "Task-Level Analysis for a Language with `async-finish` Parallelism" [3] follows.

Task-Level Analysis for a Language with `async`/`finish` Parallelism

Elvira Albert Puri Arenas Samir Genaim
SIC, Complutense University of Madrid, E-28040 Madrid, Spain
{elvira,puri,samir}@clip.dia.fi.upm.es

Damiano Zanardini
Technical University of Madrid, E-28660 Boadilla del Monte, Madrid, Spain
{damiano}@clip.dia.fi.upm.es

Abstract

The *task level* of a program is the maximum number of tasks that can be *available* (i.e., not finished nor suspended) simultaneously during its execution for any input data. Static knowledge of the task level is of utmost importance for understanding and debugging parallel programs as well as for guiding task schedulers. We present, to the best of our knowledge, the first static analysis which infers safe and precise approximations on the task level for a language with `async-finish` parallelism. In parallel languages, `async` and `finish` are basic constructs for, respectively, spawning tasks and waiting until they terminate. They are the core of modern, parallel, distributed languages like X10. Given a (parallel) program, our analysis returns a *task-level upper bound*, i.e., a function on the program's input arguments that guarantees that the task level of the program will never exceed its value along any execution. Our analysis provides a series of useful (over)approximations, going from the total number of tasks spawned in the execution up to an accurate estimation of the task level.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: [Concurrent Programming] *Distributed programming, Parallel programming*; D.3 [*Programming Languages*]: [Formal Definitions and Theory]

General Terms Algorithms, Languages, Theory, Verification

Keywords Parallelism, Static Analysis, Resource Consumption, X10, Java

1. Introduction

As embedded systems increase in number, complexity, and diversity, there is an increasing need of exploiting new hardware architectures, and scaling up to multicores and distributed systems built from multicores. This brings to the embedded-systems area wide interest in developing techniques that help in understanding, optimizing, debugging, finding optimal schedulings for parallel programs. Two of the key constructs for parallelism are `async` and `finish`. The `async`{s} statement is a notation for spawning tasks: the task s can run in parallel with any statements following it. The `finish`{s} statement waits for termination of s and of all `async` statement bodies started while executing s. In order to develop our analysis, we consider a Turing-complete language with a minimal syntax and a simple formal semantics. A program consists of a collection of methods that all have access to a shared array. The body of the method is a sequence of statements. Each statement can be an assignment, conditional, loop, `async`, `finish`, or method call. If we add some boilerplate syntax to a program, the result is an executable X10 program. X10 [5] is a modern, parallel, distributed language intended to be very easily accessible to Java programmers. Note that, unlike in languages based on `fork` and `join` constructs, `async-finish` programs are guaranteed to be deadlock-free [16].

Our objective is to present a clear and concise formalization of the analysis on a simple imperative language that captures the essence of standard parallelism constructs.

As our main contribution, we present a novel static analysis which infers the *task level* of a parallel program, i.e., the maximum number of *available* tasks (i.e., not finished nor suspended) which can be run simultaneously along any execution of the program (provided that sufficient computing resources are available). A starting point for our work is the observation that spawning parallel tasks can be regarded as a *resource* consumed by a program. This leads to the idea of adapting powerful techniques developed in resource analysis frameworks for *sequential* programs [2, 9, 10, 21] in order to obtain sound task-level Upper Bounds (UBs) on *parallel* programs. Such adaptation is far from trivial since, among other things, the task level of a program is not an *accumulative* resource, but rather it can increase and/or decrease along the execution. This renders direct application of cost analysis frameworks for accumulative resources (such as time, total memory consumption, etc.) unsuitable. We present our novel analysis to accurately (over)-approximate the task level of a program in the following steps, which are the main contributions of this work.

1. We first produce over-approximations of the *total* number of tasks spawned along any execution of the program. This can be done by lifting existing accumulative cost analyses developed for sequential programs to the parallel setting. The results of such analysis are sound w.r.t. any particular scheduling of tasks.
2. Secondly, we present a novel approach to approximate the *peak* (or maximum) of *alive* tasks, a resource which is not accumulative. The challenge here is to come up with a technique which over-approximates the peak of alive tasks among all possible states that might occur in any execution of the program.
3. As a further step, we refine the previous approach and approximate the peak of *available* tasks, i.e., we exclude those tasks which are alive but suspended, thus resulting in smaller UBs.

LCTES'11, April 11–14, 2011, Chicago, Illinois, USA.

4. Then, we show how our task-level analysis can be improved by first inferring the tasks *escaped* from a method m, i.e., those tasks that have been created during the execution of m and can be available on return from m. This improvement requires a more complex analysis but, when doable, leads to strictly more precise bounds than those in points 2 and 3.
5. We report on a prototype implementation and experimentally evaluate it on a set of simplified versions of applications from the X10 website that contain interesting parallelism patterns.

2. Motivating Examples

In this section, we introduce by means of examples the main notions that our task-level analysis over-approximates, and show the results it can produce. These examples will also illustrate the following applications of our analysis: (1) It is useful for both understanding and debugging parallel programs. For instance, our analysis can infer a task-level upper bound which is larger (or smaller) than the programmer's expectations. This can clearly help finding bugs in the program. Even more, the analysis results would be "unbounded" when an instruction which spawns new tasks is (wrongly) placed within a loop which does not terminate. (2) The results are also useful for optimizing and finding optimal deployment configurations. For example, when parallelizing the program, it is not profitable to have more processors than the inferred task level.

2.1 Total Tasks versus Alive Tasks

The first example implements a parallel version of the Gauss elimination algorithm. An invocation `gaussian(n)` applies the algorithm on the matrix defined by the elements (i, j) where $0 \leq i, j \leq n-1$. It assumes that the two-dimensional array `A` is initialized with integer values.

```
1  int A[][];
2
3  void gaussian(int n) {
4    for(int k=0;k<n;k++) {
5
6      finish for(int j=k+1;j<n;j++) async {
7        A[k,j]=A[k,j]/A[k,k];
8      }
9
10     finish for(int i=k+1;i<n;i++) async {
11       for(int j=k+1;j<n;j++)
12           A[i,j]:=A[i,j]-A[i,k]*A[k,j]
13     }
14   }
15 }
```

The total number of tasks spawned by this method is quadratic on n, since at each iteration of the outer loop, each of the inner loops spawn (in the worst case) $n-1$ tasks. Note that the loop at line 11 does not spawn any task. Our analysis accurately infers that at most $1 + 2n(n-1)$ tasks will be spawned along an execution.

Due to the use of `finish` at line 6, it is ensured that before entering the loop at line 10, all asynchronous tasks spawned at line 6 are finished. Likewise, all asynchronous tasks spawned by the loop at line 10 must be finished before starting the next iteration of the outer loop. Hence, in any execution of this program, the maximum number of tasks that can be alive simultaneously (or *peak of alive tasks*) corresponds to the maximum of the tasks spawned by the loops at lines 6 and 10. Our analysis precisely infers that the peak of alive tasks is n.

Observe that both the total and the peak number of alive tasks are useful pieces of information for the programmer. E.g., by comparing the inferred upper bounds with the programmer's expectations we might detect bugs, as explained above.

2.2 Alive Tasks versus Available Tasks

The following method implements the merge-sort algorithm. It sorts the elements of the array `A` between the indexes `from` and `to`. We omit the code of `merge` and only assume that it does not spawn further tasks.

```
1  int A[];
2
3  void msort(int from, int to) {
4    if ( from < to ) {
5      mid=(from+to)/2;
6      finish {
7        async msort(from,mid);
8        async msort(mid+1,to);
9      }
10     merge(from,to,mid);
11   }
12 }
```

The total number of tasks spawned by a call `msort(from,to)` is bounded by $2 * (to - from + 1) - 2$. This upper bound is obtained by proving that, in both recursive calls, the number of elements to be sorted is decreased by half and, at each recursive call, two new tasks are spawned.

For this example, we infer that the peak of alive tasks and the total number of tasks are identical. This is because the recursive calls are performed within the scope of the `finish` construct. Thus, in the worst case, all tasks can be alive simultaneously (though the current task always blocks after launching the asynchronous calls). We can improve the analysis result by proving that, at each recursive call, after spawning the two asynchronous tasks, the current process becomes *inactive* by suspending its execution until the spawned tasks terminate. By taking advantage of this knowledge, our analysis accurately infers that the peak of *available* tasks is at most $to - from + 1$, which is almost half of the one we obtained for alive tasks.

2.3 Improving Available Tasks with Escape Information

Next example is a pre-order traversal of a binary tree where, for each node i, we spawn two tasks: `activity_a(i)` and `activity_b(i)`. We omit the code of `activity_a` and `activity_b` and only assume that they do not spawn further tasks. The binary tree is represented using the array `A`, such that the nodes at positions `2*i+1` and `2*i+2`, respectively, are the left and right children of the node at position `i`. The first argument `n` is the depth of the tree. The method is supposed to be called with `f(n,0)`.

```
1  int A[];
2
3  void f(int n, int i) {
4    if ( n > 0 ) {
5      finish {
6        async activity_a(i);
7        async activity_b(i);
8      }
9      f(n-1,2*i+1);
10     f(n-1,2*i+2);
11   }
12 }
```

By accumulating all asynchronous calls spawned along the execution, our analysis generates the upper bound $2 * (2^n - 1) + 1$. As expected, the obtained bound is exponential on the depth of the tree due to the two recursive calls which traverse all nodes in the tree. For the peak of available tasks, we can greatly improve the task-level bound. In particular, we can see that the asynchronous calls in lines 6 and 7 will be finished at line 8 before the recursive calls. This means that, given a call to `f`, there are no tasks that *escape* from its scope, i.e., all tasks created during a call to `f` (di-

rectly or transitively) are terminated before the execution of the call finishes. The use of escape information during our analysis allows proving that there cannot be more than 2 processes simultaneously available. From the above examples, it should become clear that an upper bound on the available tasks can be of utmost importance for finding an optimal deployment configuration. For instance, in the above example, it is not worth having more than 2 processors when executing the code.

3. A Simplified X10-like Language

We develop our analysis on a representative subset of X10 [5], a parallel language which is similar to Java in its sequential part, and relies on the `async`/`finish` mechanism for parallelism. From X10, we take:

- a Turing-complete core consisting of conditionals, loops, assignments and a single one-dimensional array;
- methods and method calls;
- the `async` and `finish` statements.

We omit many features from X10, including places, distributions and clocks. Indeed, our simplified language is very similar to Featherweight X10 [13] (FX10 for short), a subset of X10 which has been proposed to develop formal analyses on X10. For the sake of expressiveness, our language is richer than FX10 in that it allows input parameters in method calls (in order to handle recursion), has no restriction on conditional statements and has local variables. The treatment of object fields is similar to (and simpler than) the treatment of array accesses; details are omitted for simplicity.

3.1 The Recursive Intermediate Representation

As customary in the formalization of static analyses for realistic languages, we develop our analysis on an intermediate representation (IR) of the language which allows us to provide a clearer and more concise formalization. Similar representations are used by other static analysis tools for Java (and Java bytecode) and .NET [2, 7, 8, 18, 22]. Essentially, all these tools work by first building the control flow graph (CFG) from the program, and then representing each block of the CFG in the intermediate language (in our IR, by means of rules).

Methods in the original program are represented by one or more *procedures* in the IR. A procedure is defined by one or more *guarded rules*. The translation is as follows. Given a method, each block in its CFG is represented by means of a guarded rule. Guards state the conditions under which the corresponding block can be executed (they contain the conditions in the edges of the CFG). Each rule contains as arguments those variables that are input values to the block. When the block has more than one successor in the CFG, we just create a *continuation procedure* and a corresponding call in the rule. Blocks in the continuation will be in turn defined by means of guarded rules (with mutually exclusive conditions). As a result, all forms of iteration in the program are represented by means of *recursive* calls. The array remains as a global variable in the IR. The process of obtaining the intermediate representation from X10-like programs is completely automatic. Since it is identical as for Java programs, we will not go into the technical details of the transformation (we refer to any of [2, 7, 8, 18, 22]) but just show the intuition by means of an example.

EXAMPLE 3.1. *Fig. 1 shows the intermediate representation for the example in Sec. 2.1. This example shows an interesting aspect of the IR: loops are detected and extracted in separate procedures as described in [19]. It can be observed that within the rule $gaussian$ we invoke procedure for, which corresponds to the for-loop in line 4. Similarly, when entering the remaining for-loops in the program, we have calls in the IR to corresponding procedures defining them. By looking at the two rules defining procedure for, we observe the more interesting aspects of our IR: (1) rules contain as arguments those variables that are input values to the scope of the loop; (2) by means of guards, we distinguish the case of exiting the loop (first rule) and entering the loop (second rule); (3) iteration is transformed into recursion. Note that, in the IR, the `finish` construct is applied on a single instruction. If there are several instructions within the scope of `finish` in the original program, we just create an auxiliary procedure which contains them all.*

$gaussian(n) \leftarrow k{:=}0, for(k,n)$
$for(k,n) \leftarrow k \geq n$
$for(k,n) \leftarrow k < n,$
$\quad j{:=}k+1,$
$\quad \texttt{finish}\{for_1(k,n,j)\}, i{:=}k+1,$
$\quad \texttt{finish}\{for_2(k,n,i)\},$
$\quad k'{:=}k+1, for(k',n)$
$for_1(k,n,j) \leftarrow j \geq n$
$for_1(k,n,j) \leftarrow j < n,$
$\quad \texttt{async}\{op_1(k,j)\}$
$\quad j'{:=}j+1, for_1(k,n,j')$

$for_2(k,n,i) \leftarrow i \geq n$
$for_2(k,n,i) \leftarrow i < n,$
$\quad j{:=}k+1$
$\quad \texttt{async}\{for_{2.1}(k,n,j,i)\},$
$\quad i'{:=}i+1,$
$\quad for_2(k,n,i')$
$for_{2.1}(k,n,j,i) \leftarrow j \geq n$
$for_{2.1}(k,n,j,i) \leftarrow j \leq n,$
$\quad op_2(k,j,i),$
$\quad j'{:=}j+1,$
$\quad for_{2.1}(k,n,j',i)$

Figure 1. Recursive Intermediate Representation of Ex. in Sec. 2.1

3.2 Syntax

A *program* in our intermediate representation consists of a set of *procedures*, each of them defined by one or more *guarded rules*. In the following, given any entity a, we use $\bar{a}$ to denote the sequence $a_1, \ldots, a_n$, $n \geq 1$. A procedure p with k input arguments $\bar{x}$ is defined by rules which adhere to this grammar:

$$
\begin{array}{rl}
\mathit{rule} ::= & p(\bar{x}) \leftarrow g, b, \ldots, b \\
g ::= & \mathit{true} \mid \mathit{exp}\ \mathit{op}\ \mathit{exp} \\
b ::= & y{:=}\mathit{exp} \mid \mathtt{A}[y]{:=}\mathit{exp} \mid q(\bar{x}) \mid \mathtt{async}\{q(\bar{x})\} \mid \mathtt{finish}\{q(\bar{x})\} \\
\mathit{exp} ::= & y \mid d \mid \mathtt{A}[y] \mid \mathit{exp}-\mathit{exp} \mid \mathit{exp}+\mathit{exp} \mid \mathit{exp}*\mathit{exp} \mid \mathit{exp}/\mathit{exp} \\
\mathit{op} ::= & > \mid < \mid \leq \mid \geq \mid = \mid \neq
\end{array}
$$

where $p(\bar{x})$ is the *head* of the rule; g is its guard, which specifies conditions for the rule to be applicable; the sequence $b, \ldots, b$ is the body of the rule; d is an integer; y is a variable name; $q(\bar{x})$ is a procedure call, $\mathtt{async}\{q(\bar{x})\}$ is an asynchronous procedure call, and $\mathtt{finish}\{q(\bar{x})\}$ is a synchronized call. All variables are of type integer. Computations work on a single shared memory given by a one-dimensional array of integer values named `A` with indexes $0 \ldots \mathtt{N}-1$. When the execution begins, input values are loaded into all elements of the array. Thus, the array `A` is fully initialized for all indices $0 \ldots \mathtt{N}-1$. In the examples, to simplify the presentation, we use several (possibly multidimensional) arrays.

3.3 Semantics

Fig. 2 shows the *operational semantics* for X10-like programs in the IR. It adapts the small-step operational semantics of FX10 [13] to our syntax and extends it to handle the additional language features discussed in the beginning of the section. It uses the binary operator $\parallel$ in the semantics of `async` and $\triangleright$ in `finish`. A *state* is a pair, consisting of the state of the array and a tree which describes the code executing. Namely, it is of the form $(\mathtt{A}\ ;\ T)$ where $\mathtt{A} : \{0, \ldots, \mathtt{N}-1\} \mapsto \mathbb{Z}$ is an array of integers and T is an execution tree defined by the following grammar:

$$T ::= \ T \triangleright T \mid T \parallel T \mid \langle id, \mathit{instr}, tv \rangle$$

where $id \in \mathbb{N}$ is a unique task identifier, instr is a sequence of instructions (as in Sec. 3.2) and $tv{:}\mathcal{V}{\mapsto}\mathbb{Z}$ is a partial map from the set of variable names $\mathcal{V}$ to integers. The symbol ϵ denotes an empty sequence of instructions. We refer to the tuple $\langle id, \bar{b}, tv \rangle$ as a record.

$$(1)\ \frac{}{(\mathtt{A}\ ;\ \langle id,\epsilon,tv\rangle \triangleright T) \to (\mathtt{A}\ ;\ T)} \qquad (2)\ \frac{(\mathtt{A}\ ;\ T_1) \to (\mathtt{A}'\ ;\ T_1')}{(\mathtt{A}\ ;\ T_1 \triangleright T_2) \to (\mathtt{A}'\ ;\ T_1' \triangleright T_2)}$$

$$(3)\ \frac{}{(\mathtt{A}\ ;\ \langle id,\epsilon,tv\rangle \parallel T) \to (\mathtt{A}\ ;\ T)} \qquad (4)\ \frac{}{(\mathtt{A}\ ;\ T \parallel \langle id,\epsilon,tv\rangle) \to (\mathtt{A}\ ;\ T)}$$

$$(5)\ \frac{(\mathtt{A}\ ;\ T_1) \to (\mathtt{A}'\ ;\ T_1')}{(\mathtt{A}\ ;\ T_1 \parallel T_2) \to (\mathtt{A}'\ ;\ T_1' \parallel T_2)} \qquad (6)\ \frac{(\mathtt{A}\ ;\ T_2) \to (\mathtt{A}'\ ;\ T_2')}{(\mathtt{A}\ ;\ T_1 \parallel T_2) \to (\mathtt{A}'\ ;\ T_1 \parallel T_2')}$$

$$(7)\ \frac{b \equiv x{:=}exp,\ \ v = eval(exp,tv,\mathtt{A})}{(\mathtt{A}\ ;\ \langle id, b\cdot instr, tv\rangle) \to (\mathtt{A}\ ;\ \langle id, instr, tv[x\mapsto v]\rangle)}$$

$$(8)\ \frac{b \equiv \mathtt{A}[x]{:=}exp,\ v = eval(exp,tv,\mathtt{A}),\ n = tv(x),\ 0 \le n \le \mathtt{N}-1}{(\mathtt{A}\ ;\ \langle id, b\cdot instr, tv\rangle) \to (\mathtt{A}[n\mapsto v]\ ;\ \langle id, instr, tv\rangle)}$$

$$(9)\ \frac{b \equiv q(\bar{x}), r \equiv q(\bar{x}') \leftarrow g, b_1, \cdots, b_k \ll_{tv} P,\ \ eval(g,tv',\mathtt{A}) \equiv true \quad tv'{=}tv \cup \{x_i' \mapsto tv(x_i) \mid \forall x_i' \in \bar{x}'\} \cup \{y \mapsto 0 \mid \forall y \in vars(r)\setminus \bar{x}'\}}{(\mathtt{A}\ ;\ \langle id, b\cdot instr, tv\rangle) \to (\mathtt{A}\ ;\ \langle id, b_1\cdots b_k\cdot instr, tv'\rangle)}$$

$$(10)\ \frac{b \equiv \mathtt{async}\{q(\bar{x})\},\ \ id' \text{ is a new identifier not used before}}{(\mathtt{A}\ ;\ \langle id, b\cdot instr, tv\rangle) \to (\mathtt{A}\ ;\ \langle id', q(\bar{x}), tv\rangle \parallel \langle id, instr, tv\rangle)}$$

$$(11)\ \frac{b \equiv \mathtt{finish}\{q(\bar{x})\}}{(\mathtt{A}\ ;\ \langle id, b\cdot instr, tv\rangle) \to (\mathtt{A}\ ;\ \langle id, q(\bar{x}), tv\rangle \triangleright \langle id, instr, tv\rangle)}$$

Figure 2. Operational semantics

Executions start from an *initial state* of the form $(\mathtt{A}\ ;\ \langle 1, p(\bar{x}), tv\rangle)$, where p is the entry procedure name, and the elements of the array $\mathtt{A}$ and $tv(x_i)$ for all $x_i \in \bar{x}$ are initialized to some initial values. We often view tv as a set $\{x_1 \mapsto v_1, \cdots, x_n \mapsto v_n\}$ where each x_i is a variable name and each v_i is an integer value. Executions are regarded as *traces* of the form $(\mathtt{A}_0\ ;\ T_0) \to (\mathtt{A}_1\ ;\ T_1) \to \cdots \to (\mathtt{A}_n\ ;\ T_n)$, sometimes denoted as $(\mathtt{A}_0\ ;\ T_0) \to^* (\mathtt{A}_n\ ;\ T_n)$. Infinite traces correspond to non-terminating executions. We say that a call to a procedure *locally* terminates if the execution of its procedure's body terminates, and we say that it *globally* terminates if, in addition, all tasks it spawns terminate.

The left side of Fig. 2 contains the rules for dealing with parallelism and synchronization. A tree $T_1 \triangleright T_2$ gives the semantics of the `finish` statement. As shown in rule (2), T_1 must complete execution before moving on to executing T_2, i.e., T_1 must be reduced to $\langle id, \epsilon, tv\rangle$ in order to apply rule (1). Rules (3) and (4) remove trees whose evaluation is completely finished whereas (5) and (6) allow choosing trees T_1 or T_2 non-deterministically (i.e., there is no assumption on the task scheduler).

The right side of Fig. 2 contains the rules for executing instructions. Intuitively, rule (7) accounts for all instructions in the semantics which perform arithmetic and assignment operations. We assume that $eval(exp, tv, \mathtt{A})$ returns the evaluation of the arithmetic expression exp using the values of the corresponding variables from tv and $\mathtt{A}$ in the standard way. Moreover, we assume that it fails when trying to access $\mathtt{A}$ with an index which is not in the range $0 \ldots \mathtt{N}-1$. Rule (8) deals with assignments on $\mathtt{A}$. After evaluating exp, the resulting value is stored in the position $tv(x)$ of $\mathtt{A}$. Rule (9) corresponds to invoking a procedure $q(\bar{x})$. It first takes a rule r for q. The notation $\ll_{tv}$ means that we rename the rule variables so they will not clash with names already in the domain of tv. Then, we generate a new variable mapping tv' which extends tv by initializing the formal parameters $\bar{x}'$ with the values of the actual parameters $\bar{x}$, and the remaining variables not in $\bar{x}$ (i.e., $vars(r)\setminus\bar{x}$) to 0. We require that the guard g of rule r is evaluated to $true$ (as usual, the values $true$ and $false$ can be simulated with 0 and non-0 integers). Rule (10) takes care of the `async` statement by spawning a new task to be executed in parallel. Finally, rule (11) introduces the operator $\triangleright$ to wait for the termination of the task, when we have a `finish` instruction.

EXAMPLE 3.2. *As an example of how the semantics works, consider the following simple program. For brevity, we ignore the code of procedures $q_1, \ldots, q_5$ and assume that they neither make directly or indirectly any asynchronous call, nor modify the array.*

$p \leftarrow \mathtt{async}\{q_1\}, \mathtt{finish}\{q\}, ①\mathtt{async}\{q_2\}, q_3$
$q \leftarrow \mathtt{async}\{q_4\}, \mathtt{async}\{q_5\}$

The following derivation starts from the entry procedure p:

$(\mathtt{A}\ ;\ \langle 1, p, tv\rangle) \to$
$(\mathtt{A}\ ;\ \langle 1, \mathtt{async}\{q_1\}\cdot\mathtt{finish}\{q\}\cdot\mathtt{async}\{q_2\}\cdot q_3, tv\rangle) \to$
$*_1$ $(\mathtt{A}\ ;\ \langle 2, q_1, tv\rangle \parallel \langle 1, \mathtt{finish}\{q\}\cdot\mathtt{async}\{q_2\}\cdot q_3, tv\rangle) \to$
$*_2$ $(\mathtt{A}\ ;\ \langle 2, q_1, tv\rangle \parallel \langle 1, q, tv\rangle \triangleright \langle 1, \mathtt{async}\{q_2\}\cdot q_3, tv\rangle) \to$
$(\mathtt{A}\ ;\ \langle 2, q_1, tv\rangle \parallel$
$\langle 1, \mathtt{async}\{q_4\}\cdot\mathtt{async}\{q_5\}, tv\rangle \triangleright \langle 1, \mathtt{async}\{q_2\}\cdot q_3, tv\rangle) \to$
$(\mathtt{A}\ ;\ \langle 2, q_1, tv\rangle \parallel$
$(\langle 3, q_4, tv\rangle \parallel \langle 1, \mathtt{async}\{q_5\}, tv\rangle) \triangleright \langle 1, \mathtt{async}\{q_2\}\cdot q_3, tv\rangle) \to$
$\diamond$ $(\mathtt{A}\ ;\ \langle 2, q_1, tv\rangle \parallel$
$(\langle 3, q_4, tv\rangle \parallel \langle 4, q_5, tv\rangle \parallel \langle 1, \epsilon, tv\rangle) \triangleright \langle 1, \mathtt{async}\{q_2\}\cdot q_3, tv\rangle) \to^*$
$(\mathtt{A}\ ;\ \langle 2, q_1, tv\rangle \parallel \langle 1, \epsilon, tv\rangle \triangleright \langle 1, \mathtt{async}\{q_2\}\cdot q_3, tv\rangle) \to$
$*_3$ $(\mathtt{A}\ ;\ \langle 2, q_1, tv\rangle \parallel \langle 1, \mathtt{async}\{q_2\}\cdot q_3, tv\rangle) \to$
$(\mathtt{A}\ ;\ \langle 2, q_1, tv\rangle \parallel \langle 5, q_2, tv\rangle \parallel \langle 1, q_3, tv\rangle) \to$
$(\mathtt{A}\ ;\ \langle 2, q_1, tv\rangle \parallel \langle 5, q_2, tv\rangle \parallel \langle 1, \epsilon, tv\rangle) \to^* (\mathtt{A}\ ;\ \langle 2, \epsilon, tv\rangle)$

Note that since q_1 is invoked asynchronously, p can continue to the next statement at $*_1$. However, when executing $\mathtt{finish}\{q\}$ at $*_2$, the execution of p blocks until q and its asynchronous sub-tasks q_4 and q_5 terminate, then resumes from the program point ① (step $*_3$).

4. Concrete Definitions in Task Parallelism

We first introduce basic notions related to the task parallelism of a program. They define the notions that later we want to approximate by means of static analysis. First, we introduce two auxiliary definitions to count the number of tasks that can be simultaneously alive at some program point by means of the following function $\mathsf{alive}(T)$ which goes from the set of trees to the set of task identifiers $\wp(\mathbb{N})$:

$$\begin{array}{lcl} \mathsf{alive}(T_1 \parallel T_2) & = & \mathsf{alive}(T_1) \cup \mathsf{alive}(T_2) \\ \mathsf{alive}(T_1 \triangleright T_2) & = & \mathsf{alive}(T_1) \cup \mathsf{alive}(T_2) \\ \mathsf{alive}(\langle id, \epsilon, tv\rangle) & = & \emptyset \\ \mathsf{alive}(\langle id, instr, tv\rangle) & = & \{id\} \end{array}$$

Note that when a task does not have any further instruction to execute (third equation) it is not counted as alive. The above definition includes tasks which are *blocked*, i.e., are not available in the current state. For instance, for $(S_1 \parallel S_2) \triangleright S_3$ such that each S_i has $\mathsf{alive}(S_i) = 1$, the function alive returns 3. However, the semantics of $\triangleright$ ensures that S_3 is blocked, i.e., it remains suspended until the execution of $S_1 \parallel S_2$ finishes. The function $\mathsf{available}$ counts only the available tasks, i.e., alive tasks which are not suspended. It is defined as alive except for $\mathsf{available}(T_1 \triangleright T_2) = \mathsf{available}(T_1)$.
Given a trace $t \equiv T_0 \to T_1 \to \cdots \to T_n$, by relying on the above two functions, we can define the following three important notions that our analysis approximates:

- $\mathsf{total}(t)$. First, we define the *total number of spawned tasks* along an execution, which corresponds to the total number of tasks that have been started, as: $\mathsf{total}(t) = |\cup_{i=0}^{n} \mathsf{alive}(T_i)|$ (here, $|X|$ is the size of the set X). Note that this resource is accumulative, i.e., it always increases as the execution proceeds.
- $\mathsf{peakAlive}(t)$. Another interesting notion is the peak of *alive* tasks, i.e., the maximum number of tasks that are simultaneously started and not finished. The function $\mathsf{peakAlive}(t)$ is defined as $\mathsf{max}(\{|\mathsf{alive}(T_0)|, \ldots, |\mathsf{alive}(T_n)\}|)$. Note that this resource is not accumulative: instead, the number of alive tasks can increase or decrease at any state. Thus, in order to approximate it, we need to observe all states and capture the maximum.
- $\mathsf{peakAvailable}(t)$. Similarly, we can define the peak of *available* tasks along the execution, i.e., the maximum number of tasks that are simultaneously started and not blocked. Thus, $\mathsf{peakAvailable}(t)$=$\mathsf{max}(\{|\mathsf{available}(T_0)|, .., |\mathsf{available}(T_n)|\}$. We also refer to this notion as the *task level* of the execution, the two definitions above being over-approximations of it.

EXAMPLE 4.1. *By applying the above definitions to the derivation of Ex. 3.2, we have:* $\mathsf{total}(t) = 5$, $\mathsf{peakAlive}(t) = 4$ *and* $\mathsf{peakAvailable}(t) = 3$. *Note that the difference between* peakAlive *and* peakAvailable *occurs in the state labeled* $\diamond$. *This is because, after creating the new tasks, the task on which* q *is executing is alive but blocked (hence not available).*

5. Static Inference of Spawned Tasks

In the previous section, our definitions assume a trace. Thus, the program must be executed on a specific input in order to compute them. Now, we want to approximate these notions *statically*, i.e., without executing the program and the results must be valid for any input. In particular, by concentrating on the total number of spawned tasks first, given a method $p(\bar{x})$, the goal of our analysis is to infer $p^{ub}(\bar{x})$, called *task-level UB for* p, which is a function on the input data of p which guarantees that, given any concrete values $\bar{v}$ for $\bar{x}$, the total number of tasks spawned along the trace t resulting from executing $p(\bar{v})$ (i.e., $\mathsf{total}(t)$) is smaller than or equal to $p^{ub}(\bar{v})$ plus one for the main task.

Since the total number of tasks is an accumulative resource, in principle, any of the existing resource analysis frameworks that count a particular form of accumulative resource (e.g., instructions [9], total memory [10, 21], etc.) can be adapted to the total number of spawned tasks by counting the instructions `async` that spawn tasks and ignoring the rest. However, all above approaches assume sequential programs and must be lifted to the parallel setting. This is because, as we will see later, the resulting UB can be affected by the fact that tasks can run in parallel. Among all possible resource analysis frameworks, we rely on the most traditional one, proposed by Wegbreit [24] in 1975. As our first contribution, we adapt such approach to infer sound results on the task level in a parallel setting. The next three subsections present the main steps of the analysis:

- First, we discuss in Sec. 5.1 the value abstraction component which is used to infer inter-relations between the program variables. Interestingly, by losing information about the global data during the value abstraction, we are able to ensure soundness of the overall UBs in the parallel setting.
- Given the value relations, we proceed in Sec. 5.2 to define, for our intermediate language, how to generate the recurrence equations which define the spawned tasks.
- Finally, in Sec. 5.3, we briefly describe the process of obtaining safe over-approximations from the generated equations by relying on existing solvers of recurrence equations (e.g., computer algebra systems like MAXIMA).

5.1 Value Abstraction

Given a rule, we describe how to generate a conjunction of (linear) constraints (sometimes written as a set) that describes the relations between the values of the rule's variables at the different program points. This information is later used, for example, to understand how values change when moving from one procedure to another. In particular, it is essential for bounding the number of recursive calls (i.e., iterations of loops). The following definition presents the notion of *value abstraction* for a given rule. In order to distinguish between the values of a variable at different program points (inside a single rule), rules are given in static single assignment (SSA) form [3] (array accesses remain the same). The rules in Fig. 1 and in all remaining examples are in SSA. This transformation is straightforward for a single rule, as rules do not have branching.

DEFINITION 5.1. *Give a rule* $r \equiv p(\bar{x}) \leftarrow g, b_1 \ldots, b_n$ *in SSA form, its value abstraction is* $\varphi_r = \alpha(g) \wedge \alpha(b_1) \wedge \cdots \wedge \alpha(b_n)$ *where:*

- $\alpha(y{:=}exp) = (y{=}exp)$ *if* exp *is a linear expression which does not involve arrays;*
- $\alpha(exp_1\ op\ exp_2){=}(exp_1\ op\ exp_2)$ *if* $op \in \{>, \geq, <, \leq, =\}$ *and* exp_1 *and* exp_2 *are linear expressions not involving arrays;*
- $\alpha(b) = true$, *otherwise.*

For simplicity, the above abstraction ignores non-linear arithmetic expressions by abstracting the corresponding instructions to *unknown* ($true$). Non-linear arithmetic can be handled at the price of performance using non-linear constraints manipulation techniques.

EXAMPLE 5.2. *Applying Def. 5.1 on the second rule for "*for*" of Fig. 1, we obtain as value abstraction* $\{k{<}n, k'{=}k{+}1, j{=}k', i{=}k'\}$.

An important point in the above abstraction is that we ignore data which resides in the global array `A`. This provides us correctness in the context of parallel execution without requiring any other sophisticated heap analysis for ensuring the *independence* [23] between tasks. Let us see an example.

EXAMPLE 5.3. *Consider the following program and observe that when* m *invokes the two asynchronous calls, procedures* p *and* q *might run in parallel depending on the underlying task scheduler.*

$$
\begin{array}{ll}
m(n) & \leftarrow \texttt{async}\{p(0,n)\}, \texttt{async}\{q(n)\} \\
p(i,n) & \leftarrow i \geq \texttt{A}[n] \\
p(i,n) & \leftarrow i < \texttt{A}[n], \texttt{async}\{q_1\}, i' {:=} i+1, p(i',n) \\
q(n) & \leftarrow \texttt{A}[n] {:=} \texttt{A}[n]+1, q(n)
\end{array}
$$

By looking at a complete execution of p *in isolation (i.e., if it does not interleave with that of* q*), we can see that a sound upper bound on the number of tasks spawned by* p *is* $\texttt{A}[n]$ *(the value of the* n*-the element of the array). However, if the execution of* q *interleaves with that of* p*, the execution of* p *might not terminate since* q *increases the value of* $\texttt{A}[n]$. *Hence, the previous UB is not correct.*

Our practical solution to avoid the above problem is to abstract instructions that involve global data (i.e., array elements) to unknown (i.e., $true$). In the above case, the guard $i < \texttt{A}[n]$ is abstracted to $true$ and thus the value of $\texttt{A}[n]$ is lost. Hence, we will fail to infer an UB for the method. This does not mean that we cannot analyze programs that use the array but rather that, when the UB is a function of an array element, we cannot find it. In Sec. 11, we discuss how to improve the accuracy by relying on a may-happen-in-parallel analysis [13] in combination with a field-sensitive value analysis [14]. It should be noted that the value abstraction is an independent component in our analysis and we can improve it regardless of the next components that we will introduce in what follows. Also, when improving it, we can integrate advanced value abstractions for data structures such as path-length [18] or term value [15], without any modification to the rest of our analysis.

5.2 Generation of Recurrence Equations

Given a program P and the value abstractions of its rules, a recurrence relation (RR) system for P is generated by applying the following definition to all rules in P.

DEFINITION 5.4 (total number of spawned tasks). *Let r be a rule of the form $p(\bar{x}) \leftarrow g, b_1, \ldots, b_n$ and φ_r its corresponding value relations as computed in Def. 5.1. Then, its* total tasks equation *is defined as $p(\bar{x}) = \Sigma_{i=1}^{n} \mathcal{T}(b_i), \varphi_r$, where*

$$\begin{array}{ll} \mathcal{T}(b) = 1 + q(\bar{x}) & \text{if } b = \texttt{async}\{q(\bar{x})\} \\ \mathcal{T}(b) = q(\bar{x}) & \text{if } b = \texttt{finish}\{q(\bar{x})\} \\ \mathcal{T}(b) = q(\bar{x}) & \text{if } b = q(\bar{x}) \\ \mathcal{T}(b) = 0 & \text{otherwise} \end{array}$$

The set of equations generated for a program P is denoted by $\mathcal{S}_P$.

EXAMPLE 5.5. *By applying the above definition to the rules of Fig. 1, we obtain the following set of total tasks equations:*

$$\begin{array}{ll} gaussian(n) = for(k,n) & \{k{=}0\} \\ for(k,n) = 0 & \{k \geq n\} \\ for(k,n) = & \{k < n, k' = k+1, \\ \quad for_1(n,j){+}for_2(k,n,i){+}for(k',n) & \quad j = k', i = k'\} \\ for_1(n,j) = 0 & \{j \geq n\} \\ for_1(n,j) = 1 + for_1(n,j') & \{j < n, j' = j+1\} \\ for_2(k,n,i) = 0 & \{i \geq n\} \\ for_2(k,n,i) = & \{i < n, i' = i+1, \\ \quad 1 + for_{2.1}(n,j) + for_2(k,n,i') & \quad j = k+1\} \\ for_{2.1}(n,j) = 0 & \{j \geq n\} \\ for_{2.1}(n,j) = for_{2.1}(n,j') & \{j < n, j' = j+1\} \end{array}$$

It can be observed that the only two rules in Fig. 1 that contain `async` *constructs are the second ones in for_1 and for_2. Their corresponding equations accumulate "1" for such instruction. Note that the value relations of the variables in the original are transformed into linear constraints attached to the equations. They contain the applicability conditions for the rules and how the values of variables change when moving from one procedure to another.*

5.3 Closed-form Upper Bounds

Once the RR are generated, a worst-case cost analyzer uses a solver in order to obtain closed-form UBs, i.e., *cost expressions* without recurrences. Traditionally, cost analyzers rely on computer algebra systems (e.g., MAXIMA, MAPLE) to solve the obtained recurrences. Advanced systems develop their own solvers [2, 20] in order to be able to handle more types of RR. The technical details of the process of obtaining a cost expression from the RR are not explained in the paper as our analysis does not require any modification to this part. Given a RR $p(\bar{x})$, we denote by $p^{ub}(\bar{x})$ its closed-form UB, which is a cost expression of the following form (and could be obtained by any of the above solvers):

$$e \equiv q | \mathsf{nat}(l) | \log(\mathsf{nat}(l)+1) | e * e | e + e | 2^{\mathsf{nat}(l)} | \max(e, \ldots, e)$$

where q is positive rational number, l is a linear expression, and function nat is defined as $\mathsf{nat}(v) = \max(\{v, 0\})$.

EXAMPLE 5.6. *As usual, UBs are obtained by first computing UBs for cost relations which do not depend on any other relation and continuing by replacing the computed UBs on the equations which call such relations. The solutions for the equations in Ex. 5.5 are:*

$$\begin{array}{l} for_{2.1}(n,j) = 0 \in O(1) \\ for_2(k,n,i) = n{-}i \in O(n{-}i) \\ for_1(n,j) = n{-}j \in O(n{-}j) \\ for(k,n) = 2(n{-}k)(n{-}k{-}1) \in O((n{-}k)^2) \\ gaussian(n) = \quad 2n(n{-}1) \in O(n^2) \end{array}$$

As intuitively explained in Sec. 2.1, the UB we obtain for the method gaussian *is quadratic on n. We will add 1 to this UB in order to count the task in which the initial call* gaussian(n) *is executing.*

The following theorem states the soundness of our total tasks analysis. Proofs of all technical results are available from the program chair. Intuitively, the main issue is to prove that derivations in the equations of Def. 5.4 capture all possible paths in a parallel execution of the program (and due to the overapproximation in the value abstraction possibly more). We then assume soundness of the UBs solver. In what follows, in all theorems we add one to the UB in order to count the current task on which the initial call is executing.

THEOREM 5.7. *Let P be a program with an entry procedure p, and let $p^{ub}(\bar{x})$ be a closed-form UB function for $p(\bar{x}) \in \mathcal{S}_P$. Then, for any trace $t \equiv (\mathtt{A}_0 \ ; \ \langle 1, p(\bar{x}), tv \rangle) \rightarrow^* (\mathtt{A}_n \ ; \ T_n)$, it holds that $p^{ub}(\bar{v}) + 1 \geq \mathsf{total}(t)$, where $v = tv(\bar{x})$.*

6. Inference of Peak of Alive Tasks

In the previous section, we have (over)approximated total, an accumulative resource, as defined in Sec. 4. In this section, our goal is to (over)approximate alive, a non-accumulative resource that might increase and/or decrease along execution. The main difference is that in accumulative resources one can reason by overapproximating the resource consumption in the final state of execution. This is what traditional RR (like those in Def. 5.4) do. However, in the case of non-accumulative resources, one aims at observing and (over)approximating all those states of the execution in which the consumption can be maximal and not only the final one. For our particular task-level resource, an important observation is that it is enough to approximate the behavior of the program around the program points in which the number of tasks can decrease, i.e., when reaching a `finish` construct. Such points can be detected syntactically from the program. The key idea of our analysis is to introduce a *disjunction* between the task level just before executing each `finish` and the task level reached after the `finish` resumes execution. The peak is the maximum of both disjuncts.

EXAMPLE 6.1. *Consider again the simple program of Ex. 3.2. The peak of alive tasks can be defined as the maximum of the following two scenarios:*

1. the peak before `finish`{q} (globally) terminates: *one task for* `async`$\{q_1\}$, *plus the peak of alive tasks of q (which is 2); and*
2. the peak after `finish` is executed: *one task for* `async`$\{q_1\}$, *since it might still be alive at program point* ①, *plus 1 task for* `async`$\{q_2\}$ *and 0 tasks for q_3.*

Note that, in scenario 2, we do not count the tasks created during the execution of q since `finish` *guarantees that they are not alive when we reach program point* ①. *In summary, the peak of alive tasks when executing p is 3. Additionally, we add 1 for the task in which p is running. This coincides what we have obtained in Ex. 4.1 for a particular trace.*

The next definition presents a novel form of RR, called *peak alive equations*, which overapproximates the peak of alive tasks along any execution of the program, according to the above intuition.

DEFINITION 6.2 (peak alive equations). *Let r be a rule $p(\bar{x}) \leftarrow g, b_1, \ldots, b_n$ in SSA form and φ_r its corresponding value abstraction. Then, its equation for the peak of alive tasks is $\hat{p}(\bar{x}) = \mathcal{P}(b_1, \ldots, b_n), \varphi_r$, where $\mathcal{P}$ is defined recursively as follows:*

$$\begin{array}{ll} \mathcal{P}(\epsilon) = 0 & \\ \mathcal{P}(b \cdot instr) = 1 + \hat{q}(\bar{z}) + \mathcal{P}(instr) & \text{if } b{=}\texttt{async}\{q(\bar{z})\} \\ \mathcal{P}(b \cdot instr) = \max(\hat{q}(\bar{z}), \mathcal{P}(instr)) & \text{if } b{=}\texttt{finish}\{q(\bar{z})\} \\ \mathcal{P}(b \cdot instr) = \hat{q}(\bar{z}) + \mathcal{P}(instr) & \text{if } b{=}q(\bar{z}) \\ \mathcal{P}(b \cdot instr) = \mathcal{P}(instr) & \text{otherwise} \end{array}$$

The set of equations generated for a program P is denoted by $\hat{\mathcal{S}}_P$.

Intuitively, in the above definition, we transform the peak of tasks for a given (non-empty) sequence of instructions by transforming each instruction as follows: (i) when we find an $\texttt{async}\{q(\bar{x})\}$ statement, we accumulate one new task plus the peak of tasks created along the execution of $q(\bar{x})$; (ii) in the case of $\texttt{finish}\{q(\bar{x})\}$, since it is ensured that all tasks created during the execution of $q(\bar{x})$ are terminated, we introduce a disjunction between the peak reached during the execution of $q(\bar{x})$ and the peak reached after executing the $\texttt{finish}\{q(\bar{x})\}$, and we then take the maximum of both; (iii) when we find a method call, we accumulate the peak reached during its execution with the continuation; and (iv) the remaining instructions are ignored.

EXAMPLE 6.3. *Let us first see the equations generated for the simple program of Ex. 6.1. Note that, as there are no variables, all φ_r are simply $true$ and we ignore them.*

$$\hat{p} = 1 + \hat{q}_1 + max(\hat{q}, 1 + \hat{q}_2 + \hat{q}_3)$$
$$\hat{q} = 1 + \hat{q}_4 + 1 + \hat{q}_5$$

In order to solve the above recurrence equations, the max *operator can be eliminated by transforming the equation into several non-deterministic equations, e.g., $\hat{p}(\bar{x}) = A + max(B, C), \varphi$ is translated into the two equations $\hat{p}(\bar{x}) = A + B, \varphi$ and $\hat{p}(\bar{x}) = A + C, \varphi$. Solving the above equations, under the assumption that $\hat{q}_i = 0$ for all $1 \leq i \leq 5$, results in $\hat{q} = 2$ and $\hat{p} = 3$. In this example, the accuracy gain of* alive *w.r.t.* total *is just constant but, in general, it can be much larger. For instance, the peak alive equations for the example in Sec. 2.1 are:*

$$\hat{gaussian}(n) = \hat{for}(k, n) \quad \{k = 0\}$$
$$\hat{for}(k, n) = 0 \quad \{k \geq n\}$$
$$\hat{for}(k, n) = \max\{\hat{for}_1(n, j), \max\{\hat{for}_2(k, n, i), \hat{for}(k', n)\}\} \quad \{k < n, k' = k + 1, j = k', i = k'\}$$
$$\hat{for}_1(n, j) = 0 \quad \{j \geq n\}$$
$$\hat{for}_1(n, j) = 1 + \hat{for}_1(n, j') \quad \{j < n, j' = j + 1\}$$
$$\hat{for}_2(k, n, i) = 0 \quad \{i \geq n\}$$
$$\hat{for}_2(k, n, i) = 1 + \hat{for}_2(k, n, i') \quad \{i < n, i' = i + 1, j = k + 1\}$$

The solution of $\hat{for}_1$ and $\hat{for}_2$ is like in Ex. 5.6. After replacing them in the second equation of $\hat{for}$ and eliminating the max operator, we obtain as peak alive UB is $\hat{gaussian}(n) = n - 1 \in O(n)$. Note that the total UB was quadratic on n. Again, we should add 1 to count the task in which the initial call is being executed.

The following theorem states that the solutions of the equations generated in Def. 6.2 is a sound approximation of peakAlive.

THEOREM 6.4. *Let P be a program with an entry procedure p, and let $\hat{p}^{ub}(\bar{x})$ be a closed-form UB function $\hat{p}(\bar{x}) \in \hat{\mathcal{S}}_P$. Then, for any trace $t \equiv (\mathtt{A}_0 \ ; \ \langle 1, p(\bar{x}), tv\rangle) \rightarrow^* (\mathtt{A}_n \ ; \ T_n)$ it holds that $\hat{p}^{ub}(\bar{v}) + 1 \geq \mathsf{peakAlive}(t)$ where $\bar{v} = tv(\bar{v})$.*

7. Inference of Peak of Available Tasks

The goal of this section is to accurately approximate peakAvailable, or the task level. Note that, when inferring peakAlive in the previous section, we have possibly included tasks which are alive but suspended. For the applications discussed in Sec. 2, it is clearly useful to exclude suspended tasks from the peak, e.g., it is not worth allocating suspended tasks in a separate processor.

EXAMPLE 7.1. *Consider again the program of Ex. 6.1, and recall that in 6.3 we have inferred that the peak of alive tasks is $\hat{p} = 3$ plus 1 for the task in which p is running. However, during the execution of p the maximum number of tasks which are available (not suspended) is only 3. This is because the task in which p is executing is available until it reaches the call $\texttt{async}\{q_5\}$ since, as soon as q_5 is invoked asynchronously, p suspends and has to wait for q_4 and q_5 to terminate before proceeding to program point ①.*

In general, it is not easy to detect when tasks are blocked, since often the execution of $\texttt{finish}\{p(\bar{x})\}$ spawns asynchronous calls but it also executes other instructions. Therefore, the task in which $\texttt{finish}\{p(\bar{x})\}$ is executed does not always block. However, in all cases where the last instruction of $p(\bar{x})$ (directly or indirectly) is an asynchronous call, we have a behavior similar to the above example, i.e, at the same time the task in which $\texttt{finish}\{p(\bar{x})\}$ is executing suspends and another task starts. Many of these cases can be syntactically detected and treated in a special way. In what follows, we explain how to handle a common pattern in which $p(\bar{x})$ consists of only asynchronous calls, as in the above example. In order to keep the task-level analysis as simple as possible, we introduce an auxiliary construct in the language, called `finish--async`, by means of the following program transformation.

DEFINITION 7.2 (`finish-async`). *Given an instruction of the form $\texttt{finish}\{p(\bar{x})\}$, if p is defined by a single rule of the form $p(\bar{x}) \leftarrow \texttt{async}\{q_1(\bar{x}_1)\}, \ldots, \texttt{async}\{q_n(\bar{x}_n)\}$, then we replace the original instruction by $\texttt{finish}{-}\texttt{async}\{q_1(\bar{x}_1), \ldots, q_n(\bar{x}_n)\}$.*

The use of well-known transformations such as *unfolding* can be useful to detect the above pattern in the presence of intermediate rules and be able to apply the transformation more often. For instance, if we have, $p \leftarrow q, \ldots, \texttt{async}\{q_n\}$ where q is defined as $q \leftarrow \texttt{async}\{q_1\}$, we need to unfold the body of q in order to be able to introduce the `finish-async` construct. Luckily, this is a well-studied problem in the field of partial evaluation [12] and existing unfolding strategies can be directly applied in our context.

DEFINITION 7.3 (peak available equations). *The peak available equations extend those of Def. 6.2 with the additional case*

$$\mathcal{P}(b \cdot instr) = \max(n{-}1 + \hat{q}_1(\bar{z}_1){+}\cdots{+}\hat{q}_n(\bar{z}_n), \mathcal{P}(instr))$$

which is applied when $b = \texttt{finish}{-}\texttt{async}\{q_1(\bar{z_1}), \ldots, q_n(\bar{z_n})\}$.

EXAMPLE 7.4. *Applying the `finish-async` transformation on the program of Ex. 3.2 results in the following rule for p*

$$p \leftarrow \texttt{async}\{q_1\}, \texttt{finish}{-}\texttt{async}\{q_4, q_5\}, \texttt{async}\{q_2\}, q_3$$

Applying Def. 7.3, we obtain the following peak available equation: $\hat{p} = 1 + \hat{q}_1 + max(1 + \hat{q}_4 + \hat{q}_5, 1 + \hat{q}_2 + \hat{q}_3)$. Solving the above equation, under the assumption that $\hat{q}_i = 0$ for all $1 \leq i \leq 5$, results in $\hat{p} = 2$. Therefore, at most $\hat{p} + 1 = 3$ tasks might be available at the same time during the execution of p. The accuracy achieved by the peak available equations w.r.t. the alive ones can be large. For instance, consider the (intermediate representation for the) program in Sec. 2.2:

$$msort(from, to) \leftarrow from \geq to.$$
$$msort(from, to) \leftarrow from < to, mid{:=}(from + to)/2,$$
$$\texttt{finish}{-}\texttt{async}\{msort(from, mid), msort(mid + 1, to)\}$$
$$merge(from, to, mid).$$

We show at the top (resp. bottom) the equations obtained by applying Def. 6.2 (resp. Defs. 7.2 and 7.3) to the above rules:

$$\hat{msort}(f, t) = 0 \ \{f \geq t\}$$
$$\hat{msort}(f, t) = \max(\hat{aux}(f, t, m'), \hat{merge}(f, t, m')) \quad \{f < t, \ 2m' = f + t\}$$
$$\hat{aux}(f, t, m) = 2 + \hat{msort}(f, m) + \hat{msort}(m', t) \quad \{m' = m + 1\}$$

$$\hat{msort}(f, t) = 0 \ \{f \geq t\}$$
$$\hat{msort}(f, t) = \max(1 + \hat{msort}(f, m') + \hat{msort}(m'', t), \hat{merge}(f, t, m')) \quad \{f < t, \ 2m' = f + t, m'' = m' + 1\}$$

As pointed out in Sec. 2.2, the solution for the equations at the top is $2 * (t - f + 1) - 2$, while for the ones at the bottom is $(t - f + 1)$.

Clearly, the available tasks are a more useful piece of information when deciding how to distribute execution.

The following theorem states the soundness of Def. 7.3 when rules are transformed using Def. 7.2.

THEOREM 7.5. *Let P be a program with an entry procedure p, and let $\hat{p}^{ub}(\bar{x})$ be a closed-form UB function for $\hat{p}(\bar{x}) \in \hat{\mathcal{S}}_P$ where $\hat{\mathcal{S}}_P$ is the cost relation generated after applying the* finish--async *transformation of Def. 7.2. Then, for any trace $t \equiv (\mathtt{A}_0 \ ; \ \langle 1, p(\bar{x}), tv\rangle) \rightarrow^* (\mathtt{A}_n \ ; \ T_n)$ it holds that $\hat{p}^{ub}(\bar{v}) + 1 \geq \mathsf{peakAvailable}(t)$, where $\bar{v} = tv(\bar{x})$.*

8. Combining Escaped and Peak

In this section, our goal is to improve the accuracy of the UBs we have obtained in the previous sections by exploiting knowledge on which tasks *escape* from the scope of a method call. The number of escaped tasks from a (normal) method call $q(\bar{x})$, refers to the number of tasks created during the execution of the method call $q(\bar{x})$ which are alive after its local termination. Such escaped tasks could start their execution even after the local termination of $q(\bar{x})$. For an asynchronous call $\mathtt{async}\{p(\bar{x})\}$, in principle, the number of tasks that can escape from it is bounded by its peak, and for $\mathtt{finish}\{q(\bar{x})\}$ is 0 by definition. In this section, we use this information in order to improve the peak of alive and available tasks. We use the term peak of tasks to refer to any of the former, alive or available. Let us see the idea on a simple example.

EXAMPLE 8.1. *Consider the following program:*

$m \leftarrow p, ①\mathtt{async}\{q\}$
$p \leftarrow \mathtt{async}\{q\}, \mathtt{finish}\{h\}, \mathtt{async}\{q\}$
$h \leftarrow \mathtt{async}\{q\}, \mathtt{async}\{q\}, \mathtt{async}\{q\}$

and assume that procedure q does not make any asynchronous call. By applying Def. 7.3, we generate the following equations for the peak of available tasks:

$\hat{m} = \hat{p} + 1 + \hat{q}$
$\hat{p} = \ 1 + \hat{q} + \max(2 + \hat{q} + \hat{q} + \hat{q}, 1 + \hat{q})$

which, since $\hat{q} = 0$, are solved to $\hat{m} = 4$. Let us explain how we can refine this peak using escape *information. While the peak of available tasks when executing p is 3, only 2 tasks can* escape *from p, i.e., they can be available after program point ①. The idea is that the peak of available tasks for m (ignoring the task in which m is being executed) can be defined as the maximum of the following two scenarios: (a) the peak of the tasks while executing p or (b) those that escape from p plus 1 for the last asynchronous call in m. This will lead to 3, which improves the previous peak by one.*

Let us first specify the notion of escaped tasks from a given call more precisely in the concrete setting.

DEFINITION 8.2 (escaped tasks). *Consider a program P with an entry procedure p and a trace $t = (\mathtt{A} \ ; \ \langle 1, p(\bar{x}), tv\rangle) \rightarrow^* (\mathtt{A}_n \ ; \ T_n)$ such that p locally terminates before reaching T_n. The number of escaped tasks from p in t can be defined as* $\mathsf{escape}(p) = |\mathsf{available}(T_n)|$.

The following definition presents a novel form of equations, called *combined peak/escape* equations, which allows us to take advantage of static knowledge on the escaped tasks in order to approximate the peak of tasks more accurately. Given a procedure $p(\bar{x})$, the main idea is to set up two kinds of relations: (1) *the peak equations* $\hat{p}(\bar{x})$: which define the peak of tasks reached during the execution of p and (2) *the escaped equations* $\check{p}(\bar{x})$: which define the escaped tasks from a call to $p(\bar{x})$. The definition for both relations is mutually recursive, as the next definition shows.

DEFINITION 8.3 (combined peak/escape equations). *Let r be a rule and φ_r its corresponding value abstraction as in Def. 6.2. The* combined peak and escaped *equations for r consist of its* escape equation $\check{p}(\bar{x}) = \sum_{i=1}^{n} \mathcal{E}(b_i), \varphi_r$, *where:*

$\mathcal{E}(b) = 1 + \hat{q}(\bar{z})$ *if* $b = \mathtt{async}\{q(\bar{z})\}$
$\mathcal{E}(b) = \check{q}(\bar{z})$ *if* $b = q(\bar{z})$
$\mathcal{E}(b) = 0$ *otherwise*

and its peak equation *which is obtained like the peak equations of Def. 7.3, but changing the definition of $\mathcal{P}$ when $b = q(\bar{z})$ by:* $\mathcal{P}(b \cdot instr) = \max(\hat{q}(\bar{z}), \check{q}(\bar{z}) + \mathcal{P}(instr))$

In the above definition, it can be observed that the peak equation modifies that in Def. 6.2 in the case of a synchronous call in order to take advantage of the escape information, as intuitively explained in Ex. 8.1. In the escape equation, we distinguish three cases: (i) when we find an asynchronous call, then such new task can escape plus the *peak* of tasks created along the execution of such call; (ii) for synchronous calls, we count those that escape from such call; (iii) the remaining instructions map to zero, e.g., when we have a $\mathtt{finish}\{\mathtt{s}\}$, we are sure that nothing escapes from it.

EXAMPLE 8.4. *The solution of the following combined equations, obtained by applying Def. 8.3 to the rules of the program of Ex. 8.1, corresponds to the improved peak UB, as explained in Ex. 8.1:*

$\hat{m} = max(\hat{p}, \check{p} + 1 + \hat{q})$ $\quad\quad$ $\check{m} = \check{p} + 1 + \hat{q}$
$\hat{p} = \ 1 + \hat{q} + max(2 + \hat{q} + \hat{q} + \hat{q}, 1 + \hat{q})$ $\quad$ $\check{p} = \ 1 + \hat{q} + 1 + \hat{q}$

In the above example, the accuracy gain is constant. In general, it can be much larger (even in complexity order). Let us consider the program in Sec. 2.3 whose intermediate representation is:

$f(n, i) \leftarrow n \leq 0$
$f(n, i) \leftarrow n > 0, \mathtt{finish}{-}\mathtt{async}\{activity\_a(i), activity\_b(i)\},$
$\quad n' := n - 1, i' := 2 * i + 1, i'' := 2i + 2,$
$\quad f(n', i'), f(n', i'')$

By applying Def. 8.3, we obtain the equations:

$\check{f}(n, i) = 0 \quad \{n \leq 0\}$
$\check{f}(n, i) = \check{f}(n', i') + \check{f}(n', i'') \quad \varphi$
$\hat{f}(n, i) = 0 \quad \{n \leq 0\}$
$\hat{f}(n, i) = \max(1 + \hat{activity\_a}(i) + \hat{activity\_b}(i),$
$\quad \max(\hat{f}(n', i'), \check{f}(n', i') + \max(\hat{f}(n', i''), \check{f}(n', i'')))) \quad \varphi$

where $\varphi = \{n > 0, n' = n - 1, i' = 2i + 1, i'' = 2i + 2\}$ and $\hat{activity\_a}(i) = \hat{activity\_b}(i) = 0$. Since $\check{f}(n, i)$ is solved to 0, the solution to the combined equations is the constant 1. Note that, applying Def. 5.4, we obtain the exponential bound shown in Sec. 2.3. Applying either Def. 6.2 or Def. 7.3, we obtain an exponential bound as well. Hence, the solution of the combined equations is much more accurate than all previous solutions.

Soundness of our analysis guarantees that $\hat{p}$ and $\check{p}$ correctly approximate the peak of available tasks and the escaped tasks, respectively. The proof relies on an auxiliary notion of escaped tasks from a given state and derivation that appears in the technical report.

THEOREM 8.5. *Let P be a program with an entry procedure p. Let q be a procedure defined in P. Let $\hat{p}^{ub}(\bar{x})$ be a closed-form UB function for its combined peak/escape equations. Given a trace $t \equiv (\mathtt{A}_0 \ ; \ \langle 1, p(\bar{x}), tv\rangle) \rightarrow^* (\mathtt{A}_n \ ; \ T_n)$. Then, it holds that*

1. $\hat{p}^{ub}(\bar{v}) + 1 \geq \mathsf{peakAvailable}(t)$*; and*
2. $\check{p}^{ub}(\bar{v}) + 1 \geq \mathsf{escape}(t)$.

where $\bar{v} = tv(\bar{x})$

Note that if we use the peak equations as in Def. 6.2 instead of point 1 above it holds that $\hat{p}^{ub}(\bar{v}) + 1 \geq \mathsf{peakAlive}(t)$.

#	$\mathbf{U}_\mathsf{T}$/ $\mathbf{U}_\mathsf{A}$/ $\mathbf{U}_\mathsf{E}$	ms	#	$\mathbf{U}_\mathsf{T}$/ $\mathbf{U}_\mathsf{A}$/ $\mathbf{U}_\mathsf{E}$	ms
1	$(N-1)(\log N)$	500	2	$61441N+61441$	310
	$(N-1)(\log N)$	310		$61441N+61441$	270
	$N-1$	450		61569	460
3	$2048N+48$	240	4	$2^{N-1}-1$	200
	$2048N+48$	260		$2^{N-1}-1$	210
	$1024N+16$	390		$2^{N-1}-1$	240
5	kN^3+3kN^2+kN	830	6	$50*(2N+2000)$	170
	$(k+1)N^3+(k+2)$ $N^2+(k+1)N$	760		$\max(N,2000)$	170
	$(k+1)N^3+(2k+3)$ $N^2+(k+3)N+1$	1340		$\max(N,2000)$	210
7	$10N_1N_2$	2680	8	N	100
	$10N_1N_2$	1780		1	90
	$N_1N_2+N_1$	2850		1	140

Table 1. Benchmarks: 1 ArraySum (1044 Kb); 2 CUDABlackScholes (1071); 3 FRASimpleDist (1134); 4 Fib (717); 5 HeatTransfer_v1 (1913); 6 KMeansDist (1124); 7 PLU_2_C (8520); 8 method print()V of SparseMat (706).

As final remarks, we note that the further accuracy of the combined equations might come at the price of efficiency and effectiveness of the analysis. As regards efficiency, the fact that for each procedure in the program, we generate two sets of equations, increases the analysis time. In particular, the time required to infer closed-form UBs for the combined relations almost doubles. As regards effectiveness, the fact that the definition of both relations are mutually recursive, can make their solving process more complex. Nonetheless, the mutual recursion disappears in many cases, e.g., when the number of escaped tasks is constant. Also, certain solvers (e.g., MAXIMA) have support to solve such mutual recursions. After solving the equations, it is guaranteed that the obtained UBs are strictly more precise than those obtained in the previous sections.

9. Experimental Results

We have implemented our technique within the XYZ[1] system which can be tried out online at: XYZ[2]. The experimental evaluation has been performed on a set of small but representative X10 programs (available at the X10 website `http://x10-lang.org/`) containing interesting parallelism patterns. In the implementation, we are using existing tools developed for Java to translate the original program into the IR. Hence, the examples have been first (manually) translated from X10 to Java, preserving the structure of the parallelism. From that point on, the analysis is fully automatic. In some cases, purely numerical computations have been omitted (e.g., most of the method doBlackScholes in CUDABlackScholes), and pieces of code which manipulate data structures in a way which is specific to X10 have been simplified. *Places* have been ignored. Also, to avoid virtual invocations that often complicates the analysis, we sometimes translate calls $o.m()$ to $m(o)$ and define m as a static method. Finally, async and finish statements have been simulated (only for the sake of the analysis, not for actual execution in the JVM) by means of special method calls. Overall, the translation is done in such a way that the Java code arguably preserves the properties of interest.

The results are shown in Table 1. For each benchmark, the total number $\mathbf{U}_\mathsf{T}$ of spawned tasks (first row), the peak $\mathbf{U}_\mathsf{A}$ of alive tasks (second row), and the refined peak $\mathbf{U}_\mathsf{E}$ of alive tasks using escape information (third row) are inferred. We do not add 1 for the initial task. Most examples take as input a numerical parameter, which is a measure of the size of the problem. Such parameter is usually taken to be the length of the array of String which is the argument of the main method, and appears as N in the table (N_1 and N_2 if the input consists of two parameters). In two cases, $\mathbf{U}_\mathsf{A}$ is better than $\mathbf{U}_\mathsf{T}$, meaning that the analysis was able to infer that some tasks cannot be alive at the same time. Moreover, $\mathbf{U}_\mathsf{E}$ improves on $\mathbf{U}_\mathsf{A}$ in four examples, thus showing the usefulness of considering escape information. The table also shows (next to the name of the benchmark) the size in Kbytes of the (transformed) .class file, and the total analysis time ms in milliseconds.

Let us explain the results in more detail. ArraySum is interesting because the sum is executed many times under different assumptions about the number of tasks which are going to be spawned: at each iteration, this number is multiplied by 2 (starting from 1) until a threshold N is reached (note that the X10 code uses a constant threshold 4, so that our version is more general). The result is that at most $N-1$ tasks are spawned at each one of the $\log N$ iterations, thus giving a total of $(\log N)*(N-1)$ tasks. On the other hand, due to the finish statement which wraps each iteration, only $N-1$ tasks can be alive at the same time, thus giving such number as $\mathbf{U}_\mathsf{E}$. Note that the analysis of alive tasks needs escape information in order to get the linear upper bound.

In CUDABlackScholes, N is the number of iterations which is the constant 512 in the original program. It can be seen that $\mathbf{U}_\mathsf{T}$ is bigger since every iteration is performed inside a finish statement, so that tasks created during different iterations cannot be alive at the same time. The UB of Fib is exponential due to the structure of the recursive calls. The total number and the peak number of tasks are equal and indeed all spawned tasks can be alive at the same time.

In HeatTransfer_v1, the UB is cubic in all cases, since the operations on the data structures spawn a cubic number of tasks, and all tasks are alive at the same time since a single finish statement wraps this part of the code. The difference (not in the order of magnitude) between the UBs is due to the different loss of precision when solving the equations. The number of iterations of the loop in run() depends on the guard delta<epsilon on double numbers. This bound is unpredictable by most state-of-the-art static analyzers, so that the program has been modified in order to iterate a fixed number of times k. In KMeansDist, the constants 2000 and 50 appearing in the UBs are constants in the X10 code, while N is a measure of the size of the data structure. In the biggest example PLU_2_C, considering escape information allows to remove a constant factor 10 which is a constant in the program code.

Overall, we argue that, although our implementation is still a prototype, the experiments show that our approach is promising and leads to reasonably accurate task-level UBs in a fully automatic way.

10. Related Work

As regards the language, several subsets of X10 ([1, 13, 17]) have been defined in the literature. For the parallel part of the language, the subset we consider is like [13]. The sequential part is richer than [13], as not handling recursion would be an important restriction for the task-level analysis. The majority of related work around the X10 language is on may-happen-in-parallel analysis [13] and determinism [23]. This is a complementary line of research to ours, in the sense that we can use the results of such analyses to improve ours, as we will discuss in Sec. 11.

Due to our interpretation of the task level of a program as a resource consumed along its execution, our work is more directly related to cost analysis (or resource usage analysis) frameworks [2, 9, 10, 21]. All such frameworks assume a sequential ex-

[1] the system name is withheld

[2] the actual link is withheld

ecution model. Moreover, they often are applied to measure accumulative resources. Another non-accumulative resource is memory consumption in the presence of garbage collection. There has been a respectable development in heap space analysis for Java-like and functional languages [2, 4, 6, 11, 21] during the last years. Among them, our work is more related to those that rely on RR [2, 21]. Still, heap space bounds are fundamentally different from task-level bounds, as in the case of memory, the challenge is to model the behaviour of the garbage collector at the level of the cost equations. In our case, the challenge is to handle concurrency and be able to capture in the equations the states in which tasks terminate.

11. Conclusions and Future Work

We have presented a novel static analysis to approximate the task level of parallel X10-like programs. Our approach is based by the view that the task level of a program is a particular (non-accumulative) resource consumed along its (parallel) execution. Existing cost analysis frameworks assume a standard *sequential* programming model on resources which are typically *accumulative*. It is clear that both these deviations from existing frameworks add significant complexity to the problem of inferring task-level bounds. Our key contribution is the generation of task-level *recurrence relations* that soundly and accurately approximate the task level of the program in the parallel setting. An important observation (and a side-effect contribution of our work, and due to the characteristics of X10) is that obtaining an UB from the RR implies bounding the number of iterations of loops in the original X10 program. Therefore, our work indirectly provides a *global termination analysis* for X10 programs. In other words, if the analysis finds a task-level UB, it is guaranteed that the original X10 program terminates for any input data.

The abstraction performed by the value analysis component, though simple, ensures that the UBs obtained are sound for any particular task scheduler. One direction for future work is to improve the precision of the analysis by enriching the value analysis assuming a particular scheduling. To do this, we first need to make some assumption on the policy which establishes which tasks run in parallel. Then, we can reuse existing may-happen-in-parallel analyses as those in [13], which specifically treat the `async-finish` constructs of X10. The output of such analysis annotates each instruction with the set of instructions that can be executed in parallel with it. One could then prove that the fragments of code which might be executed in parallel are independent [23] (i.e., they do not read/write on the same global data). In such case, we can then use existing field-sensitive value analyses [14] developed for similar languages in order to improve the precision of our UBs.

As another direction for future work, we plan to extend our analysis to the full X10 language. In particular, we believe handling places can give us some interesting results. This requires enhancing the `async` construct as `async`$\{s, id\}$, with the identifier id of the server encharged of running the task s asynchronously. An interesting application of our analysis in this setting is to infer the throughput of the different servers of the system, which could be very useful to balance the workload in distributed applications.

Our approach can be easily adapted to count the peak at a program point, i.e., the maximum number of tasks that can be alive (or available) in parallel at that specific program point. Suppose that the program point of interest is ⓐ, then we can modify Def. 6.2 as follows: we add $\mathcal{P}(ⓐ \cdot instr) = 1 + \mathcal{P}(instr)$ and remove the constant 1 from the equation of `async`. Such information is useful, for example, when at the program point of interest, we query a server. The obtained UB indicates the load of the server.

References

[1] M. Abadi and G. D. Plotkin. A model of cooperative threads. In *Proc. of POPL'09*, pages 29–40. ACM, 2009.

[2] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Parametric Inference of Memory Requirements for Garbage Collected Languages. In *9th International Symposium on Memory Management (ISMM'10)*, pages 121–130, New York, NY, USA, June 2010. ACM Press.

[3] Andrew W. Appel. Ssa is Functional Programming. *SIGPLAN Notices*, 33(4):17–20, 1998.

[4] V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric Prediction of Heap Memory Requirements. In *ISMM*. ACM Press, 2008.

[5] P. Charles, C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster computing. In *OOPSLA*, pages 519–538. ACM, 2005.

[6] W-N. Chin, H.H. Nguyen, C. Popeea, and S. Qin. Analysing Memory Resource Bounds for Low-Level Programs. In *ISMM*. ACM Press, 2008.

[7] R. DeLine and K.R.M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[8] M. Fähndrich. Static Verification for Code Contracts. In *SAS*, volume 6337 of *LNCS*, pages 2–5. Springer, 2010.

[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] J. Hoffmann and M. Hofmann. Amortized Resource Analysis with Polynomial Potential. In *ESOP*, volume 6012 of *LNCS*, pages 287–306. Springer, 2010.

[11] M. Hofmann and S. Jost. Type-Based Amortised Heap-Space Analysis. In *15th European Symposium on Programming, ESOP 2006*, volume 3924 of *Lecture Notes in Computer Science*, pages 22–37. Springer, 2006.

[12] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, New York, 1993.

[13] Jonathan K. Lee and Jens Palsberg. Featherweight X10: A Core Calculus for Async-Finish Parallelism. In *Proc. of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'10)*, pages 25–36, New York, NY, USA, 2010. ACM.

[14] A. Miné. Field-Sensitive Value Analysis of Embedded C Programs with Union Types and Pointer Arithmetics. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*, 2006.

[15] C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Automated Termination Analysis of Java Bytecode by Term Rewriting. In *Proc. of RTA'10*, volume 6 of *LIPIcs*, pages 259–276, 2010.

[16] R. Raman, J. Zhao, V. Sarkar, M. T. Vechev, and E. Yahav. Efficient data race detection for async-finish parallelism. In *Proc. of RV*, pages 368–383, 2010.

[17] V. A. Saraswat and R. Jagadeesan. Concurrent Clustered Programming. In *Proc. of CONCUR'05*, volume 3653 of *Lecture Notes in Computer Science*. Springer, 2005.

[18] F. Spoto, F. Mesnard, and É. Payet. A Termination Analyser for Java Bytecode based on Path-Length. *ACM TOPLAS*, 32(3), 2010.

[19] W. Zou T. Wei, J. Mao and Y. Chen. A new algorithm for identifying loops in decompilation. In *SAS'07*, LNCS 4634, pages 170–183, 2007.

[20] L. Unnikrishnan and S. Stoller. Parametric heap usage analysis for functional programs. In *Proc. of ISMM'09*. ACM Press, 2009.

[21] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Optimized Live Heap Bound Analysis. In *Proc. of VMCAI'03*, volume 2575 of *LNCS*, pages 70–85, 2003.

[22] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java Optimization Framework. In *1999 conference*

of the Centre for Advanced Studies on Collaborative Research (CASCON'99), 1999.

[23] M. T. Vechev, E. Yahav, R. Raman, and V. Sarkar. Automatic Verification of Determinism for Structured Parallel Programs. In *Proc. of SAS'10*, volume 6337 of *Lecture Notes in Computer Science*, pages 455–471. Springer, 2010.

[24] B. Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9), 1975.

Appendix B

Modular Termination Analysis of Java Bytecode and its Application to phomeME Core Libraries

The paper "Modular Termination Analysis of Java Bytecode and its Application to phomeME Core Libraries" [58] follows.

Modular Termination Analysis of Java Bytecode and its Application to phoneME Core Libraries

D. Ramírez-Deantes[1], J. Correas[2], and G. Puebla[1]

[1] DLSIIS, Technical University of Madrid (UPM), Spain
[2] DSIC, Complutense University of Madrid (UCM), Spain

Abstract. Termination analysis has received considerable attention, traditionally in the context of declarative programming and, recently, also for imperative and Object Oriented (OO) languages. In fact, there exist termination analyzers for OO which are capable of proving termination of medium size applications by means of *global* analysis, in the sense that all the code used by such applications has to be proved terminating. However, global analysis has important weaknesses, such as its high memory requirements and its lack of efficiency, since often some parts of the code have to be analyzed over and over again, libraries being a paramount example of this. In this work we present how to extend the termination analysis in the COSTA system in order to make it modular by allowing separate analysis of individual methods. The proposed approach has been implemented. We report on its application to the termination analysis of the core libraries of the phoneME project, a well-known open source implementation of Java Micro Edition (JavaME), a realistic but reduced version of Java to be run on mobile phones and PDAs. We argue that such experiments are relevant, since handling libraries is known to be one of the most relevant open problems in analysis and verification of real-life applications. Our experimental results show that our proposal dramatically reduces the amount of code which needs to be handled in each analysis and that this allows proving termination of a good number of methods for which global analysis is unfeasible.

1 Introduction

It has been known since the pre-computer era that it is not possible to write a program which correctly decides, in all cases, if another program will *terminate*. However, termination analysis tools strive to find proofs of termination for as wide a class of (terminating) programs as possible. Automated techniques are typically based on analyses which track *size* information, such as the value of numeric data or array indexes, or the size of data structures. This information is used for specifying a *ranking function* which strictly decreases on a well-founded domain on each computation step, thus guaranteeing termination.

In the last two decades, a variety of sophisticated termination analysis tools have been developed, primarily for less-widely used programming languages. These include analyzers for term rewrite systems [16], and logic and functional

languages [18, 9, 17]. Termination-proving techniques are also emerging in the imperative paradigm [7, 10, 16] and the object oriented (OO for short) paradigm, where static analysis tools such as Julia [25], AProVE [21], and COSTA [1] are able to prove termination of non-trivial medium-size programs.

In the context of OO languages, we focus on the problem of proving whether the execution of a method m terminates for any possible input value which satisfies m's precondition, if any. Solving this problem requires, at least in principle, a *global analysis*, since proving that the execution of m terminates requires proving termination of all methods transitively invoked during m's execution. In fact, the three analysis tools for OO code mentioned above require the code of all methods reachable from m to be available to the analyzer and aim at proving termination of all the code involved. Though this approach is valid for medium-size programs, we quickly get into scalability problems when trying to analyze larger programs. It is thus required to reach some degree of *compositionality* which allows decomposing the analysis of large programs into the analysis of smaller parts.

In this work we propose an approach to the termination analysis of large OO programs which is compositional and we (mostly) apply it by analyzing a method at a time. We refer to the latter as *modular*, i.e., which allows reasoning on a method at a time. Our approach provides several advantages: first, it allows the analysis of larger programs, since the analyzer does not need to have the complete code of the program nor the intermediate results of the analysis in memory. Second, methods are often used by several other methods. The analysis results of a shared method can be reused for multiple uses of the method.

The approach presented is flexible in the level of granularity: it can be used in a component-based system at the level of components. A specification can be generated for a component C by analyzing its code, and it can be deployed together with the component and used afterwards for analyzing other components that depend on this one. When analyzing a component-based application that uses C, the code of C does not need to be available at analysis time, since the specification generated can be used instead.

In order to evaluate the effectiveness of our approach, we have extended the COSTA analyzer to be able to perform modular termination analysis and we have applied the improved system to the analysis of the phoneME implementation of the core libraries of JavaME. Note that analysis of API libraries is quite challenging and a significant stress test for the analyzer for a number of reasons which are discussed in more detail in Section 5 below.

The main contribution of this paper is that it provides a practical framework for the modular analysis of Java bytecode, illustrating its applicability to real programs by analyzing phoneME librares. These contributions are detailed from Section 4 onwards.

2 Non-Modular Termination Analysis in COSTA

COSTA (see [4] and its references) is a cost [2] and termination [1] analyzer for Java bytecode. COSTA receives as input the signature of the method m whose

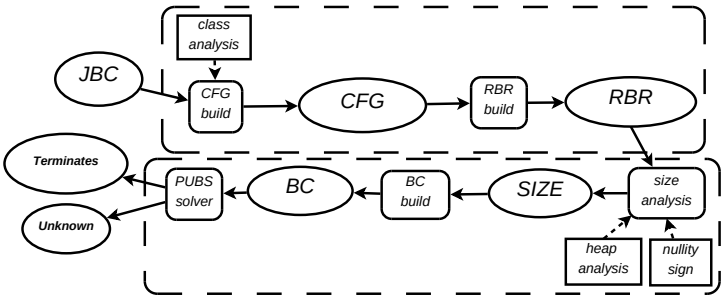


Fig. 1. Architecture of COSTA

termination (or cost) we want to infer. Method m is assumed to be available in the classpath or default Java run-time environment (jre for short) libraries, together with all other methods and classes transitively invoked by m. Since there can be many more classes and methods in the classpath and jre than those reachable from m, a first step during analysis consists in identifying a set M of methods which includes all methods reachable from m. This phase is sometimes referred to as *program extraction* or *application extraction*. Then, COSTA performs a *global analysis* since, not only m, but all methods in the program M are analyzed.

We now briefly describe the overall architecture of COSTA, which is graphically represented in Figure 1. More details can be found in [3]. The dashed frames represent the two main phases of the analysis: (i) consists of extracting a program M from the method m plus the transformation of the bytecode for all methods in M into a suitable internal representation; and (ii) the actual static analysis. Input and output of the system are depicted on the left: by *JBC* we denote the bytecode of all classes in the classpath and jre plus the signature of a method and yields information about termination, indicated by *Terminates* (the analyzer has proved that the program terminates for all valid inputs) or *Unknown* (otherwise). Ellipses (e.g. *CFG*) represent *what* the system produces at each intermediate stage of the analysis; rounded boxes (e.g. "*CFG build*") indicate the *main steps* of the analysis process; square boxes (e.g. *class analysis*), which are connected to the main steps by dashed arrows, denote auxiliary analyses which allow obtaining more precise results.

During the first phase, depicted in the upper half of the figure, the incoming *JBC* is transformed into a *rule-based representation* (*RBR*). In the second phase, depicted in the lower half of the figure, the system performs the actual termination analysis on the RBR.

2.1 From the Bytecode to the Rule-based Representation

Generation of Control Flow Graphs guided by Class Analysis COSTA transforms the bytecode of a method into *Control Flow Graphs* (CFGs) by using techniques from compiler theory. As regards *Virtual invocation*, computing a precise approximation of the methods which can be executed at a given program point is not trivial. As customary in the analysis of OO languages, COSTA uses *class analysis* [24] (or points-to analysis) in order to precisely approximate this

information. First, the CFG of the initial method is built, and class analysis is applied in order to approximate the possible runtime classes at each program point. This information is used to *resolve* virtual invocations. Methods which can be called at runtime are loaded, and their corresponding CFGs are constructed. Class analysis is applied to their body to include possibly more classes, and the process continues iteratively. Once a fixpoint is reached, it is guaranteed that all reachable methods have been loaded, and the corresponding CFGs have been generated.

As regards *exceptions*, COSTA handles internal exceptions (i.e., those associated to bytecodes as stated in the JVM specification), exceptions which are thrown (bytecode athrow) and possibly propagated back in methods, as well as finally clauses. Exceptions are handled by adding edges to the corresponding handlers. COSTA provides the options of ignoring only internal exceptions, all possible exceptions or considering them all.

Rule-based Representation Given a method m and its CFGs, a RBR for m is obtained by producing, for each basic block m_j in its CFGs, a rule which:

(1) contains the set of bytecode instructions within the basic block;
(2) if there is a method invocation within the instructions, includes a call to the corresponding rule; and
(3) at the end, contains a call to a *continuation rule* m_j^c which includes mutually exclusive rules to cover all possible continuations from the block.

Note that several rules may be produced with the same name. A *procedure* P is the set of all rules with name P.

2.2 Context-Sensitive (Pre-)Analyses to Improve Accuracy

COSTA performs three context-sensitive analyses on the RBR based on abstract interpretation [12]: *nullity*, *sign* and *heap* analysis. These analyses improve the accuracy (and efficiency) of subsequent steps inferring information from individual bytecodes, and propagating it via a standard, top-down *fixpoint* computation.

Nullity Analysis aims at keeping track of reference variables which are definitely *null* or are definitely *non-null*. For instance, the bytecode $\mathsf{new}(s_i)$ allows assigning the abstract value *non-null* to s_i. The results of nullity analysis often allow removing rules corresponding to NullPointerException.

Sign Analysis aims at keeping track of the sign of variables. The abstract domain contains the elements $\geq$, $\leq$, $>$, $<$, $= 0$, $\neq 0$, $\top$ and $\bot$, partially ordered in a lattice. For instance, sign analysis of $\mathsf{const}(s_i, V)$ evaluates the integer value V and assigns the corresponding abstract value $= 0$, $>$ or $<$ to s_i, depending, resp., on if V is zero, positive or negative [12]. Knowing the sign of data allows removing RBR rules for arithmetic exceptions which are never thrown.

Heap Analysis obtains information related to variables and arguments located in the heap, a global data structure which contains objects (and arrays) allocated by the program. Infers properties like *constancy* and *cyclicity* of variables and arguments, and *sharing*, *reachability* and *aliasing* between variables and arguments in the heap [15]. They are used for inferring sound size relations on objects.

2.3 Size Analysis of Java Bytecode

From the RBR, *size* analysis takes care of inferring the relations between the values of variables at different points in the execution. To this end, the notion of *size measure* is crucial. The size of a piece of data at a given program point is an abstraction of the information it contains, which may be fundamental to prove termination. The COSTA system uses several size measures:

- *Integer-value* maps an integer value to its value (i.e., the size of an integer is the value itself). It is typically used in loops with an integer counter.
- *Path-length* [23] maps an object to the length of the maximum path reachable from it by dereferencing. This measure can be used to predict the behavior of loops which traverse linked data structures, such as lists and trees.
- *Array-length* maps an array to its length and is used to predict the behavior of loops which traverse arrays.

Size analysis works in two phases. In the first one, called *abstract compilation*, each bytecode, call or guard is *abstracted* by *linear constraints* on the size of its variables: for example, $\mathsf{iadd}(s_0, s_1, s_0')$ will be abstracted by the constraint $s_0'{=}s_1{+}s_0$, meaning that the size of s_0 after executing the instruction is the sum of the size of s_0 and s_1 before.

In the second phase, linear constraints replacing parts of the program can be propagated via a standard, bottom-up *fixpoint* computation, in order to combine the information about single rules. The goal of this global analysis is to have *size relations* on variables between the input of a rule (i.e., a block in the CFG) and another one which can be (directly or indirectly) called by the first one.

2.4 Inferring Termination

From the RBR and the results of size analysis, a set of *binary clauses* (*BC* in Figure 1) is produced, which capture calls among blocks together with information on how the values of variables change from one call to another. On such binary clauses, standard termination analysis techniques developed for i.e., termination of logic program can be applied. In particular, COSTA proves termination by using semantic-based techniques, relying on *binary unfolding* combined with *ranking functions*, as those in [9]. This is performed by means of the *PUBS* solver. More details on how termination proofs are performed in COSTA can be found in [1].

3 Abstract Interpretation Fundamentals

Before describing the modular analysis framework, a brief description to abstract interpretation is in order. *Abstract interpretation* [12] is a technique for static program analysis in which execution of the program is simulated on a description (or abstract) domain (D) which is simpler than the actual (or concrete) domain (C). Values in the description domain and sets of values in the actual domain are related via a pair of monotonic mappings $\langle \alpha, \gamma \rangle$: *abstraction* $\alpha : 2^C \to D$ and *concretization* $\gamma : D \to 2^C$ which form a Galois connection, i.e.

$$\forall x \in 2^C : \ \gamma(\alpha(x)) \supseteq x \quad \text{and} \quad \forall \lambda \in D : \ \alpha(\gamma(\lambda)) = \lambda.$$

The set of all possible descriptions represents a description domain D which is usually a complete lattice for which all ascending chains are finite. Note that in general $\sqsubseteq$ is induced by $\subseteq$ and α (in such a way that $\forall \lambda, \lambda' \in D : \ \lambda \sqsubseteq \lambda' \Leftrightarrow \gamma(\lambda) \subseteq \gamma(\lambda')$). Similarly, the operations of *least upper bound* ($\sqcup$) and *greatest lower bound* ($\sqcap$) mimic those of 2^C in some precise sense that depends on the particular abstract domain. A description $\lambda \in D$ *approximates* a set of concrete values $x \in 2^C$ if $\alpha(x) \sqsubseteq \lambda$. Correctness of abstract interpretation guarantees that the descriptions computed approximate all of the actual values which occur during the execution of the program.

In COSTA, abstract interpretation is performed on the rule based representation introduced in Section 2. We first introduce some notation. CP and AP stand for descriptions in the abstract domain. The expression $P{:}CP$ denotes a *call pattern*. This consists of a procedure P together with an entry pattern for that procedure. Similarly, $P \mapsto AP$ denotes an answer pattern, though it will be referred to as AP when it is associated to a call pattern $P{:}CP$ for the same procedure. Since a method is represented in the RBR as a set of interconnected procedures that start from a single particular procedure, the same notation will be used for methods: $m{:}CP$ denotes a call pattern that corresponds to an invocation to method m (i.e., the entry procedure for method m), and $m \mapsto AP$ denotes the answer pattern obtained after analyzing method m.

Context-sensitive abstract interpretation takes as input a program R and an initial call pattern $P{:}CP$, where P is a procedure and CP is a restriction of the values of arguments of P expressed as a description in the abstract domain D and computes a set of triples, denoted $analysis(R, P{:}CP) = \{P_1{:}CP_1 \mapsto AP_1, \ldots, P_n{:}CP_n \mapsto AP_n\}$. In each element $P_i{:}CP_i \mapsto AP_i$, P_i is a procedure and CP_i and AP_i are, respectively, the abstract call and answer patterns.

An analysis is said to be *polyvariant* if more than one triple $P{:}CP_1 \mapsto AP_1, \ldots, P{:}CP_n \mapsto AP_n$ $n \geq 0$ with $CP_i \neq CP_j$ for some i, j may be computed for the same procedure P, while a *monovariant* analysis computes (at most) a single triple $P{:}CP \mapsto AP$ for each procedure (with a call pattern CP general enough to cover all possible patterns that appear during the analysis of the program for P).

Although in general context-sensitive, polyvariant analysis algorithms are more precise than those obtained with context-insensitive or monovariant analyses, monovariant algorithms are simpler and have smaller memory requirements. Context-insensitive analysis does not consider call pattern information, and therefore obtains as result of the analysis a set of pairs $\{P_1 \mapsto AP_1, \ldots, P_n \mapsto AP_n\}$, valid for any call pattern.

COSTA includes several abstract interpretation based analyses: nullity and sign are context-sensitive and monovariant, size is context-insensitive, and heap properties analysis [15] is context-sensitive and polyvariant.

4 Extending COSTA to Modular Termination Analysis

As described in Section 2, the termination analysis performed by COSTA is in fact a combination of different processes and analyses that receive as input a

complete program and eventually produce a termination result. Our goal now is to obtain a *modular* analysis framework for COSTA which is able to produce termination proofs by analyzing programs one method at a time. I.e., in order to analyze a method m, we analyze the code of m only and (re-)use the analysis results previously produced for the methods invoked by m.

The communication mechanism used for this work is based on *assertions*, which store the analysis results for those methods which have already been analyzed. Assertions are stored by COSTA in a file per class basis and they keep information regarding the different analyses performed by COSTA: nullity, sign, size, heap properties, and termination.

Same as analysis results, assertions are of the form $m{:}Pre \mapsto Post$, where Pre is the *precondition* of the assertion and $Post$ is the *postcondition*. The precondition states for which call pattern the method has been analyzed. It includes information regarding all domains previously mentioned except size, which is context-insensitive. Pre_D (resp., $Post_D$) denotes the information of the precondition (resp., postcondition) related to analysis domain D. For example, $Pre_{nullity}$ corresponds to the information related to nullity in the precondition Pre. The postcondition of an assertion contains the analysis results for all domains produced after analyzing method m. Furthermore, the assertion also states whether COSTA has proved termination for that method.

In addition to assertions inferred by the analysis, COSTA has been extended to handle assertions written by the user, namely *assumed* assertions. These assertions are relevant for the cases in which analysis is not able to infer some information of interest that we know is correct. This can happen either because the analyzer is not precise enough or because the code of the method is not available to the analyzer, as happens with *native* methods, i.e., those implemented at low-level and for which no bytecode is available.

The user can add assumed assertions with information for any domain. However, for the experiments described in Section 6 assumed assertions have been added manually for providing information about termination only, after checking that the library specification provided by Sun is consistent with the assertion. In assumed assertions where only termination information is available, abstract interpretation-based analyses take $\top$ as the postcondition for the corresponding methods.

4.1 Modular Bottom-up Analysis

The analysis of a Java program using the modular analysis framework consists in analyzing each of the methods in the program, and eventually determining if the program will terminate or not for a given call pattern. Analyzing a method separately presents the difficulty that, from the analysis point of view, the code to be analyzed is *incomplete* in the sense that the code for methods invoked is not available. More precisely, during analysis of a method m there may be calls $m'{:}CP$ and the code for m' is not available. Following the terminology in [14], we refer to determining the value of AP to be used for $m'{:}CP \mapsto AP$ as the *answer patterns problem*.

Several analysis domains existing in COSTA are context-sensitive, and all of them, except heap properties analysis, are monovariant. For simplicity, the modular analysis framework we present is monovariant as well. That means that at most one assertion $m{:}Pre \mapsto Post$ is stored for each method m. If there is an analysis result for m', $m'{:}Pre \mapsto Post$, such that CP is applicable, that is, $CP \sqsubseteq Pre_D$ in the domain D of interest, then $Post_D$ can be used as answer pattern for the call to method m' in m.

For applying this schema, it is necessary that all methods invoked by m have been analyzed already when analyzing method m. Therefore, the analysis must perform a bottom-up traversal of the call graph of the program. In order to obtain analysis information for m' which is applicable during the analysis of m, it is necessary to use a call pattern for m' in its precondition such that it is equal or more general than the pattern actually inferred during the analysis of m. We refer to this as the *call patterns problem*.

Solving the *call* and *answer patterns problems*. A possibility for solving the *call patterns problem* would be to make the modular analysis framework polyvariant: store all possible call patterns to methods in the program and then analyze those methods for each call pattern. This approach has two main disadvantages: on one hand, it is rather complex and inefficient, because all call patterns are stored and every method must be analyzed for all call patterns that appear in the program. On the other hand, it requires performing a fixpoint computation through the methods in the program instead of a single traversal of the call graph, since different call patterns for a method may generate new call patterns for other methods.

Another alternative is a context-insensitive analysis. All methods are analyzed using $\top$ as call pattern for all domains. In this approach, all assertions are therefore applicable, although in a number of cases $\top$ is too general as call pattern for some domains, and the information obtained is too imprecise.

The approach finally used in this work tries to find a balance between both approaches. A monovariant modular analysis framework simplifies a great deal the behavior of the modular analysis, since a single traversal of the call graph is required. In contrast, it is context-sensitive: instead of $\top$, a default call pattern is used, and the result of the analysis is obtained based on this pattern. This framework uses different values as call patterns, depending on the particular analysis being performed. The default call pattern for nullity and sign is $\top$. For Heap properties analysis, in cyclicity it is the pattern that indicates that no argument of the method is cyclic. For variable sharing, it is the one that states that no arguments share.

The default call patterns used for analyzing methods are general enough to be applicable to most invocations used in the libraries and in user programs, solving the *call patterns problem*. However, there can be cases in which the call pattern of an invocation from other method is not included in the default pattern, i. e., $CP \not\sqsubseteq Pre_D$. If the code of the invoked method is available, COSTA will reanalyze it with respect to $CP \sqcup Pre_D$, even though it has been analyzed before for the default pattern. If the code is not available, $\top$ is used as answer pattern.

A potential disadvantage of this approach is that all methods are analyzed with respect to a default call pattern, instead of the specific call pattern produced by the analysis. This means that the analyses in COSTA could produce more precise results when applied non modularly, even though they are monovariant, and it represents a possible loss of precision in the modular analysis framework. Nonetheless, in the experiments performed in Section 6 no method has been found for which it was not possible to prove termination using modular analysis, but it was proved in the non-modular model.

Cycles in the call graph. Analyzing just a method at a time and (re-)using analysis information while performing a bottom-up traversal of the call graph only works under the assumption that there are no cyclic dependencies among methods. In the case where there are strongly connected components (SCCs for short) consisting of more than one method, we can analyze all the methods in the corresponding SCC simultaneously. This presents no technical difficulties, since COSTA can analyze multiple methods at the same time. In some cases, we have found large cycles in the call graph that require analyzing many methods at the same time. In that case a different approach has been followed, as explained in Section 6. Therefore, in COSTA we perform a SCC study first to decide whether there are sets of methods which need to be handled as a unit.

Field-Sensitive Analysis. In some cases, termination of a method depends on the values of variables stored in the heap, i.e., fields. COSTA integrates a field-sensitive analysis [5] which, at least in principle, is a global analysis and requires that the source code of all the program be available to the analyzer. Nevertheless, in order to be able to use this analysis in the modular setting, a preliminary adaptation of that analysis has been performed.

The field-sensitive analysis in COSTA is based on the analysis of program fragments named *scopes*, and modelling those fields whose behaviour can be reproducible using local variables. Fields must satisfy certain conditions in order to be handled as local variables. As a first step of the analysis, related scopes are analyzed in order to determine the fields that are consulted or modified in each scope. Given a method for which performing field-sensitive analysis is required in order to prove termination, an initial approximation to the set of methods that need to be analyzed together is provided by grouping those methods that use the same fields. We have precomputed these sets of methods by means of a non-modular analysis. Since the implementation of this preanalysis is preliminary and can be highly optimized, the corresponding time has not been included in the experiments in Section 6.

5 Application of Modular Analysis to phoneME libraries

We have extended the implementation of COSTA for the modular analysis framework. In order to test its applicability, we have analyzed the core libraries of the phoneME project, a well-known open-source implementation of Java Micro Edition (JavaME). We now discuss the main difficulties associated to the analysis of libraries:

– *Entry points.* Whereas a self contained program has a single entry method (`main(String[])`), a library has many entry points that must be taken into account during the analysis.
– *It is designed to be used in many applications.* Each entry point must be analyzed with respect to a call pattern that represents any *valid* call from any program that might use it. By valid we mean that the call satisfies the precondition of the corresponding method.
– *Large code base.* A system library, especially in the case of Java, usually is a large set of classes that implement most of the features in the source language, leaving only a few specific functionalities to the underlying virtual machine, mainly for efficiency reasons or because they require low-level processing.
– *With many interdependencies.* It is usual that library classes are extensively used from within library code. As a result of this, library code contains a great number of interdependencies among the classes in the library. Thus, non-modular analysis of a library method often results in analyzing a large portion of the library code.
– *Implemented with efficiency in mind.* Another important feature of library code is that it is designed to be as efficient as possible. This means that readability and structured control flow is often sacrified for relatively small efficiency gains. Section 6 shows some examples in phoneME libraries.
– *Classes can be extended and methods overridden.* Using a library in a user program usually not only involves object creation and method invocation, but also library classes can be extended and library methods overridden.
– *Use of native code.* Finally, it is usual that a library contains calls to native methods, implemented in C or inside the virtual machine, and not available to the analyzer.

5.1 Some Further Improvements to COSTA

While trying to apply COSTA to the phoneME libraries, we have identified some problems which we discuss below, together with the solutions we have implemented.

As mentioned above, our approach requires analyzing methods in reverse topological order of the call graph. For this purpose, we extended COSTA in order to produce the call graph of the program after transforming the bytecode to a CFG. The call graph shows the complex structure of the classes in phoneME libraries. Furthermore, apparently, some cycles among methods existed in some of the call graphs, mainly caused by virtual invocations. However, we observed that some potential cycles did not occur in practice. In these cases, either nullity and sign analyses remove some branches if they detect that are unreachable, or COSTA proves termination when solving the binary clauses system. A few cases include a large cycle that involves many methods. Those cycles are formed by small cycles focused in few methods (basically from `Object`, `String` and `StringBuffer` classes), and a large cycle caused by virtual invocations from those methods. In order to speed up analysis, methods in small cycles have been

analyzed at the same time, as mentioned above, and large cycles have been analyzed considering the modular, method at a time bottom up approach.

In addition, COSTA has been extended for a more refined control of which pieces of code we want to include or exclude from analysis. Now there are several *visibility* levels: `method`, `class`, `package`, `application`, and `all`. When `all` is selected, all related code is loaded and included in the RBR. In the other extreme, when `method` is selected only the current method is included in the RBR and only the corresponding assertions are available for other methods.

5.2 An Example of Modular Analysis of phoneME libraries

As an example of the modular analysis framework presented in this paper, let us consider the method `Class.getResourceAsStream` in the phoneME libraries. It takes a string with the name of a resource in the application jar file and returns an object of type `InputStream` for reading from this resource, or `null` if no resource is found with that name in the jar file. Though COSTA analyzes bytecode, we show below the corresponding Java source for clarity of the presentation:

```
public java.io.InputStream getResourceAsStream(String name) {
    try {
        if (name.length() > 0 && name.charAt(0) == '/') {
           name = name.substring(1);
        } else {
           String clName = this.getName();
           int idx = clName.lastIndexOf('.');
           if (idx >= 0)
              name = clName.substring(0, idx+1).replace('.', '/') + name;
        }
        return new com.sun.cldc.io.ResourceInputStream(name);
    } catch (java.io.IOException x) { return null; }
}
```

In the source code of this method there are invocations to eleven methods of different classes (in addition to the eight methods explicitly invoked in the method code, the string concatenation operator in line 9 is translated to a creation of a fresh `StringBuffer` object and invocations to some of its methods.)

If the standard, non-modular approach of analysis is used, the analyzer would load the code of this method and all related methods invoked. In this case, there are 65 methods related to `getResourceAsStream`, from which 10 are native methods. In fact, using this approach COSTA is unable to prove termination.

Using modular analysis, the call graph is traversed bottom-up, analyzing each method related to `getResourceAsStream` one by one. For example, the analysis of the methods invoked by `getResourceAsStream` has obtained the following information related to the nullity domain[1]:

[1] These analysis results have been obtained ignoring possible exceptions thrown by the Java virtual machine (e.g., no method found, unable to create object, etc.) for clarity of the presentation.

Method	call	result
StringBuffer.toString()	n/a	nonnull
StringBuffer.append(String)	$\top$	nonnull
StringBuffer.<init>()V	n/a	n/a
String.replace(char,char)	$(\top, \top)$	nonnull
com.sun.cldc.io.ResourceInputStream.<init>(String)	nonnull	n/a
String.substring(int)	$\top$	nonnull
String.length()	n/a	$\top$
String.substring(int,int)	$(\top, \top)$	nonnull
String.charAt(int)	$\top$	$\top$

In this table, the call pattern refers to nullity information regarding the values of arguments and the result is related to the method return value. Despite of the call patterns generated by the analysis of `getResourceAsStream` shown above, when the bottom-up modular analysis computation is performed, all methods are analyzed with respect to the default call pattern $\top$. The analysis of `getResourceAsStream` uses the results obtained for those methods to generate the nullity analysis results for `getResourceAsStream`. The same mechanism is used for other domains: sign, size and heap related properties.

Finally, two native methods are invoked from `getResourceAsStream` (`lastIndexOf` and `getName`) that require assumed assertions. In this case, $\top$ is assumed as the answer pattern for those invocations.

5.3 Contracts for Method Overriding

As mentioned above, one of the most important features of libraries in OO languages is that classes can be extended by users at any point in time, including the possibility of overriding methods. This poses significant problems to modular static analysis, since classes and methods which have already been analyzed may be extended and overridden, thus possibly rendering the previous analysis information incorrect. Let us illustrate this issue with an example:

```
class A {
   void m(){/* code for A.m() */};
   void caller_m(){this.m();};
};
class B extends A {
   void m(){/* code for B.m() */};
};
```

```
class Example {
   void method_main(A a){
      a.caller_m();
   };
};
```

Here, there are three different classes: `A`, `B`, and `Example`. But for now, let us concentrate on classes `A` and `Example` only. If `A` is analyzed, the result obtained for `caller_m` depends directly on the result obtained for `A.m` (for instance, `caller_m` could be guaranteed to terminate under the condition that `A.m` terminates). Then, the class `Example` is analyzed, using the analysis results obtained for `A`. Let us suppose that analysis concludes that `method_main` terminates.

Now, let us suppose that `B` is added to the program. As shown in the example, `B` extends `A` and overrides `m`. Imagine now that the analysis concludes that the

new implementation of `m` is not guaranteed to terminate. The important point now is that the termination behavior of some of the methods we have already analyzed can be altered, and we have to make sure that analysis results can correctly handle this situation. In particular, `caller_m` is no longer guaranteed to terminate, and the same applies to `method_main`.

Note, however, that class inheritance is correctly handled by the analyzer if all the code (in this case the code of `B`) is available from the beginning. This is done by considering, at the invocation program point, the information about both implementations of `m`.

However, in general, the analyzer does not know, during the analysis of `A`, that the class will be extended by `B`. Such a situation is very common in the analysis of libraries, since they must be analyzed without knowing which user-defined classes will override their methods. In this example, corresponds to `A` and `Example` being library classes and `B` being defined by the user.

In order to avoid this kind of problems, the concept of *contract* can be used (in the sense of *subcontracting* of [20]). This means that the analysis result for a given method `m` is taken as the contract for m, i.e., information about how `m` and any redefinition of it is supposed to behave with respect to the analysis of interest.

A *contract*, same as an assertion, has two parts: the calling *preconditions* which must hold in order the contract can be applicable; and the *postcondition*, the result of the analysis with respect to that preconditions. For example, a contract for `A.m()` may say that it terminates under the condition that the *this* object of type `A` is an acyclic data structure.

In the example above, when `B` is added to the program, we have to analyze `B.m` taking as call pattern the precondition (Pre) in the contract for `A.m`. This guarantees that the result obtained for `B.m` will be valid in the same set of input states as the contract for `A.m`. Then, we need to compare the postconditions. If $m_B{:}Pre \mapsto Post^B$ and $m_A{:}Pre \mapsto Post^A$ are the assertions generated for `B.m` and `A.m`, respectively, and Pre is the default calling pattern for both implementations, there are two possible cases:

1. If $Post^B \sqsubseteq Post^A$ then `B.m` satisfies the contract for `A.m`.
2. Otherwise, the contract cannot be applied. The user can manually inspect the code of `B.m` and if the analyzer loses precision, add an assumed assertion for `B.m`. Otherwise, `B.m` is considered incorrect.

Interfaces and abstract methods are similar to overriding methods of a superclass, with the difference that there is no code to analyze in order to generate the contract. In this case, assumed assertions written by the user can be used as contracts.

6 Experiments

After obtaining the call graph for the classes of phoneME's java.lang package, a bottom-up traversal of the call graphs has been performed. In a few particular cases, it was required to enable other analyses included in COSTA (e.g., field

Class	Modular				Non Modular			Assumed		Related	
	$\#B_c$	#T	T_{cg}	$Time_a$	$\#B_c$	#T	$Time_a$	Nat	NNat	1st	All
Boolean	56	6	0.02	0.19	67	6	0.22	0	0	1	1
Byte	59	7	0.40	0.22	1545	7	21.10	0	0	4	22
Character	64	11	0.16	0.27	513	11	1.03	0	0	6	11
Class	110	4	1.17	1.10	4119	3	842.70	11	1	20	58
Double	107	17	3.66	1.12	107	13	0.36	2	0	8	57
Error	7	2	0.02	0.04	60	2	0.12	0	0	2	4
FDBigInt	1117	14	0.80	16.10	2513	12	158.39	0	2	23	47
Float	106	18	3.74	1.16	3105	15	5674.96	2	0	9	60
FloatingDecimal	3028	12	4.32	1201.10	3402	9	4983.88	0	8	49	64
Integer	469	21	1.35	18.76	4519	21	62.51	0	0	7	20
Long	268	11	0.64	10.99	2164	11	36.08	0	0	7	20
Math	207	16	0.14	0.67	212	16	0.69	6	0	3	3
NoClassDefFoundError	7	2	0.02	0.04	108	2	0.13	0	0	2	6
Object	737	3	0.21	46.21	891	3	129.31	5	0	7	28
OutOfMemoryError	7	2	0.02	0.03	170	2	0.18	0	0	2	8
Runtime	14	3	0.02	0.08	27	3	0.08	4	0	1	1
Short	59	7	0.39	0.24	1545	7	20.83	0	0	4	22
String	1784	39	5.88	21.11	8709	32	7217.43	6	3	34	120
StringBuffer	1509	37	6.74	11.01	14206	33	12103.35	0	0	37	86
System	45	7	0.38	0.31	2778	6	4864.33	5	0	11	62
Throwable	615	4	0.16	1.23	628	4	60.54	2	0	6	22
VirtualMachineError	7	2	0.02	0.04	108	2	0.14	0	0	2	6
Exception Classes (18)	136	38	0.61	0.74	3961	38	21.27	0	0	11	18
com/sun/* (7)	1584	26	5.55	22.36	11293	16	5161.29	0	0		
java/io/* (8)	106	11	1.47	0.65	2337	9	4983.35	0	0		
java/util/* (3)	265	13	0.88	3.33	2171	12	51.93	0	0		
Total	**12473**	**333**	**38.77**	**1359.10**	**71258**	**295**	**46396.17**	**43**	**14**	**256**	**746**

Table 1. Termination Analysis for java.lang package in COSTA (execution times are in seconds).

sensitive analysis [5], as mentioned above) for proving termination, or disabling some features such as handling jvm exceptions.

Table 1 shows the results of termination analysis of java.lang package, plus some other packages used by java.lang. This table compares the analysis using the modular analysis described in this paper with the non-modular analysis previously performed by COSTA.

The columns under **Modular** show the modular analysis results, while under the **Non Modular** heading non-modular results are shown. $\#\mathbf{B}_c$ shows the number of bytecode instructions analyzed for all methods in the corresponding class, **#T** shows the number of methods of each class for which COSTA has proved termination and $\mathbf{Time}_a$ shows the analysis time of all the methods in each class. In the modular case, the total analysis time is $\mathbf{Time}_a$ plus $\mathbf{T}_{cg}$, the time spent building the call graph of each class. The two columns under **Assumed** show the number of methods for which assumed assertions were required: **Nat** is the number of native methods in each class, and **NNat** contains the number of non-native methods that could not be proved terminating. Finally, the last two columns under **Related** contain the number of methods from other classes that are invoked by the methods in the class, either directly, shown in **1st** or the total number of methods transitively invoked, shown in **All**. Some rows in the table contain results accumulated for a number of classes (in parenthesis). The last three rows in the table contain accumulated information for methods directly or transitively invoked by the java.lang package which belong to phoneME packages

other than java.lang. These rows do not include information about **Related** methods, since they are already taken into account in the corresponding columns for java.lang classes. The last row in the table, **Total**, shows the addition for all classes of all figures in each column. A number of interesting conclusions can be obtained from this table. Probably, the most relevant result is the large difference between the number of bytecode instructions which need to be analyzed in the modular and non-modular cases: 12,473 vs 71,258 instructions, i.e. nearly 7 times more code needs to be analyzed in the non-modular approach. The reason for this is that though in the modular approach methods are (at least in principle) analyzed just once, in the non-modular approach methods which are required for the analysis of different initial methods are analyzed multiple times. Obviously, this difference in code size to be analyzed has a great impact on the analysis times: the **Total** row shows that the modular analysis of all classes in java.lang is more than 30 times faster than the non-modular case.

Another crucial observation is that by using the modular approach we have been able to prove termination of 38 methods for which the non-modular approach is not able, either because the analysis runs out memory or because it fails to produce results within a reasonable time. Furthermore, the modular approach in this setting has turned out to be strictly more precise than the non-modular approach, since for all cases where the non-modular approach has proved termination, it has also been proved by the modular approach. This results in 333 methods for which termination has been proved in the modular approach, versus 295 in the non-modular approach.

Altogether, in our experiments we have tried to prove termination of 389 methods. In the studied implementation of JavaME, 43 of those methods are native. Therefore, COSTA could not analyze them, and assumed assertions have been added for them. In addition, COSTA was not able to prove termination of 14 methods, neither in the modular nor non-modular approaches, as shown in the **NNat** column. For these methods, assumed assertions have also been added, and have not been taken into account in the other columns except in the last two ones. These two columns provide another view on the difference between using modular and non-modular analyses with respect to the number of transitively invoked methods (746) that required analysis, w.r.t. those directly invoked (256). In the modular case, only directly invoked methods need to be considered, and only for loading their assertions, whereas the non-modular approach requires loading (and analyzing) all related methods.

We now describe in more detail the methods whose termination has not been proved by COSTA and the reasons for this.

Bitwise operations. The size analysis currently available in COSTA is not capable of tracking numeric values after performing bitwise operations on them. Therefore, we cannot prove termination of some library methods which perform bitwise operations (in most cases, right or left shift operations) on variables which affect a loop termination condition.

Arrays issues. During size analysis, arrays are abstracted to their size. Though this is sufficient for proving termination of many loops which traverse arrays,

termination cannot be proved for loops whose termination depends on the value of specific elements in the array, since such values are lost by size abstraction.

Concurrency. Though it is the subject of ongoing work, COSTA does not currently handle concurrent programs. Nonetheless, it can handle Java code in which `synchronized` constructs are used for preventing thread interferences and memory inconsistencies. In particular, few `java.lang` phoneME classes make real use of concurrency. For this reason, `Thread` class has not been included in the test, neither Table 1 does include information regarding `Class.initialize` nor `wait` methods defined in `Object`.

Unstructured control flow. There are some library methods in which the control flow is unstructured, apparently for efficiency reasons. For example, `String.indexOf` uses a *continue* statement wrapping several nested loops, the outer most of them being an endless loop as in the following code (on the left):

```
indexOf(String str, int i){
 ...
 searchForFChar:
  while (true) {
   ...
   if (i > max) return -1;
   while (j < end) {
    if (v1[j++] != v2[k++]){
     i++; continue searchForFChar;}}
   return i - offset;} }
```

```
fixResourceName(String n){
 int stI = 0;
 int e = 0;

 while((e=n.indexOf('/',stI))!= -1){
  if (e == stI) {
   stI++;  continue;}
    .... } } }
```

Other Cases. `ResourceInputStream.fixResourceName` involves a call to a native method in the loop condition (see code above on the right). A termination assertion is not enough to find a ranking function of the loop to prove termination.

7 Discussion

Modular analysis has received considerable attention in different programming paradigms, ranging from, e.g., logic programming [14, 11, 8] to object-oriented programming [22, 6, 19]. A general theoretical framework for modular abstract interpretation analysis was defined in [13], but most of the existing works regarding modular analysis have focused on specific analyses with particular properties and using more or less ad-hoc techniques.

A previous work from some of the authors of this paper presents and empirically tests a modular analysis framework for logic programs [14, 11]. There are important differences with this paper: in addition to the programming paradigm, the framework of [14] is designed to handle one abstract domain, while the framework presented in this paper handles several domains at the same time, and the previous work is based on `CiaoPP`, a polyvariant context-sensitive analyzer in which an intermodular fixpoint algorithm was performed.

In [22] a control-flow analysis-based technique is proposed for call graph construction in the context of OO languages. Although there have been other works

in this area, the novelty of this approach is that it is context-sensitive. Also, [6] shows a way to perform modular class analysis by translating the OO program into *open* DATALOG programs. In [19] an abstract interpretation based approach to the analysis of class-based, OO languages is presented. The analysis is split in two separate semantic functions, one for the analysis of an object and another one for the analysis of the context that uses that object. The interdependence between context and object is expressed by two mutually recursive equations. In addition, it is context-sensitive and polyvariant.

As conclusion, in this work we have presented an approach which is, to the best of our knowledge, the first modular termination analysis for OO languages. Our approach is based on the use of assertions as communication mechanism between the analysis of different methods. The experimental results show that the approach increases the applicability of termination analysis. The flexibility of this approach allows a higher level of scalability and makes it applicable to component-based systems, since is not required that all code be available to the analyzer. Furthermore, the specification obtained for a component can be reused for any other component that uses it.

It remains as future work to extend the approach to other intermediate cases between modular and global analysis, i.e., by allowing analysis of several methods as one unit, even if they are not in the same cycle. This can be done without technical difficulties and it should be empirically determined what granularity level results in more efficient analysis.

Acknowledgments

The authors would like to thank Damiano Zanardini for interesting discussions and for his help with the heap analysis in COSTA. This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, by the Spanish Ministry of Science and Innovation (MICINN) under the TIN-2008-05624 *DOVES* project, the TIN2008-04473-E (Acción Especial) project, the HI2008-0153 (Acción Integrada) project, the UCM-BSCH-GR58/08-910502 Research Group and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS* project.

References

1. E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination Analysis of Java Bytecode. In *FMOODS*, LNCS 5051, pages 2–18, 2008.
2. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In *ESOP'07*, LNCS, 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, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Resource Usage Analysis and its Application to Resource Certification. In *FOSAD 2007/2008/2009 Tutorial Lectures*, LNCS 5705, pages 258–288. Springer, 2009.

5. Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Diana Ramírez. From Object Fields to Local Variables: A Practical Approach to Field-Sensitive Analysis. In *SAS 2010 Proceedings*, LNCS. Springer, 2010.
6. F. Besson and T. Jensen. Modular class analysis with datalog. In *10th International Symposium on Static Analysis, SAS 2003*, number 2694 in LNCS. Springer, 2003.
7. A.R. Bradley, Z. Manna, and H.B. Sipma. Termination of polynomial programs. In *VMCAI*, 2005.
8. M. Codish, S. K. Debray, and R. Giacobazzi. Compositional analysis of modular logic programs. In *Proc. POPL'93*, 1993.
9. M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *J. Log. Program.*, 41(1):103–123, 1999.
10. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.
11. J. Correas, G. Puebla, M. Hermenegildo, and F. Bueno. Experiments in Context-Sensitive Analysis of Modular Programs. In *LOPSTR'05*, number 3901 in LNCS, pages 163–178. Springer-Verlag, April 2006.
12. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL'77*, pages 238–252. ACM, 1977.
13. P. Cousot and R. Cousot. Modular Static Program Analysis, invited paper. In *Compiler Construction*, 2002.
14. G. Puebla et al. A Generic Framework for Context-Sensitive Analysis of Modular Programs. In M. Bruynooghe and K. Lau, editors, *Program Development in Computational Logic, A Decade of Research Advances in Logic-Based Program Development*, number 3049 in LNCS, pages 234–261. Springer-Verlag, August 2004.
15. Samir Genaim and Damiano Zanardini. The acyclicity inference of COSTA. In *11th International Workshop on Termination*, July 2010.
16. J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. In *IJCAR*, 2006.
17. C.S. Lee, N.D. Jones, and A.M. Ben-Amram. The size-change principle for program termination. In *POPL'01*, pages 81–92. ACM, 2001.
18. N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs. In *ICLP*, 1997.
19. Francesco Logozzo. Separate Compositional Analysis of Class-based Object-oriented Languages. In *AMAST'2004*, volume 3116 of *LNCS*, pages 332–346. Springer-Verlag, July 2004.
20. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2nd edition, 1997.
21. C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Termination Analysis of Java Bytecode by Term Rewriting. In Johannes Waldmann, editor, *WST'09*, Leipzig, Germany, June 2009.
22. Christian W. Probst. Modular Control Flow Analysis for Libraries. In *Static Analysis Symposium, SAS'02*, volume 2477 of *LNCS*, pages 165–179. Springer-Verlag, 2002.
23. F. Spoto, P.M. Hill, and E. Payet. Path-length analysis of object-oriented programs. In *EAAI'06*, ENTCS. Elsevier, 2006.
24. F. Spoto and T. Jensen. Class analyses as abstract interpretations of trace semantics. *ACM Trans. Program. Lang. Syst.*, 25(5):578–630, 2003.
25. F. Spoto, F. Mesnard, and É. Payet. A Termination Analyser for Java Bytecode based on Path-Length. *ACM TOPLAS*, 32(3), 2010.

Appendix C

Asymptotic Resource Usage Bounds

The paper “Asymptotic Resource Usage Bounds” [5] follows.

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 φ, ϕ or ψ, 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 φ_1 implies the linear constraint φ_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, $\mathsf{nat} : \mathbb{Z} \to \mathbb{N}$ 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 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) $\langle C_m(i,n) = 3$
	$, \varphi_1 = \{i \geq n\}\rangle$
	(2) $\langle C_m(i,n) = 15 + C_g(i,n) + C_m(i',n)$
	$, \varphi_2 = \{i < n, i' = i+1\}\rangle$
	(3) $\langle C_m(i,n) = 17 + C_g(0,i) + C_m(i,n')$
	$, \varphi_3 = \{i < n, , n' = n-1\}\rangle$

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* φ *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:

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

where $\mathsf{l_b}$ and $\mathsf{l_r}$ 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 $\mathsf{l_r} = 2^{\mathsf{nat}(n-i)} - 1$ times, which in this paper, we simplify to $\mathsf{l_r} = 2^{\mathsf{nat}(n-i)}$ 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:

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

where 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 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 φ_2 by $\varphi_2' = \{i<n, i' = i*2\}$ and φ_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:

$$worst(\{rec_1, \ldots, rec_{nr}\}) = \max(maximize(\mathcal{I}, \{rec_1, \ldots, rec_{nr}\}))$$

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 $worst(\{rec_1, rec_2\}) = \max(maximize(\mathcal{I}, \{\mathsf{nat}(n-i), \mathsf{nat}(i)\}))$ which results in $worst(\{rec_1, rec_2\}) = \max(\{\mathsf{nat}(n_0 - i_0), \mathsf{nat}(n_0 - 1)\})$. 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 Θ 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 ∞ when the input is ∞. 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 ∞, i.e., when, for example, n goes to ∞ and i stays constant, but not when both n and i go to ∞. 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 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* 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 nat-free expressions. Note that by obtaining their 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 φ associated to the cost expression can sometimes be used to check whether some 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 nat-expressions (represented by their corresponding nat-variables A and B), we say that $\varphi{\models}A \succeq B$ if A asymptotically subsumes B when both go to ∞.

4 Asymptotic Orders of Cost Expressions

As it is well-known, by using Θ 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 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 nat-free cost expression $\tilde{e}$, we describe how to simplify it and obtain another 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* 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 nat-free cost expression of Ex. 3 results in: $\tau(\tilde{e}) = A*(A^2*B^4 + 3^B*A^2) + \log(A)*2^B*\log(B)*C$. □

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

Once the τ 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* 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 nat-free cost expression). *A* normalized 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* 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 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 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 $\{\{A^3, B^4\}, \{2^{\log(3)*B}, A^3\}, \{\log(A), 2^B, \log(B), C\}\}$ □

4.2 Asymptotic Subsumption

Given a normalized nat-free cost expression $\tilde{e} = \{M_1, \ldots, M_n\}$ and a context constraint φ, 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, φ 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 $1 \leq i \leq m$ 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) *$r \geq \Sigma_{i=1}^{m} \mathsf{pow}(b_i)$ 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 $r \geq \Sigma_{i=1}^{m} \mathsf{deg}(b_i)$; or*
   (b) *there is at least one b_i of the form $\log(A_i)$, and $r \geq 1 + \Sigma_{i=1}^{m} \mathsf{deg}(b_i)$*
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 ∞ 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 $\varphi = \{x \geq y, x \geq 0, y \geq 0\}$. M is asymptotically subsumed by b since $\varphi \models (A \succeq B) \wedge (A \succeq C)$, and condition 2b in Lemma 2 holds. □

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 nat-free cost expression in M_1. Note that M_1 can contain additional basic nat-free cost expressions which are not needed for subsuming M_2.

Lemma 3. *Let M_1 and M_2 be two products, and φ a context constraint. If there exists a partition of M_2 into k sets $P_1, \ldots, P_k$, and k distinct basic* 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 φ 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 $M_2' \in O(M_1)$. □

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* nat*-variable by the corresponding* 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 $e \in \Theta(\mathtt{asymp}(e))$. □

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 φ, 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 φ 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 $\mathsf{nat}(n_0{+}m_0)$. □

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: $C^+_{asymp}(a,b) = 2^{\mathsf{nat}(a+b)} * \mathsf{nat}(2a+3b)$. □

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 Θ. 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 "×", 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*. Springer, 2008.
2. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *ESOP*, volume 4421 of *LNCS*, 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.

Appendix D

Comparing Cost Functions in Resource Analysis

The paper "Comparing Cost Functions in Resource Analysis" [6] follows.

Comparing Cost Functions in Resource Analysis

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

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

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

1 Introduction

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

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

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

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

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

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

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

2 Cost Functions

Let us introduce some notation. The sets of natural, integer, real, non-zero natural and non-negative real values are denoted by $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, $\mathbb{N}^+$, and $\mathbb{R}^+$, respectively. We write x, y, and z, to denote variables which range over $\mathbb{Z}$. A *linear*

expression has the form $v_0 + v_1x_1 + \ldots + v_nx_n$, where $v_i \in \mathbb{Z}$, $0 \leq i \leq n$. Similarly, a *linear constraint* (over $\mathbb{Z}$) has the form $l_1 \leq l_2$, where l_1 and l_2 are linear expressions. For simplicity we write $l_1 = l_2$ instead of $l_1 \leq l_2 \wedge l_2 \leq l_1$, and $l_1 < l_2$ instead of $l_1 + 1 \leq l_2$. Note that constraints with rational coefficients can be always transformed into equivalent constraints with integer coefficients, e.g., $\frac{1}{2}x > y$ is equivalent to $x > 2y$. The notation $\bar{t}$ stands for a sequence of entities $t_1, \ldots, t_n$, for some $n>0$. We write φ, ϕ or ψ, to denote sets of linear constraints which should be interpreted as the conjunction of each element in the set. An assignment σ over a tuple of variables $\bar{x}$ is a mapping from $\bar{x}$ to $\mathbb{Z}$. We write $\sigma \models \varphi$ to denote that $\sigma(\varphi)$ is satisfiable.

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

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

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

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

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

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

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

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

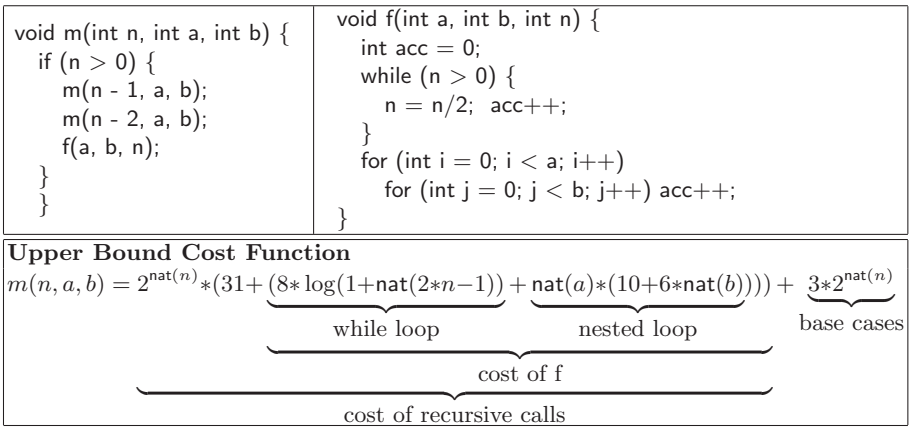


Fig. 1. Running example and upper bound obtained by COSTA on the number of executed bytecode instructions.

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

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

3 Comparison of Cost Functions

In this section, we state the problem of comparing two cost functions represented as cost expressions. As we have seen in Ex. 1, a cost function $\langle e, \varphi \rangle$ for a method m is a single cost expression which approximates the cost of any possible execution of m which is consistent with the context constraints φ. This can be done by means of nat subexpressions which encapsulate conditions on the input data sizes in a single cost expression. Besides, cost functions often contain max sub-

expressions, e.g., $\langle \mathsf{max}(\{\mathsf{nat}(x) * \mathsf{nat}(z), \mathsf{nat}(y) * \mathsf{nat}(z)\}), \mathit{true}\rangle$ which represent the cost of disjunctive branches in the program (e.g., the first sub-expression might correspond to the cost of a then-branch and the second one the cost of the else-branch of a conditional statement).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4 Inclusion of Cost Functions

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

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

4.1 Normalization Step

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

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

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

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

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

□

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

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

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

□

4.2 Product Comparison

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

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

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

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

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

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

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

In order to compare two products, first we apply the above operator τ_* iteratively until there are no more terms to transform. In each iteration we pick e_1 and e_2 and modify $\mathcal{M}_1$ and $\mathcal{M}_2$ accordingly, and then repeat the process on the new

sets. The result of this operation is denoted $fp_*(\mathcal{M}_1, \mathcal{M}_2)$. This process is finite because the size of $\mathcal{M}_1$ strictly decreases at each iteration.

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

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

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

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

4.3 Comparison of Sums of Products

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

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

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

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

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

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

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

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

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

4.4 Composite Comparison of Sums of Products

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

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

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

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

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

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

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

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

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

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

5 Implementation and Experimental Evaluation

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

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

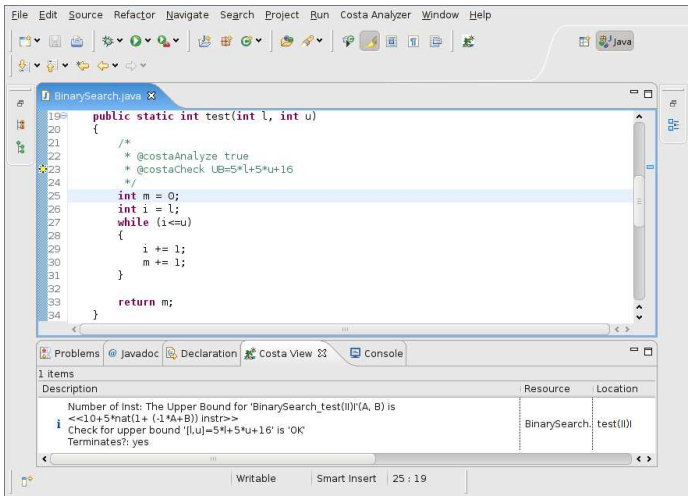


Fig. 2. Screenshot of the COSTA plugin for Eclipse, showing how annotations are used to interact with COSTA

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

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

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

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

Table 2. Experiments in Cost Function Comparison

T_1 Time taken by the comparison $CF \preceq rev(CF)$, where $rev(CF)$ is just the reversed version of CF. I.e., $rev(x + y + 1) = 1 + x + y$. The size of the expressions involved in the comparison is shown in the fifth column of the table (Size_1).

T_2 Time taken by the comparison $CF + CF \preceq CF * CF$, assuming that CF takes at least the value 2 for all input values. In this case, the size of the expression grows considerably and hence the comparison takes a longer time than the previous case. The size of the largest expression in this case is shown in the sixth column of the table (Size_2).

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

6 Other Approaches and Related Work

In this section, we discuss other possible approaches to handle the problem of comparing cost functions. In [7], an approach for inferring non-linear invariants

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

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

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

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

Another approach for this problem is based on numerical methods since our problem is analogous to proving whether $0 \preceq b - f_m$ in the context ϕ_b. There are at least two numerical approaches to this problem. The first one is to find the roots of $b - f_m$, and check whether those roots satisfy the constraints ϕ_b. If they do not, a single point check is enough to solve the problem. This is because, if the equation is verified at one point, the expressions are continuous, and there is no sign change since the roots are outside the region defined by ϕ_b, then we can ensure that the equation holds for all possible values satisfying ϕ_b. However, the problem of finding the roots with multiple variables is hard in general and often not solvable. The second approach is based on the observation that there is no need to compute the actual values of the roots. It is enough to know whether there are roots in the region defined by ϕ_b. This can be done by finding the minimum values of expression $b - f_m$, a problem that is more affordable using numerical methods [9] . If the minimum values in the region

defined by ϕ_b are greater than zero, then there are no roots in that region. Even if those minimum values are out of the region defined by ϕ_b or smaller than zero, it is not necessary to continue trying to find their values. If the algorithm starts to converge to values out of the region of interest, the comparison can be proven to be false. One of the open issues about using numerical methods to solve our problem is whether or not they will be able to handle cost functions output from realistic programs and their performance. We have not explored these issues yet and they remain as subject of future work.

7 Conclusions

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

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

References

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

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

Appendix E

More Precise yet Widely Applicable Cost Analysis

The paper "More Precise yet Widely Applicable Cost Analysis" [14] follows.

More Precise yet Widely Applicable Cost Analysis

Elvira Albert[1], Samir Genaim[1], and Abu Naser Masud[2]

[1] DSIC, Complutense University of Madrid (UCM), Spain
[2] DLSIIS, Technical University of Madrid (UPM), Spain

Abstract. Cost analysis aims at determining the amount of resources required to run a program in terms of its input data sizes. Automatically inferring *precise* bounds, while at the same time being able to handle a *wide* class of programs, is a main challenge in cost analysis. (1) Existing methods which rely on computer algebra systems (*CAS*) to solve the obtained *cost recurrence equations* (*CR*) are very precise when applicable, but handle a very restricted class of *CR*. (2) Specific solvers developed for *CR* tend to sacrifice accuracy for wider applicability. In this paper, we present a novel approach to inferring precise *upper* and *lower* bounds on *CR* which, when compared to (1), is strictly more widely applicable while precision is kept and when compared to (2), is in practice more precise (obtaining even tighter complexity orders), keeps wide applicability and, besides, can be applied to obtain useful lower bounds as well. The main novelty is that we are able to accurately bound the worst-case/best-case cost of each iteration of the program loops and, then, by summing the resulting sequences, we achieve very precise upper/lower bounds.

1 Introduction

Static cost analysis [13] aims at automatically inferring the resource consumption (or cost) of executing a program as a function of its input data sizes. The classical approach to cost analysis consists of two phases. First, given a program and a *cost model*, the analysis produces *cost relations* (*CRs*), i.e., a system of recursive equations which capture the cost of the program in terms of the size of its input data. Let us motivate our work on the contrived example depicted in Fig. 1a. The example is sufficiently simple to explain the main technical parts of the paper, but still interesting to understand the challenges and precision gains. For this program and the *memory consumption* cost model, the cost analysis of [3] generates the *CR* which appears in Fig. 1b. This cost model estimates the number of objects allocated in the memory. Cost analyzers are usually parametric on the cost model, e.g., cost models widely used are the number of executed bytecode instructions, number of calls to methods, etc. Observe that the structure of the Java program and its corresponding *CR* match. The equations for C correspond to the `for` loop, those of B to the inner `while` loop and those of A to the outer `while` loop. The recursive equation for C states that the memory consumption of executing the inner loop with $\langle k, j, n\rangle$ such that $k{<}n{+}j$ is 1 (one object) plus

```
void f(int n) {
 List l = null;
 int i=0;
 while (i<n) {
   int j=0;
   while ( j<i ) {
     for(int k=0;k<n+j;k++)
       l=new List(i*k*j,l);
     j=j+random()?1:3;}
   i=i+random()?2:4;
 }  }
```

(a) Running Example

$$
\begin{array}{lll}
F(n) & = A(0,n) & \{\} \\
A(i,n) & = 0 & \{i \geq n\} \\
A(i,n) & = B(0,i,n) + A(i',n) & \\
 & \{\ i < n, i+2 \leq i' \leq i+4\} & \\
B(j,i,n) & = 0 & \{j \geq i\} \\
B(j,i,n) & = C(0,j,n) + B(j',i,n) & \\
 & \{j < i, j+1 \leq j' \leq j+3\} & \\
C(k,j,n) & = 0 & \{k \geq n+j\} \\
C(k,j,n) & = 1 + C(k',j,n) & \\
 & \{k' = k+1, k < n+j\} &
\end{array}
$$

(b) *CRs* for Memory Consumption

Fig. 1: Running Example and its Cost Relation System

that of executing the loop with $\langle k', j, n\rangle$ where k'=k+1. The recursive equation for B states that executing the loop with $\langle j, i, n\rangle$ costs as executing $C(0, j, n)$ plus executing the same loop with $\langle j', i, n\rangle$ where j+1≤j'≤j+3. While, in the Java program, j' can be either j+1 or j+3, due to the static analysis, the case for j+2 is added in order to have a convex shape [7]. The process of generating *CRs* heavily depends on the programming language and, thus, multiple analyses have been developed for different paradigms. However, the resulting *CRs* are a common target of cost analyzers.

Our work focuses on the second phase of cost analysis: once *CRs* are generated, analyzers try to compute *closed-forms* for them, i.e., cost expressions which are not in recursive form. Two main approaches exist: (1) Since cost relations are syntactically quite close to *recurrence relations*, most cost analysis frameworks rely on existing *Computer Algebra Systems* (*CAS*) for finding closed-forms. Unfortunately, only a restricted class of *CRs* can be solved using *CAS*, namely only some of those which have an exact solution. In practice, this seldom happens. For instance, in the cost relation B, variable j' can increase by one, by two or by three at each iteration, so an exact cost function which captures the cost of any possible execution does not exist. (2) Instead, specific upper-bound solvers developed for *CRs* try to reason on the worst-case cost and obtain sound *upper-bounds* (UBs) of the resource consumption. As regards *lower-bounds* (LBs), due in part to the difficulty of inferring under-approximations, general solvers for *CRs* able to obtain useful approximations of the best-case cost have not been developed yet. As regards the number of iterations, for B, the worst-case (resp. best-case) cost must assume that j' increases by one (resp. three) at each iteration. Besides, there is the problem of bounding the cost of each of the iterations. For UBs, the approach of [2] assumes the worst-case cost for all loop iterations. E.g., an UB on the cost of any iteration of B is n_0+i_0−1, where n_0 and i_0 are respectively the initial values for n and i. This corresponds to the memory allocation of the last iteration of the corresponding `while` loop. This approximation, though often imprecise, makes it possible to obtain UBs for most *CRs* (and thus

programs). Observe that it is not useful to obtain LBs since by assuming the best-case cost for all iterations, the obtained LB would be in most cases zero.

Needless to say, precision is fundamental for most applications of cost analysis. For instance, UBs are widely used to estimate the space and time requirements of programs execution and provide resource guarantees [8]. Lack of precision can make the system fail to prove the resource usage requirements imposed by the software client. LBs are used to scheduling the distribution of tasks in parallel execution. Likewise, precision will be essential to achieve a satisfactory scheduling. A main achievement in this paper is the seamless integration of both approaches so that we get the best of both worlds: precision as (1), whenever possible, while applicability as close to (2) as possible. Intuitively, the precision gain stems from the fact that, instead of assuming the worst-case cost for all iterations, we infer tighter bounds on each of them in an automatic way and then approximate the summation of the sequence. For UBs, we do so by taking advantage of existing automatic techniques, which are able to infer UBs on the number of loop iterations and the worst-case cost of all of them, in order to generate a novel form of *(worst-case) recurrence relations* which can be solved by *CAS*. The exact solution of such recurrence relation (*RR*) is guaranteed to be a precise UB of the original *CR*. As another contribution, we present a new technique for inferring LBs on the number of iterations. Then, the problem of inferring LBs on the cost becomes dual to the UBs.

To the best of our knowledge, this is the first general approach to inferring LBs from *CRs* and, as regards UBs, the one that achieves a better precision vs. applicability balance. Importantly, when *CRs* originate from nested loops in which the cost of the inner loop depends on the outer loop, our approach obtains more precise bounds than [9, 2]. Moreover, as our experiments show, we are able to produce upper bounds with a tighter complexity order than those inferred by [9, 2], e.g., improving from $O(n * log(n))$ to $O(n)$. On the other hand, when compared to [10], our approach is of wider applicability in the sense that it can infer general polynomial, exponential and logarithmic bounds, not only univariate polynomial bounds as [10]. Since *CRs* obtained from different programming languages have the same features, our work is applicable to cost analysis of any language. Preliminary experiments on Java (bytecode) programs confirm the good balance between the accuracy and applicability of our analysis.

2 Preliminaries

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, 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 $v_i{\in}\mathbb{Z}$. A *linear constraint* (over $\mathbb{Z}$) has the form $l_1{\leq}l_2$, where l_1 and l_2 are linear expressions. 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 $l_1{+}1{\leq}l_2$. We use $\bar{t}$ to denote a sequence of entities $t_1, \ldots, t_n$. We use φ or Ψ to denote a set (conjunction) of linear constraints and $\varphi_1 \models \varphi_2$ to indicate that φ_1 implies φ_2. A mapping from a set of variables to integers is denoted by σ.

2.1 Cost Relations: The Common Target of Cost Analyzers

Let us now recall the general notion of *cost relation* (*CR*) as defined in [2] which generalizes the *CRs* yield by most analyzers. The basic building blocks of *CRs* are the so-called *cost expressions* e which are generated using this grammar:

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

where $r \in \mathbb{R}^+$, $n \in \mathbb{N}^+$, l is a linear expression, S is a non empty set of cost expressions and $\mathsf{nat} : \mathbb{Z} \to \mathbb{N}$ is defined as $\mathsf{nat}(v) = \max(\{v, 0\})$. Importantly, linear expressions are always wrapped by nat in order to avoid negative evaluations. For instance, as we will see later, an UB for $C(k, j, n)$ is $\mathsf{nat}(n + j - k)$. Without the use of nat, the evaluation of $C(5, 5, 11)$ results in the negative cost -1 which must be evaluated to zero, since they correspond to executions in which the `for` loop is not entered (i.e., $k \geq n + j$).

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

The evaluation of a *CR* C for a given valuation $\bar{v}$, denoted $C(\bar{v})$, is like a constraint logic program [11] and consists of the next steps: (1) first a matching equation of the form $\langle C(\bar{x}) = e + \sum_{i=1}^{k} D_i(\bar{y}_i) + \sum_{j=1}^{n} C(\bar{z}_j), \varphi \rangle$ is chosen; (2) then, we need to choose an assignment σ s.t. $\sigma \models \bar{v} = \bar{x} \wedge \varphi$; (3) then, evaluate e w.r.t. σ and accumulate it to the result; and (4) evaluate each call $D_i(\bar{v}_i)$ where $\bar{v}_i = \sigma(\bar{y}_i)$ and $C(\bar{v}_j)$ where $\bar{v}_j = \sigma(\bar{z}_j)$. The result (i.e., the cost of the execution) of the evaluation is the sum of all cost expressions accumulated in step (3). Even if the original program is deterministic, due to the abstractions performed during the generation of the *CR*, it might happen that several results can be obtained for a given $C(\bar{v})$. Correctness of the underlying analysis used to obtain the *CR* must ensure that the actual cost is one of such solutions (see [2]). This makes it possible to use *CR* to infer both UBs and LBs from them.

Example 1. Let us evaluate $B(0, 3, 3)$. The only matching equation is the second one for B. In step (2), we choose an assignment σ. Here we have a nondeterministic choice for selecting the value of j' which can be 1, 2 or 3. In step (4), we evaluate the cost of $C(0, 0, 3)$. Finally, one of the recursive calls of $B(1, 3, 3)$, $B(2, 3, 3)$ or $B(3, 3, 3)$ will be made, depending on the chosen value for j'. If we continue executing all possible derivations until reaching the base cases, the final result for $B(0, 3, 3)$ is any of $\{9, 10, 13, 14, 15, 18\}$. The actual cost is guaranteed to be one of such values.

W.l.o.g., we formalize our method by making two simplifications: (1) *Direct recursion:* we assume that all recursions are *direct* (i.e., cycles in the call graph are of length one). Direct recursion can be automatically achieved by applying partial evaluation as described in [2]. (2) *Standalone cost relations:* we assume that *CRs* do not depend on any other *CR*, i.e., the equations do not contain external calls and thus have this simplified form $\langle C(\bar{x}) = e + \sum_{j=1}^{n} C(\bar{z}_j), \varphi \rangle$. This

can be assumed because our approach is compositional. We start by computing bounds for the *CRs* which do not depend on any other *CRs*, e.g., C in Fig. 1b is solved by providing the UB $\mathsf{nat}(n + j - k)$. Then, we continue by replacing the computed bounds on the equations which call such relation, which in turn become standalone. For instance, replacing the above solution in the relation B results in the equation $B(j, i, n) = \mathsf{nat}(n + j) + B(j', i, n), \{j < i, j + 1 \leq j' \leq j + 3\}$. This operation is repeated until no more *CR* need to be solved. In what follows, *CR* refers to standalone *CRs* in direct recursive form.

2.2 Single-Argument Recurrence Relations

It is fundamental for this paper to understand the differences between *CRs* and *RRs*. The following features have been identified in [2] as main differences, which in turn justify the need to develop specific solvers to bound *CRs*:

1. *CRs* often have *multiple arguments* that increase or decrease over the relation (e.g., in A variable i' increases). The number of evaluation steps (i.e., recursive calls performed) is often a function of such several arguments.
2. *CRs* often contain *inexact size relations*, e.g., variables range over an interval $[a, b]$ (e.g., variable j' in B). Thus, given a valuation, we might have several solutions which perform a different number of evaluation steps.
3. Even if the original programs are deterministic, due to the loss of precision in the first stage of the static analysis, *CRs* often involve several *non-deterministic equations*. This will be further explained in Sec. 4.3.

As a consequence of 2 and 3, an exact solution often does not exist and hence *CAS* just cannot be used in such cases. But, even if a solution exists, due to such three additional features, *CAS* do not accept *CRs* as a valid input. Below, we define a class of recurrence equations that *CAS* can handle.

Definition 2 (single-argument *RR*). *A* single-argument *recurrence relation C is defined by at most one recursive equation $\langle C(x) = E + \sum_{i=1}^{n} C(x - 1)\rangle$ where E is a function on x (and might have constant symbols), and a base case $\langle C(0) = \kappa\rangle$ where κ is a symbol representing the value of the base case.*

Depending on the number of recursive calls in the recursive equation and the expression E, such solution can be of different complexity classes (exponential, polynomial, etc.). A closed-form solution for $C(x)$, if exists, is an arithmetic expression that depends only on the variable x, the base-case symbol κ, and might include constant symbols that appear in E. W.l.o.g., in what follows, we assume that $\kappa = 0$. In the implementation, we replace κ in the closed-form UB (resp. LB) by the maximum (resp. minimum) value that it can take, as done in [2].

3 An Informal Account of Our Approach

This section informally explains the approximation we want to achieve and compares it to the actual cost and the approximation of [2]. Consider a *CR* in its

simplest form with a base case $\langle C(\bar{x})=0, \varphi_0\rangle$ and a recursive case with a single recursive call $\langle C(\bar{x})=e+C(\bar{x}'), \varphi_1\rangle$. The challenge is to accurately estimate the cost of $C(\bar{x})$ for any input. *CAS* aim at obtaining the exact cost function. As we have discussed in Sec. 2.2, this is often not possible since even a single evaluation has multiple solutions. Thus, the goal of static cost analysis is to infer closed-form UBs/LBs for C. Our starting point is the general approximation for UBs proposed by [2] which has two dimensions. (1) *Number of applications of the recursive case:* The first dimension is to infer an UB on the number of times the recursive equations can be applied (which, for loops, corresponds to the number of iterations). This is done by inferring an UB $\hat{n}$ on the length of chains of recursive calls; (2) *Cost of applications:* The second dimension is to infer an UB $\hat{e}$ for all e_i. Then, for a relation with a single recursive call, $\hat{n} * \hat{e}$ is guaranteed to be an UB for C. If the relation C had two recursive calls, the solution would be an exponential function of the form $2^{\hat{n}} * \hat{e}$. Programming-languages techniques of wide applicability have been proposed by [2] in order to solve the two dimensions, as detailed below.

Ranking functions. A ranking function is a function f such that for any recursive equation $\langle C(\bar{x})=e+C(\bar{x}_1)+\cdots+C(\bar{x}_k), \varphi\rangle$ in the *CR*, it holds that $\forall 1\leq i\leq k.\varphi \models f(\bar{x})>f(\bar{x}_i)\wedge f(\bar{x})>0$. This guarantees that when evaluating $C(\bar{v})$, the length of any chain of recursive calls to C cannot exceed $f(\bar{v})$. Thus, f is used to bound the length of these chains [2, 5, 6]. We rely on [2] for automatically inferring a ranking function $\hat{f}_C(\bar{x}_0)$ for C (variables $\bar{x}_0$ denote the initial values).

Maximization. In [2] the second dimension is solved by first inferring an *invariant* $\langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \Psi\rangle$, where Ψ is a set of linear constraints, which describes the relation between the values that $\bar{x}$ can take in any recursive call and the initial values $\bar{x}_0$. Then in order to generate $\hat{e}$ each $\mathsf{nat}(l) \in e$ is replaced by $\mathsf{nat}(\hat{l})$ where $\hat{l}$ is a linear expression (over $\bar{x}_0$) which is an UB for any valuation of l. We rely on the techniques of [2] in order to automatically obtain $\mathsf{nat}(\hat{l})$ for $\mathsf{nat}(l)$.

Our challenge is to improve precision of [2] while keeping a similar applicability for UBs and, besides, be able to apply our approach to infer useful LBs. The fundamental idea is to generate a sequence of (different) elements $u_1, \ldots, u_{\hat{n}}$ such that for any concrete evaluation $e_1, \ldots, e_n$ it holds $\forall 0 \leq i \leq n-1$. $u_{\hat{n}-i}\geq e_{n-i}$. Note that it is ensured that the last n elements of the u sequence are larger than (or equal to) the n elements of the e sequence, but it is not guaranteed that $u_i\geq e_i$. This guarantees that $u_1+\cdots+u_{\hat{n}}$ is an UB for $e_1+\cdots+e_n$. Our UB is potentially more precise than $\hat{n}*\hat{e}$, since each e_i is approximated more tightly by a corresponding u_j. Technically, we do this by transforming the *CR* into a (worst-case) *RR* (as in Def. 2) whose closed-form solution is $u_1 + \cdots + u_{\hat{n}}$. The novel idea is to view $u_1, \cdots, u_{\hat{n}}$ as an arithmetic sequence that starts from $u_{\hat{n}}\equiv\hat{e}$ and each time decreases by $\check{d}$ where $\check{d}$ is an under approximation of all $d_i=e_{i+1}-e_i$, i.e., $u_i=u_{i-1}+\check{d}$. In our approach the problem of inferring LBs is dual, namely we can infer a LB $\check{n}$ on the length of chains of recursive calls, the minimum value $\check{e}$ to which e_i can be evaluated, and then sum the sequence $\ell_1, \ldots, \ell_{\check{n}}$ where $\ell_i=\ell_{i-1}+\check{d}$ and $\ell_1=\check{e}$.

4 Inference of Precise Upper Bounds

In this section, we present our approach to accurately infer UBs on the resource consumption in three steps: first in Sec. 4.1, we handle a subclass of *CRs* which accumulate a constant cost, then we handle *CRs* which accumulate non-constant costs in Sec. 4.2 and *CRs* with multiple overlapping equations in Sec. 4.3.

4.1 Constant Cost Relations

We consider *CRs* defined by a single recursive equation with constant cost:

$$\langle C(\bar{x}) = 0, \varphi_1 \rangle \mid \langle C(\bar{x}) = e + C(\bar{x}_1) + \cdots + C(\bar{x}_k), \varphi_2 \rangle \quad (1)$$

where e contributes a *constant cost*, i.e., it is a constant number or an expression that always evaluates to the same value. As explained in Sec. 3, any chain of recursive calls in C is at most of length $\hat{f}_C(\bar{x}_0)$ (when starting from $C(\bar{x}_0)$). We aim at obtaining an UB for C by solving a *RR* P_C in which all chains of recursive calls are of length $\hat{f}_C(\bar{x}_0)$. Intuitively, $P_C(x)$ can be seen as a special case of a *RR* with the same number of recursive calls as in C, where all chains of recursive calls are of length x, and each application accumulates the constant cost e. Its solution can be then instantiated for the case of C by replacing x by $\hat{f}_C(\bar{x}_0)$.

Definition 3. *The worst-case RR of C is* $\langle P_C(x) = e + P_C(x-1) + \cdots + P_C(x-1) \rangle$.

The main achievement of the above transformation is that, for constant *CRs*, we get rid of their problematic features described in Sec. 2.2 which prevented us from relying on *CAS* to obtain a precise solution. The following theorem explains how the closed-form solution of the *RR* P_C can be transformed into an UB for the *CR* C.

Theorem 1. *If E is a solution for $P_C(x)$ then $E[x/\hat{f}_C(\bar{x}_0)]$ is an UB for $C(\bar{x}_0)$.*

Example 2. The worst-case *RR* of the *CR* C of Fig. 1b is $\langle P_C(x) = 1 + P_C(x-1) \rangle$, which is solved using *CAS* to $P_C(x) = x$ for any $x \geq 0$. The UB for C is obtained by replacing x by $\hat{f}_C(k_0, j_0, n_0) = \mathsf{nat}(j_0 + n_0 - k_0)$.

4.2 Non-constant Cost Relations

During cost analysis, in many cases we obtain *CRs* like the one of Eq. 1, but with a non-constant expression e which is evaluated to different values e_i in different applications of the recursive equation. The transformation in Def. 3 would not be correct since in these cases e must be appropriately related to x. In particular, the main difficulty is to simulate the accumulation of the non-constant expressions e_i at the level of the *RR*. As we have illustrated in Sec. 3, the novel idea is to simulate this behavior with an arithmetic sequence that starts from the maximum value that e can take, and in each step decreases by the minimum distance $\check{d}$ between two consecutive expressions e_i and e_{i+1}. Since the expression e might have a complex form (e.g., exponential, polynomial, etc),

inferring a precise LB on the distance $\check{d}$ is usually impractical. A key observation in our approach is that, since variables are wrapped by nat, it is enough to reason on the behavior of its nat sub-expressions, i.e., we only need to understand how each $\mathsf{nat}(l)$ of e (denoted $\mathsf{nat}(l) \in e$) changes along a sequence of recursive calls.

Definition 4 (nat with linear behaviour). *Consider the CR C of Eq. 1 with e a (possibly) non-constant expression. We say that a given $\mathsf{nat}(l) \in e$ is linearly increasing (resp. decreasing) if there exists a non-negative integer $\check{d}$, such that for a given renamed apart instance of the recursive equation $\langle C(\bar{y}) = e' + C(\bar{y}_1) + \cdots + C(\bar{y}_k),\ \varphi'_2\rangle$, it holds that $\varphi_2 \wedge \varphi'_2 \wedge \bar{x}_i = \bar{y} \models l' - l \geq \check{d}$ (resp. $\varphi_2 \wedge \varphi'_2 \wedge \bar{x}_i = \bar{y} \models l - l' \geq \check{d}$) for any $\bar{x}_i$, where $\mathsf{nat}(l') \in e'$ is the renaming of $\mathsf{nat}(l)$.*

In practice, computing $\check{d}$ for a given $\mathsf{nat}(l) \in e$ can be done using integer programming tools. In what follows, when the conditions of Def. 4 hold for a given $\mathsf{nat}(l) \in e$, we say that it has a linear behavior. Moreover, when all $\mathsf{nat}(l) \in e$ have the same linear behavior (i.e., all increasing or all decreasing), we say that e has a linear behavior.

Example 3. For B, replacing $C(0, j, n)$ by the UB $\mathsf{nat}(n{+}j)$ computed in Ex. 2 results in $\langle B(j, i, n){=}\mathsf{nat}(n{+}j){+}B(j', i, n), \varphi_1\rangle$, where $\varphi_1 = \{j{<}i, j{+}1{\leq}j'{\leq}j{+}3\}$. Its renamed apart instance is $\langle B(j_r, i_r, n_r) = \mathsf{nat}(n_r{+}j_r) + B(j'_r, i_r, n_r), \varphi_2\rangle$ where $\varphi_2{=}\{j_r{<}i_r, j_r{+}1{\leq}j'_r{\leq}j_r{+}3\}$. Then, the formula $\varphi_1{\wedge}\varphi_2{\wedge}\{j'{=}j_r, i{=}i_r, n = n_r\} \models (n_r{+}j_r){-}(n{+}j) \geq \check{d}$ holds for $\check{d}{=}1$. Therefore, $\mathsf{nat}(n{+}j)$ increases linearly.

Let us intuitively explain how our method works by focusing on a single $\mathsf{nat}(l) \in e$ within the relation C. Assume that during the evaluation of an initial query $C(\bar{x}_0)$, $\mathsf{nat}(l)$ is evaluated to $\mathsf{nat}(l_1), \ldots, \mathsf{nat}(l_n)$ in n consecutive recursive calls, and suppose that it is linearly increasing at least by $\check{d}$, i.e., $l_{i+1} - l_i \geq \check{d}$ for all $1 \leq i \leq n-1$. As explained in Sec. 3, we can infer an expression $\mathsf{nat}(\hat{l})$ which is an UB for all $\mathsf{nat}(l_i)$, and a ranking function $\hat{f}_C$ such that $n \leq \hat{f}_C(\bar{x}_0)$. A tight approximation is the arithmetic sequence which starts from $\mathsf{nat}(\hat{l})$ and each time decreases by $\check{d}$. Clearly, the first element of this sequence is greater than $\mathsf{nat}(l_n)$, the second is greater than $\mathsf{nat}(l_{n-1})$, and so on.

However, a main problem is that, since $\hat{f}_C$ provides an over-approximation of the actual number of iterations, the sequence might go to negative values. This is because an imprecise (too large) $\hat{f}_C$ would lead to a too large decrease $\check{d} * \hat{f}_C(\bar{x}_0)$ and the smallest element $\mathsf{nat}(\hat{l}) - \check{d} * \hat{f}_C(\bar{x}_0)$ (and possibly other subsequent ones) could be negative. Hence, the approximation would be unsound since the actual evaluations of such negative values are zero. We avoid this problem by viewing this sequence in a dual way: we start from the smallest value and in each step increase it by $\check{d}$. Since still the smallest values could be negative, we start from $\mathsf{nat}(\hat{l} - \check{d} * \hat{f}_C(\bar{x}_0))$ which is guaranteed to be positive and greater than or equal to $\mathsf{nat}(\hat{l}) - \check{d} * \hat{f}_C(\bar{x}_0)$. The next definition uses this intuition to replace each nat by an expression that generates its corresponding sequence at the level of RR.

Definition 5. *Consider the CR C of Eq. 1 where e has a linear behavior. Let $\hat{f}_C(\bar{x}_0) = \mathsf{nat}(l')$ be its corresponding ranking function. We define its associated*

worst-case RR as $\langle P_C(x) = E_e + P_C(x-1) + \cdots + P_C(x-1)\rangle$ *where* E_e *is obtained from* e *by replacing each* $\mathsf{nat}(l) \in e$ *by* $\mathsf{nat}(\hat{l} - \check{d} * l') + x * \check{d}$.

Definition 5 generalizes Def. 3 and an equivalent theorem to Theorem 1 holds.

Theorem 2. *If* E *is a solution for* $P_C(x)$ *then* $E[x/\hat{f}_C(\bar{x}_0)]$ *is an UB for* $C(\bar{x}_0)$.

Example 4. Following Ex. 3, we have that $\check{d}$=1. Since $\mathsf{nat}(n_0{+}i_0{-}1)$ is an UB of the cost $\mathsf{nat}(n{+}j)$ accumulated in B, and $\hat{f}_B(j_0, i_0, n_0){=}\mathsf{nat}(i_0{-}j_0)$, according to Def. 5, we have $\langle P_B(x){=}\mathsf{nat}(n_0{+}i_0{-}1{-}(i_0-j_0)*1){+}x*1{+}P_B(x-1)\rangle$ which is solved by *CAS* to $P_B(x){=}\mathsf{nat}(n_0+j_0-1)*x{+}x*(x{+}1)/2$. Thus, $B(j_0, i_0, n_0) = P_B(x)[x/\mathsf{nat}(i_0-j_0)]$. Similarly, for A we obtain the *RR* $P_A(x) = (q{+}2x)(q/2{+}x){+}r(q{+}2x){+}q/2{+}x{+}P_A(x{-}1)$ where $q = \mathsf{nat}(i_0{-}2)$ and $r = \mathsf{nat}(n_0{-}1)$, which is solved to $P_A(x) = qx^2{+}qrx{+}rx{+}2/3x^3{+}rx^2{+}3/2x^2{+}5/6x{+}1/2q^2x{+}3/2qx$. Thus, $A(i_0, n_0) = P_A(x)[x/\mathsf{nat}((n_0 - i_0)/2)]$. Finally, for F, we obtain the UB $F(n_0) = y(4y^2{+}6zy{+}9y{+}6z{+}5)/6$, whereas [2] provides $2{*}\mathsf{nat}(n_0/2{+}1/2){*}z^2$, where $y = \mathsf{nat}(n_0/2)$ and $z = \mathsf{nat}(n_0 - 1)$, which is much less precise.

Our approach can be also applied when nat expressions are increasing or decreasing geometrically, i.e., when $\mathsf{nat}(l_{i+1}) \leq k * \mathsf{nat}(l_i)$ for some positive rational k called common ratio. This is the case in a *CR* like $\langle C(n){=}\mathsf{nat}(n){+}C(n/2), \{n \geq 1\}\rangle$, which is similar to what we obtain when analyzing the recursive implementation of merge-sort algorithm (mergesort has two recursive calls). In such geometric case, the counterpart condition to Def. 4 checks if there exists a minimum ratio $\check{k}$ such that $\varphi_2 \wedge \varphi_2' \wedge \bar{x}_i = \bar{y} \models l \geq \check{k} * l'$. Then, in a counterpart definition to Def. 5, we replace such $\mathsf{nat}(l) \in e$ by $\mathsf{nat}(\hat{l}) * \check{k}^{m-x} \in E_e$ where $m = \hat{f}_C(\bar{x}_0)$. Intuitively, we accumulate $\mathsf{nat}(\hat{l})$ when $x = \hat{f}_C(\bar{x}_0)$, and, at each subsequent step, the expression is geometrically reduced by the ratio. For the above *CR*, we obtain a linear UB $C(n_0) = 2 * \mathsf{nat}(n_0)$, whereas techniques described in [2, 9] would obtain $C(n_0) = \mathsf{nat}(n_0) * log_2(\mathsf{nat}(n_0 + 1))$. Note that here our approach improves even the complexity order. Using a similar construction, for merge-sort (see experiments), we are able to infer the upper bound $63\mathsf{nat}(a+1)log_2(\mathsf{nat}(2a-1)+1) + 50\mathsf{nat}(2a-1)$ on the number of executed instructions. For conciseness, rest of the paper formalizes the arithmetic case, but all results are directly applicable to geometric progressions as described above.

4.3 Non-deterministic Non-constant Cost Relations

Any approach for solving *CRs* that aims at being practical has to consider *CRs* with several recursive equations as shown in equation 2. This kind of *CRs* is very common during cost analysis and they mainly originate from conditional statements inside loops.

$$\begin{array}{l} \langle C(\bar{x}) = e_0, \varphi_0\rangle \\ \langle C(\bar{x}) = e_1 + C(\bar{x}_1) + \cdots + C(\bar{x}_{k_1}), \varphi_1\rangle \\ \qquad\qquad \vdots \\ \langle C(\bar{x}) = e_h + C(\bar{x}_1) + \cdots + C(\bar{x}_{k_h}), \varphi_h\rangle \end{array} \tag{2}$$

For instance, the instruction `if (x[i]>0) {A;} else {B;}`, may lead to two

non-deterministic equations which accumulate the costs of `A` and `B`. This is because arrays are typically abstracted to their length and, hence, the guard `x[i]>0` is abstracted to `true`, i.e., we do not keep this information on the *CR*. Thus, $\varphi_0, \ldots, \varphi_h$ are not necessarily mutually exclusive. W.l.o.g., we assume that $k_1 \geq \cdots \geq k_h$, i.e., the first (resp. last) recursive equation has the maximum (resp. minimum) number of recursive calls among all equations. We also assume that $\hat{f}_C(\bar{x}_0) = \mathsf{nat}(l')$ is a *global* ranking function for this *CR*, i.e., a ranking function for all equations.

In non-deterministic *CRs*, the costs contributed by a chain of recursive calls might not be instances of the same cost expression, but rather of different expressions $e_1, \ldots, e_h$, i.e., the equations might interleave. Namely, we might apply one equation and for another call another different equation. The worst-case cost might originate from such interleaving sequences (see [2]). Thus, when inferring how a given $\mathsf{nat}(l) \in e_i$ changes, we have to consider subsequent instances of $\mathsf{nat}(l)$ which are not necessarily consecutive. For this, we infer an invariant that holds between two subsequent (not necessarily consecutive) applications of the same equation, similar to what [2] does, and then we compute the distance $\check{d}$ between its subsequent instances as in Def. 4 but considering this invariant.

As a first solution, similarly to Def. 5, for each expression e_i, we can generate a corresponding E_i by replacing each $\mathsf{nat}(l)$ by $\mathsf{nat}(\hat{l}-\check{d}*l')+x*\check{d}$ where $\check{d}$ is the distance for $\mathsf{nat}(l)$. Clearly, if e is a closed-form solution for the *RR* $P_C(x)=\max(E_1,\ldots,E_h)+P_C(x-1)+\ldots+P_C(x-1)$ with k_1 recursive calls, then $e[x/\hat{f}_C(\bar{x}_0)]$ is an UB for C (because in each application we take the worst-case). Unfortunately, *CAS* fail to solve *RRs* which involve (non-constant) max expressions. Therefore, this approach is not practical. Clearly, in the case that one of $e_1, \ldots, e_h$ is provable to be the maximum, this approach works since we can eliminate the max operator. Unfortunately, even comparing simple cost expressions is difficult and in many cases not feasible [1]. In what follows, we describe a practical solution to this problem, which is based on finding an expression E which does not include max and is always larger than or equal to $\max(E_1, \ldots, E_h)$. This way, we can replace the max by E and still get an UB for C.

First, observe that any cost expression (which does not include max) can be normalized to the form $\Sigma_{i=1}^{n} \Pi_{j=1}^{m_i} b_{ij}$ (i.e., sum of multiplications) where each b_{ij} is a *basic element* of the following form $\{r, \mathsf{nat}(l), n^{\mathsf{nat}(l)}, \log(\mathsf{nat}(l))\}$. We assume that all $e_1, \ldots, e_h$ of Eq. 2 are given in this form. For simplicity, we assume that all expressions have the same number of multiplicands, and all multiplicands have the same number of basic expressions (if not, we just add 1 in multiplication and 0 for sum). Now since e_i is in a normal form, the corresponding E_i will be also in a corresponding normal form. Given two cost expressions e_1 and e_2, and their corresponding E_1 and E_2, the following definition describes how to generate an expression E such that it is larger than (or equal to) both E_1 and E_2.

Definition 6. *Given two expressions* $E_i = a_{11} \cdots a_{1m_1} + \cdots + a_{n1} \cdots a_{1m_n}$ *and* $E_j = b_{11} \cdots b_{1m_1} + \cdots + b_{n1} \cdots b_{1m_n}$. *We define the generalization of* E_i *and* E_j *as* $E_i \sqcup E_j = c_{11} \cdots c_{1m_1} + \cdots + c_{n1} \cdots c_{1m_n}$ *where* $c_{ij} = b_{ij}$ *if we can prove that* $b_{ij} \geq a_{ij}$, $c_{ij} = a_{ij}$ *if we can prove that* $a_{ij} \geq b_{ij}$, *otherwise* $c_{ij} = a_{ij} + b_{ij}$.

Although we need to compare expressions when constructing $E_i \sqcup E_j$ (namely b_{ij} to a_{ij}), this comparison is on the basic elements rather than on the whole expression and hence it is far simpler. By construction, we guarantee that $E_i \sqcup E_j$ is always greater than or equal to both E_i and E_j. Clearly, the quality of $E_i \sqcup E_j$ (i.e., how tight it is) depends on the ordering of the summands and the elements of each summand (i.e., multiplicands) in both E_i and E_j. In order to obtain tighter bounds, we use some heuristics like ordering the elements inside each multiplication in increasing complexity in such a way that we always try to compare basic elements of the same complexity order. Besides, we try to compare basic elements that involve the same variable.

Definition 7. *Let C be the CR of Eq. 2, $\hat{f}_C(\bar{x}_0)$=$\mathsf{nat}(l')$ its corresponding ranking function, and E_i generated from e_i by replacing each $\mathsf{nat}(l)$∈e_i by $\mathsf{nat}(\hat{l}-d*l')+x*\check{d}$ where $\check{d}$ is the distance of $\mathsf{nat}(l)$. The corresponding worst-case RR is $\langle P_C(x) = E_1 \sqcup \cdots \sqcup E_h + P_C(x-1) + \cdots + P_C(x-1)\rangle$ with k_1 recursive calls.*

Theorem 3. *If E is a solution for $P_C(x)$ then $E[x/\hat{f}_C(\bar{x}_0)]$ is an UB for $C(\bar{x}_0)$.*

Example 5. Let us add the contrived recursive equation $B(j,i,n) = \mathsf{nat}(n+15) + B(j',i,n) + B(j'',i,n)$ $\{j<i, j'=j+1, j''=j+2\}$ to the *CR* B. It has two recursive calls and a non-deterministic choice for accumulating either $e_1 = \mathsf{nat}(n+j)$ or $e_2 = \mathsf{nat}(n+15)$. The function $f_B(j_0,i_0,n_0) = \mathsf{nat}(i_0 - j_0)$ is a ranking function for all equations. Next, we compute $E_1 \sqcup E_2$ where $E_1 = \mathsf{nat}(n_0 + i_0 - 1 - (i_0 - j_0)) + x$ and $E_2 = \mathsf{nat}(n_0 + 15 - (i_0 - j_0)) + x$. A naive generalization results in $\mathsf{nat}(n_0+i_0-1-(i_0-j_0)) + \mathsf{nat}(n_0+15-(i_0-j_0)) + x$, but syntactically analyzing the expressions and employing the above heuristics, we automatically obtain a tighter bound $\mathsf{nat}(n_0+j_0+15)+x$. Now we generate $\langle P_B(x) = \mathsf{nat}(n_0+j_0+15) + x + P_B(x-1) + P_B(x-1)\rangle$ which can be solved to $\langle P_B(x) = 2^x(q+2) - q - x - 2\rangle$ for $q = \mathsf{nat}(n_0 + j_0 + 15)$ and therefore $B(j_0,i_0,n_0) = P_B(x)[x/\mathsf{nat}(i_0-j_0)]$.

5 The Dual Problem: Lower Bounds

We now aim at applying the approach from Sec. 4 in order to infer *lower bounds*, i.e., under-approximations of the best-case cost. Such LBs are typically useful in granularity analysis to decide if tasks should be executed in parallel. This is because the parallel execution of a task incurs various overheads, and therefore the LB cost of the task can be useful to decide if it is worth executing it concurrently as a separate task. Due in part to the difficulty of inferring under-approximations, a general framework for inferring LBs from *CR* does not exist. When trying to adapt the UB framework of [2] to LB, we only obtain trivial bounds. This is because the minimization of the cost expression accumulated along the execution is in most cases zero and, hence, by assuming it for all executions we would obtain a trivial (zero) LB. In our framework, even if the minimal cost could be zero, since we do not assume it for all iterations, but rather only for the first one, the resulting LB is non-trivial.

Existing approaches typically assume that the length of chains of recursive calls depends on a single decreasing argument. We first propose a new technique to inferring LBs on the length of such chains, which does not have this restriction. Essentially, we add a counter to the equations in the *CR* and infer an invariant which involves this counter. The invariant is indeed the same one used later to obtain $\check{l}$. The minimum value of this counter when we enter a non-recursive case is a LB on the length of those chains.

Definition 8. *Given the CR of Eq. 2, we compute* $\check{f}_C(\bar{x}_0) = \mathsf{nat}(l)$ *which is a* lower bound on the length *of any chain of recursive calls when starting from* $C(\bar{x}_0)$ *in three steps: (1) Replace each head* $C(\bar{x})$ *by* $C(\bar{x}, lb)$ *and each recursive call* $C(\bar{x}_j)$ *by* $C(\bar{x}_j, lb{+}1)$*; (2) Infer an invariant* $\langle C(\bar{x}_0, 0) \rightsquigarrow C(\bar{x}, lb), \Psi\rangle$ *for the new CR; (3) Syntactically look for* $lb \geq l$ *in* $\Psi \wedge \varphi_0$ *(projected on* $\bar{x}_0$ *and* lb*).*

Example 6. Applying step (1) on the *CR* B results in $\langle B(j,i,n,lb) = 0, \{j \leq i\}\rangle$ and $\langle B(j,i,n,lb) = \mathsf{nat}(n{+}j){+}B(j',i,n,lb{+}1), \{j{<}i, j{+}1{\leq}j'{\leq}j{+}3\}\rangle$. The invariant Ψ for this *CR* is $\{j{-}j_0{-}lb{\geq}0, j_0{+}3lb{-}j{\geq}0, i{=}i_0, n{=}n_0\}$. Projecting $\Psi{\wedge}\{j{\geq}i\}$ on $\langle j_0, i_0, n_0, lb\rangle$ results in $\{lb{\geq}0, j_0{+}3lb{-}i_0{\geq}0\}$ which implies $lb \geq (i_0 - j_0)/3$. Similarly $\check{f}_C(k_0, j_0, n_0){=}\mathsf{nat}(n_0{+}j_0 - k_0)$ and $A(i_0, n_0){=}\mathsf{nat}(\frac{n_0 - i_0}{4})$.

We present the approach directly for the non-deterministic *CR* of Eq. 2. As in Def. 6, we can reduce the expressions $E_1, \ldots, E_h$ in order to get an expression which is guaranteed to be smaller than or equal to $\min(E_1, \ldots, E_h)$.

Definition 9. *Given the expressions* E_i *and* E_j *in Def. 6, we define their* reduction *as* $E_i \sqcap E_j = c_{11} \cdots c_{1m_1} + \cdots + c_{n1} \cdots c_{1m_n}$ *where* $c_{ij} = b_{ij}$ *if we can prove that* $b_{ij} \leq a_{ij}$*,* $c_{ij} = a_{ij}$ *if we can prove that* $a_{ij} \leq b_{ij}$*, otherwise* $c_{ij} = 0$*.*

The case of $c_{ij} = 0$ can be improved to obtain a tighter LB by relying on heuristics, similarly to what we have discussed in Sec. 4.3. As intuitively explained in Sec. 3, the main idea is to simulate each $\mathsf{nat}(l)$ by a sequence that starts from $\mathsf{nat}(\check{l})$ and increases in each iteration by the minimal distance $\check{d}$.

Definition 10. *Let* C *be the CR of Eq. 2 such that for each* $\mathsf{nat}(l) \in e_i$ *it holds that* $\check{l} \geq 0$*, and let* E_i *be the expression generated from* e_i *by replacing each* $\mathsf{nat}(l)$ *by* $\mathsf{nat}(\check{l}) + (x-1) * \check{d}$*. The corresponding best-case RR is* $\langle P_C(x) = E_1 \sqcap \cdots \sqcap E_h + P_C(x-1) + \cdots + P_C(x-1)\rangle$ *with* k_h *recursive calls.*

In the above definition, it can be observed that, for the sake of soundness, we require that for each $\mathsf{nat}(l)$ it holds that $\check{l} \geq 0$. Intuitively, when such expressions take negative values, by definition of nat, they evaluate to zero and there can be a sequence of zeros until the evaluation becomes positive. Our under-approximation would be unsound in this case, because it assumes as minimum value zero and then starts to increase it by the minimum distance. Thus, for some values, the approximation could be actually bigger than the actual value.

Theorem 4. *If* E *is a solution for* $P_C(x)$*, then* $E[x/\check{f}_C(\bar{x}_0)]$ *is a LB for* $C(\bar{x}_0)$*.*

#	UBs and LBs	T
1	$24\eta(a-1)^3+36\eta(a-1)^2+27\eta(a)^2+39\eta(a)\eta(a-1)+35\eta(a-1)+72\eta(a)+54$	1240
	$8\eta(a-1)^3+27\eta(a)^2+\frac{99}{2}\eta(a-1)^2+\frac{231}{2}\eta(a-1)+72\eta(a)+54$	1395
	$8\eta(a-2)^3+46\eta(a-2)^2+105\eta(a-2)+55\eta(a-1)+54$	204
2	$24\eta(c-1)^3+36\eta(c-1)^2+28\eta(c)^2+\eta(c-1)(40\eta(c)+35)+25\eta(c)+48\eta(b-1)^2+46\eta(b-1)+74$	1270
	$8\eta(c-1)^3+28\eta(c)^2+50\eta(c-1)^2+25\eta(c)+117\eta(c-1)+24\eta(b-1)^2+70\eta(b-1)+74$	1425
	$8\eta(c-2)^3+48\eta(c-2)^2+25\eta(c-1)+111\eta(c-2)+24\eta(b-2)^2+70\eta(b-2)+74$	247
3	$24\eta(a-1)^3+56\eta(a)\eta(a-1)^2+27\eta(a)^2+46\eta(a-1)^2+75\eta(a)+77\eta(a)\eta(a-1)+49\eta(a-1)+62$	3617
	$8\eta(a-1)^3+28\eta(a)\eta(a-1)^2+27\eta(a)^2+\frac{109}{2}\eta(a-1)^2+75\eta(a)+66\eta(a)\eta(a-1)+\frac{269}{2}\eta(a-1)+62$	3890
	$18\eta(a-2)^3+81\eta(a-2)^2+75\eta(a-1)+144\eta(a-2)+62$	415
4	$25\eta(b)\eta(c)\eta(c-1)+30\eta(b)\eta(c)+16\eta(b)+6$	130
	$25/2\eta(b)\eta(c-b)^2+25\eta(b)^2\eta(c-b)+25/2\eta(b)^3+40\eta(b)^2+135/2\eta(b)\eta(c-b)+87/2\eta(b)+6$	200
	$21/2\eta(b-1)^2+21\eta(b-1)\eta(c-b)+53/2\eta(b-1)+6$	60

#	UBs and LBs	T
5	$19\eta(a-1)^2+25\eta(a-1)+7$	44
	$19/2\eta(a-1)^2+69/2\eta(a-1)+7$	63
	$18\eta(a-2)+7$	10
7	$27\eta(a-1)^2+16\eta(a-1)+9$	103
	$27/2\eta(a-1)^2+59/2\eta(a-1)+9$	120
	$13/2\eta(a-2)^2+45/2\eta(a-2)+9$	25
9	$34\eta(a)\eta(a-1)+12\eta(a)+8$	174
	$17\eta(a)^2+29\eta(a)+8$	197
	$8\eta(a-1)^2+20\eta(a-1)+8$	24

#	UBs and LBs	T
6	$43\eta(a)\eta(2a-3)+53\eta(2a-3)+17$	2127
	$63\eta(a+1)log_2(\eta(2a-1)+1)+50\eta(2a-1)$	2100
	0	40
8	$16\eta(a)^2+27\eta(a-1)^2+31\eta(a)+10\eta(a-1)+25$	200
	$27/2\eta(a)^2+27\eta(a-1)^2+10\eta(a-1)+67/2\eta(a)+25$	247
	$5/2\eta(a-1)^2+10\eta(a-2)+67/2\eta(a-1)+25$	60
10	$2^{\eta(a-1)}(5\eta(a-1)+21)+5\eta(a)-5\eta(a-1)-7$	104
	$31*2^{\eta(a-1)}+5\eta(a)-5\eta(a-1)-17$	144
	$31*2^{\eta(a-2)}+5\eta(a-1)-5\eta(a-2)-17$	34

Table 1: 1. DetEval(a) 2. LinEqSolve(a,b,c) 3. MatrixInv(a) 4. MatrixSort(a,b,c) 5. InsertSort(a) 6. MergeSort(a) 7. SelectSort(a) 8. PascalTriangle(a) 9. BubbleSort(a) 10. NestedRecIter(a).

Example 7. Consider the LBs on iterations of Ex. 6. Since $C(k_0, j_0, n_0)$ accumulates a constant cost 1, its LB cost is $\mathsf{nat}(n_0{+}j_0{-}k_0)$. We now replace the call $C(0, j, n)$ in B by its LB $\mathsf{nat}(n{+}j)$ and obtain the equation: $B(j, i, n) = \mathsf{nat}(n{+}j) + B(j', i, n) \quad \{j{<}i, j{+}1{\leq}j'{\leq}j{+}3\}$. Notice the need of the soundness requirement in Th. 3, i.e., $\mathsf{nat}(n{+}j){\geq}0$. E.g., when evaluating $B(-5, 5, 0)$ the first 4 instances of $\mathsf{nat}(n+j)$ are zero since they correspond to $\mathsf{nat}(-5), \ldots, \mathsf{nat}(-1)$. Therefore, it would be incorrect to start accumulating from 0 with a difference 1 at each iteration. After solving A and B in the same way, the computed final LB for $F(n)$ is: $\frac{1}{3}\mathsf{nat}(n)\mathsf{nat}(\frac{n}{4}{-}1){+}\frac{1}{18}\mathsf{nat}(\frac{n}{4}{-}1)\mathsf{nat}(\frac{n}{4}{-}1){+}\frac{1}{6}\mathsf{nat}(\frac{n}{4}{-}1)$.

6 Experiments and Conclusions

We have implemented our approach in COSTA, a COSt and Termination Analyzer for Java bytecode. The obtained *RRs* are solved using MAXIMA [12] or PURRS [4]. As benchmarks, we use classical examples from complexity analysis and numerical methods: DetEval evaluates the determinant of a matrix; LinEqSolve solves a set of linear equations; MatrixInverse computes the inverse of an input matrix; MatrixSort sorts the rows in the upper triangle of a matrix; InsertSort, SelectSort, BubbleSort, and MergeSort implement sorting algorithms; PascalTriangle computes and prints Pascal's Triangle; NestedRecIter is an interesting programming pattern we found in the Java libraries with a spacial form

of nested loops that uses recursion and a simple iteration for loop. Our implementation (and examples) can be tried out at `http://costa.ls.fi.upm.es` by enabling the option `series` in the manual configuration.

Table 1 illustrates the accuracy and efficiency on the above benchmarks using the cost model "*number of executed (bytecode) instructions*". We abbreviate $\mathsf{nat}(x)$ as $\eta(x)$. The second column shows: in the top row the UB obtained by [2], next the UB obtained by us and at the bottom our LB. Unfortunately, there are no other cost analysis tools for imperative languages available to compare experimentally to (e.g., SPEED [9]). As regards UBs, we improve the precision over [2] in all benchmarks. This improvement, in all benchmarks except MergeSort and NestedRecIter, is due to nested loops were the inner loops bounds depend on the outer loops counters. In these cases, we accurately bound the cost of each iteration of the inner loops, rather than assuming the worst-case cost. For MergeSort, we obtain a tight bound in the order of $a{*}log(a)$. Note that [2] could obtain $a{*}log(a)$ only for simple cost models that count the visits to a specific program point but not for number of instructions, while ours works with any cost model. For NestedRecIter, we improve the complexity order over [2] from $a{*}2^a$ to 2^a. As regards LBs, it can be observed from the last row of each benchmark that we have been able to prove the positive nat condition and obtain non-trivial LBs in all cases except MergeSort. For MergeSort, the lower bound on loop iterations is a logarithmic which cannot be inferred by our linear invariant generation tool and hence we get trivial bound 0. Note that for InsertSort we infer a linear LB which happens when the array is sorted. Column T shows the time (in milliseconds) to compute the bounds from the generated *CR*. Our approach is slightly slower than [2] mainly due to the overhead of connecting COSTA to the external *CAS*.

7 Conclusions

When comparing our approach (for UBs) to [9], since the underlying cost analysis framework is fundamentally different from ours, it is not possible to formally compare the resulting upper bounds in all cases. However, by looking at small examples, we can see why our approach can be more precise. For instance, in [9] the worst-case time usage $\sum_{i=1}^{n} i$ is over-approximated by n^2, while our series-based approach is able to obtain the precise solution. For such polynomial cases, the approach of [10] can compute also the exact solution. However, this approach is restricted to univariate polynomial bounds, while ours can be applied to obtain general polynomial, exponential and logarithmic bounds as well.

Finally, to conclude, we have proposed a novel approach to infer precise upper/lower bounds of *CRs* which, as our experiments show, achieves a very good balance between the accuracy of our analysis and its applicability. The main idea is to automatically transform *CRs* into a simple form of worst-case/best-case *RRs* that *CAS* can accurately solve to obtain upper/lower bounds on the resource consumption. The required transformation is far from trivial since it requires transforming non-deterministic equations involving multiple increas-

ing/decreasing arguments into deterministic equations with a single decreasing argument.

Acknowledgements. This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, by the Spanish Ministry of Science and Innovation (MICINN) under the TIN-2008-05624 *DOVES* project, the HI2008-0153 (Acción Integrada) project, the UCM-BSCH-GR58/08-910502 Research Group and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS* project.

References

1. E. Albert, P. Arenas, S. Genaim, I. Herraiz, and G. Puebla. Comparing cost functions in resource analysis. In *FOPARA'09*, volume 6234 of *LNCS*. Springer, 2010.
2. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 2010. To appear.
3. 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.
4. R. Bagnara, A. Pescetti, A. Zaccagnini, and E. Zaffanella. PURRS: Towards Computer Algebra Support for Fully Automatic Worst-Case Complexity Analysis. Technical report, 2005. `arXiv:cs/0512056` available from http://arxiv.org/.
5. Amir M. Ben-Amram. Size-change termination, monotonicity constraints and ranking functions. In *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 109–123. Springer, 2009.
6. P. Feautrier C. Alias, A. Darte and L. Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *SAS*, Lecture Notes in Computer Science. Springer, 2010.
7. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. POPL*. ACM, 1978.
8. K. Crary and S. Weirich. Resource bound certification. In *POPL'00*. ACM Press, 2000.
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. J. Hoffmann and M. Hofmann. Amortized Resource Analysis with Polynomial Potential. In *Proc. of ESOP*, volume 6012 of *Lecture Notes in Computer Science*, pages 287–306. Springer, 2010.
11. Kim Marriot and Peter Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.
12. Maxima.sourceforge.net. Maxima, a Computer Algebra System. Version 5.21.1 (2009). http://maxima.sourceforge.net/.
13. B. Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9), 1975.

Appendix F

Verified Resource Guarantees using COSTA and KeY

The paper "Verified Resource Guarantees using COSTA and KeY" [4] follows.

Verified Resource Guarantees using COSTA and KeY

Elvira Albert
Complutense University of Madrid
elvira@sip.ucm.es

Richard Bubel
Chalmers University of Technology
bubel@chalmers.se

Samir Genaim
Complutense University of Madrid
samir.genaim@fdi.ucm.es

Reiner Hähnle
Chalmers University of Technology
reiner@chalmers.se

Germán Puebla
Technical University of Madrid
german@fi.upm.es

Guillermo Román-Díez
Technical University of Madrid
groman@fi.upm.es

Abstract

Resource guarantees allow being certain that programs will run within the indicated amount of resources, which may refer to memory consumption, number of instructions executed, etc. This information can be very useful, especially in real-time and safety-critical applications. Nowadays, a number of automatic tools exist, often based on type systems or static analysis, which produce such resource guarantees. In spite of being based on theoretically sound techniques, the implemented tools may contain bugs which render the resource guarantees thus obtained not completely trustworthy. Performing full-blown verification of such tools is a daunting task, since they are large and complex. In this work we investigate an alternative approach whereby, instead of the *tools*, we formally verify the *results* of the tools. We have implemented this idea using COSTA, a state-of-the-art static analysis system, for producing resource guarantees and KeY, a state-of-the-art verification tool, for formally verifying the correctness of such resource guarantees. Our preliminary results show that the proposed tool cooperation can be used for automatically producing verified resource guarantees.

Categories and Subject Descriptors F3.2 [*Logics and Meaning of Programs*]: Program Analysis; F2.9 [*Analysis of Algorithms and Problem Complexity*]; D3.2 [*Programming Languages*]

General Terms Languages, Theory, Verification, Reliability

Keywords Static Analysis, Resource Guarantees, Java

1. Introduction

There is a growing awareness, both in industry and academia, of the crucial role of formally proving the correctness of systems. Verifying the correctness of modern static analyzers is rather challenging, among other things, because of the sophisticated algorithms used in them, their evolution over time, and, possibly, proprietary considerations. A simpler alternative is to construct a validating tool [7] which, after every run of the analyzer, formally confirms that the results are correct and, optionally, generates correctness proofs. Such proofs could then be translated to *resource certificates* [5, 6].

PEPM'11 January 24–25, 2011, Austin, Texas, U.S.A.

In this work, we are interested in *resource guarantees* obtained by static analysis. An essential aspect of programs is that resources be used effectively. This is especially true in the current programming trends, which provide us with mechanisms for code reuse by means of components and services: not only functionality, but also resource consumption (or *cost*) must be taken into consideration.

COSTA is a state-of-the-art COSt and Termination Analyzer for Java bytecode (and hence Java). It receives as input the bytecode of a Java program, the signature of the method whose cost is to be inferred, a choice of one among several available cost models (termination [1], number of bytecode instructions [3], memory consumption, or calls to certain method) and automatically infers an *upper bound* (UB for short) on the cost as a function of the method's input arguments. The most challenging step is to infer UBs for the loops in the program [2]. Intuitively, this requires (1) bounding the number of iterations of each loop and (2) finding the worst-case cost among all iterations. *Ranking functions* [8] give us safe approximations for requirement (1). To infer the maximal cost in requirement (2), we need to track how the values of variables change in the loop iterations and the inter-relations between (the values of) variables. As we will see, this information is obtained in COSTA by means of *loop invariants* and *size relations*. The analysis algorithms used in COSTA for inferring the main components of the UB generation were proven correct at a theoretical level. However, there is no guarantee that correctness is preserved in the actual implementation which is rather involved.

KeY [4] is a state-of-the-art source code verification tool for the Java programming language. Its coverage of Java is comparable to that of COSTA (nearly full sequential Java, plus a simplified concurrency model). KeY implements a logic-based setting of symbolic execution that allows deep integration with aggressive first-order simplification. While the degree of automation of KeY is very high on loop- and recursion-free programs, the user must in general supply suitable invariants to deal with loops and recursion. In general, invariants that are sufficient to prove complex functional properties cannot be inferred automatically. However, simpler invariants that are sufficient to establish UBs *can* be automatically derived in many cases and this is exactly COSTA's forte. Our work is based on the insight that the static analysis tool COSTA and the formal verification tool KeY have complementary strengths: COSTA is able to derive UBs of Java programs including the invariants needed to obtain them. This information is enough for KeY to *prove* the validity of the bounds and provide a certificate. The main contribution of this work is to show that, using KeY, it is possible to formally and automatically verify the correctness of the UBs obtained by COSTA.

2. Inference of Upper Bounds in COSTA

In this section, we briefly describe the techniques used in COSTA for automatically inferring UBs, and we identify the proof obligations that need to be verified using KeY.

2.1 Main Components of an Upper Bound

Consider the following (JML annotated) program that implements the insert sort algorithm.

```
1 void insert_sort (int A[]) {
2   int i, j, v;
3   //@ ghost int i0=i; int j0=j; int a0=a;
4   i=A.length-2;
5   //@ assert (i=i0-2 ∧ j=j0 ∧ a=a0)
6   //@ ghost int i1=i; int j1=j; int a1=a;
7   //@ loop_invariant i≤i1
8   //@ decreases i>0 ? i : 0
9   while ( i>=0 ) {
10    //@ ghost int i2=i; int j2=j; int a2=a;
11    j=i+1;
12    v=A[i];
13    //@ assert j=i2+1 ∧ i2 ≥ 0
14    //@ ghost int i3=i; int j3=j; int a3=a;
15    //@ loop_invariant j≤a3
16    //@ decreases a-j>0 ? a-j : 0
17    while ( j<A.length && A[j]<v) {
18      A[j-1]=A[j];
19      j++; }
20    A[j-1]=v;
21    i--; } }
```

COSTA receives a non-annotated version of the above program and, for the cost model that counts the number of executed bytecode instructions, produces the (asymptotic) UB insert_sort(a)=a^2, where a refers to A.length. The underlying analysis used in COSTA infers UBs for each iterative and recursive constructs (loops) and then composes the results in order to obtain an UB for the method of interest. Intuitively, in order to infer an UB for a single loop, it first infers an UB A on the cost of a single execution of its body, an UB I on the number of iterations that it can make, and then $A * I$ is an UB for the loop. In order to infer A and I COSTA relies on several program analysis components that provide essential information:

Ranking functions. For each loop, COSTA infers a linear function from the loop variables to $\mathbb{N}$ which is decreasing at each iteration. For example, for the loop at line 17, it infers function $f(a,j) = \mathsf{nat}(a-j)$ where $\mathsf{nat}(\ell) = \max(0,\ell)$. This function can be safely used to bound the number of iterations. In the example, if a_3 and j_3 are the initial values of a and j, then it is guaranteed that $f(a_3, j_3)$ is an UB on the number of iterations of the loop.

Loop invariants. For each loop in the program, COSTA infers an invariant that involves the loop's variables and their initial values (i.e., their values before entering the loop). Let us denote by i_1 the initial value of i when entering the loop at line 9. COSTA infers the invariant $i \leq i_1$, which states that i is always smaller than or equal to its initial value when the program reaches the loop condition. This information, together with the size relations below, is needed to compute the worst-case cost of executing one loop iteration.

Size relations. Given a fragment of code or a scope (details below), COSTA infers relations between the values of the program variables at a certain program point of interest within the scope and their initial values when entering the scope. For example, at program point 13, it infers that $j = i_2 + 1$, where i_2 is the value of i when entering the scope that contains line 13 (i.e., the scope here is the loop body). In this case the relation is a simple consequence of the instruction at line 11. In general, however, it may not be trivial to infer such relations nor to prove that they are correct.

Upper Bounds. Once the above information has been inferred, it is straightforward to compute an UB for the method. Let us show this process on the running example:

Inner loop. The process starts from the innermost loops. Thus, we start with the loop at line 17. Assuming that executing the condition costs (at most) c_1 instructions, and that the cost of each iteration (i.e., the loop body) is c_2 instructions, then it is clear that $\mathsf{nat}(a_3 - j_3) * (c_1 + c_2) + c_1$ is an UB on the cost of this loop (because c_1 and c_2 are constant).

Outer loop. Next, we move to the outer loop at line 9. Let us assume that the cost of the comparison is c_3 instructions, the code at lines 11–12 are c_4 instructions, and the code at lines 20–21 are c_5 instructions. Then, the cost of each iteration of this loop is $c_3 + c_4 + \mathsf{nat}(a_3 - j_3) * (c_1 + c_2) + c_1 + c_5$, where the highlighted subexpression corresponds to the cost of the inner loop computed above. Note that in this case, each iteration might have a different cost, since $a_3 - j_3$ is not the same for all iterations. Simply multiplying the number of iterations $\mathsf{nat}(i_1)$ by such a cost is unsound. The solution is to find an expression U in terms of the initial values of a_1, i_1, j_1 which does not change during the loop such that $U \geq a_3 - j_3$ in all iterations. Then, $\mathsf{nat}(i_1) * [c_3 + c_4 + \mathsf{nat}(U) * (c_1 + c_2) + c_1 + c_5] + c_3$ is an UB for the loop. In order to find such a U, COSTA uses the loop invariant (line 7) and the size relations (line 13) as follows: it solves the parametric integer programming (PIP) problem of maximizing the objective function $a_3 - j_3$ w.r.t. the loop invariant and the size relations where i_1, a_1, j_1 are the parameters. This produces an expression in terms of i_1, a_1, j_1 which is greater than or equal to $a_3 - j_3$ in all iterations of the loop. In our example, it is $U = a_1 - 1$.

Method. We finally can compute the cost of the insert_sort method. Assume that the cost of line 4 is c_6, then the cost of the method is $c_6 + \mathsf{nat}(i_1) * [c_3 + c_4 + \mathsf{nat}(a_1 - 1) * (c_1 + c_2) + c_1 + c_5] + c_3$. We need to express this UB in terms of the input parameter a. For this, COSTA maximizes (using PIP) i_1 and $a_1 - 1$ w.r.t. the size relation at line 5 and, respectively, obtains $a-2$ and $a-1$. Therefore, $c_6 + \mathsf{nat}(a-2) * [c_3 + c_4 + \mathsf{nat}(a-1) * (c_1 + c_2) + c_1 + c_5] + c_3$ is the UB for insert_sort.

2.2 COSTA Claims as JML Annotations

To justify that the UBs obtained by COSTA are correct, we need to provide formal correctness proofs for all the claims above. This includes the ranking functions, invariants, size relations, the cost model that provides all c_i, and the underlying PIP solver.

Correctness of the cost model is trivial as it is a simple mapping from each instruction to a number. Correctness of the underlying PIP solver is also straightforward if we use the maximization procedure defined in [2], which is based only on the Gaussian elimination algorithm. Therefore, we concentrate on verifying the correctness of the ranking functions, size relations and invariants. They are inferred by large software components whose correctness has not been verified. We now briefly describe the translation of the different pieces of information generated by COSTA to JML annotations on the Java program, which will allow their verification in KeY.

Ranking functions. For a given loop, when COSTA infers a ranking function of the form $\mathsf{nat}(\ell)$, we translate it to the JML annotation "//@ decreasing $\ell > 0$? ℓ : 0", since $\mathsf{nat}(\ell)$ can be defined as an if-then-else. COSTA might provide also ranking functions of the form $\log(\mathsf{nat}(\ell) + 1)$, which are handled similarly.

Invariants. COSTA infers an invariant φ for each loop. This invariant involves the loop variables $\bar{v}$ and auxiliary variables $\bar{w}$ such that each w_i represents the initial value of v_i. The JML annotation

for this invariant consists of one line defining all $\bar{w}$ as ghost variables ("//@ ghost int $w_1 = v_1$;...; int $w_n = v_n$") and one line for declaring the loop invariant ("//@ loop_invariant φ").

Size relations. Size relations are linear constraints between the values of a set of variables of interest between two program points. As we have seen, this allows composing the cost of the different program fragments. For each loop (or method call), COSTA infers the relation φ between the values before the loop entry (or the call) and the entry of its parent scope. Suppose that the loop (or the call) is at line L_l, its parent scope starts at line L_p, and that $\bar{v}$ are the variables of interest at L_l and $\bar{w}$ represent their values at L_p. Then we add the JML annotation "//@ ghost int $w_1 = v_1$;... ; int $w_n = v_n$;" immediately after line L_p to capture the values of $\bar{v}$ at line L_p, and the JML annotation "//@ assert φ" immediately before line L_l to state that the relation φ must hold at the program point. Additional size relations inferred by COSTA are input-output size relations. These are linear constraints that relate the return value of a given method to its input values. For example, suppose that we replace "i −−" in line 21 of the insert_sort program by "i =decrement(i)" where decrement is defined by "**int** decrement(**int** x) {**return** x−1;}". Then COSTA infers the relation "$\varphi \equiv$\result= x−1" which is used to bound the number of iterations of that loop. In order to verify this relation in KeY we add the JML annotation "//@ ensures φ" to the contract of decrement.

3. Verification of Upper Bounds using KeY

We now describe the verification techniques used in KeY to prove program correctness, focusing on those relevant to UB verification.

3.1 Verification by Symbolic Execution

The program logic used by KeY is *JavaCard Dynamic Logic* (JavaDL) [4], a first-order dynamic logic with arithmetic. Programs are first-class citizens similar to Hoare logics but, in dynamic logic, correctness assertions can appear arbitrarily nested. JavaDL extends sorted first-order logic by a program modality $\langle\cdot\rangle\cdot$ (read "diamond"). Let p denote a sequence of executable Java statements and ϕ an arbitrary JavaDL formula, then $\langle \mathsf{p} \rangle \phi$ is a JavaDL formula which states that program p terminates and in its final state ϕ holds. A typical formula in JavaDL looks like

$$\mathsf{i} \doteq i0 \wedge \mathsf{j} \doteq j0 \rightarrow \langle \overbrace{\mathsf{i=j-i;j=j-i;i=i+j;}}^{\mathsf{p}} \rangle (\mathsf{i} \doteq j0 \wedge \mathsf{j} \doteq i0)$$

where i, j are program variables represented as *non-rigid* constants. Non-rigid constants and functions are state-dependent: their value can be changed by programs. The *rigid* constants $i0, j0$ are state-independent: their value cannot be changed. The formula above says that if program p is executed in a state where i and j have values $i0$, $j0$, then p terminates *and* in its final state the values of the variables are swapped. To reason about JavaDL formulas, KeY employs a sequent calculus whose rules perform *symbolic execution* of the programs in the modalities. Here is a typical rule:

$$\mathsf{ifSplit}\ \frac{\Gamma, b \Longrightarrow \langle \mathsf{\{p\}rest} \rangle \phi, \Delta \quad \Gamma, \neg b \Longrightarrow \langle \mathsf{\{q\}rest} \rangle \phi, \Delta}{\Gamma \Longrightarrow \langle \mathsf{if\ (b)\ \{p\}\ else\ \{q\}\ rest} \rangle \phi, \Delta}$$

As values are symbolic, it is in general necessary to split the proof whenever an implicit or explicit case distinction is executed. It is also necessary to represent the *symbolic* values of variables throughout execution. This becomes apparent when statements with side effects are executed, notably assignments. The assignment rule in JavaDL looks as follows:

$$\mathsf{assign}\ \frac{\Gamma \Longrightarrow \{\mathsf{x} := \mathsf{val}\} \langle \mathsf{rest} \rangle \phi, \Delta}{\Gamma \Longrightarrow \langle \mathsf{x = val;\ rest} \rangle \phi, \Delta}$$

The expression in curly braces in the premise is called *update* and is used in KeY to represent symbolic state changes. An *elementary* update $loc := val$ is a pair of a location (program variable, field, array) and a value. The meaning of updates is the same as that of an assignment, but they can be composed in different ways to represent complex state changes. Updates u_1, u_2 can be composed into *parallel updates* $u_1 \| u_2$. In case of clashes (updates u_1, u_2 assign different values to the same location) a last-wins semantics resolves the conflict. This reflects left-to-right sequential execution. Apart from that, parallel updates are applied simultaneously, i.e., they do not depend on each other. Update application to a formula/term e is denoted by $\{u\}e$ and forms itself a formula/term. Application of updates is similar to explicit substitutions, but is aware of aliasing.

Loops and recursive method calls give rise to infinitely long symbolic executions. Invariants are used in order to deal with unbounded program structures (an example is given below). Exhaustive application of symbolic execution and invariant rules results in formulas of the form $\{u\}\langle\rangle\phi$ where the program in the modality has been fully executed. At this stage, symbolic updates are applied to the postcondition ϕ resulting in a first-order formula that represents the weakest precondition of the executed program wrt ϕ.

3.2 Proof-Obligation for Verifying Upper Bounds

To verify UBs in KeY the annotated source code files provided by COSTA are loaded. For methods where COSTA did not generate a contract, KeY provides the following default contract:

```
/*@ public behavior
  @ requires true;
  @ ensures true;
  @ signals_only Exception;
  @ signals (Exception) true; @*/
```

This contract requires to prove termination for any input and ensures that all possible execution paths are analyzed. Abrupt termination by uncaught exceptions is allowed (signals clauses). To prove that a method m satisfies its contract, a JavaDL formula is constructed which is valid iff m satisfies its contract. Slightly simplified, for insert_sort this formula (using the default contract) is:

$$\forall o; \forall a0; \{\mathsf{a} := a0 \parallel \mathsf{self} := o\}(\neg(\mathsf{a} \doteq \mathsf{null}) \wedge \neg(\mathsf{self} \doteq \mathsf{null}) \rightarrow \langle\ \mathsf{try\ \{\ self.insert\_sort(a)@NestedLoops;\ \}\ catch(Exception\ e)\{\ exc=e;\ \}} \rangle (\mathsf{exc} \doteq \mathsf{null} \vee \mathsf{instance}_{\mathsf{Exception}}(\mathsf{exc}))$$

The above formula states that for any possibly value o of self and any value $a0$ of the argument a which satisfy the implicit JML preconditions (self and a are not null), the method invocation self.insert_sort(a) *terminates* (required by the use of the diamond modality) and in its final state no exception has been thrown or any thrown exception must be of type Exception.

3.3 Verification of Proof-Obligations

The proof obligation formula must be proven valid by executing the method insert_sort symbolically starting with the execution of the variable declarations. Ghost variable declarations and assignments to ghost variables (//@ set var=val;) are symbolically executed just like Java assignments.

Verifying Size Relations. If a JML assertion assert φ; is encountered during symbolic execution, the proof is split: the first branch must prove that the assertion formula φ holds in the current symbolic state; the second branch continues symbolic execution. In the insert_sort example, a proof split occurs exactly before entering each loop. This verifies the size relations among variables as derived by COSTA and encoded in terms of JML assertion statements (see Sect. 2.2). Input-output size relations encoded in terms of method contracts are proven correct as outlined in Sect. 3.2.

Verifying Invariants and Ranking Functions. Verification of the loop invariants and ranking functions obtained from COSTA is achieved with a tailored loop invariant rule that has a variant term to ensure termination:

$$\mathsf{loopInv}\ \frac{\begin{array}{ll}(i) & \Gamma \Longrightarrow Inv \wedge dec \geq 0, \Delta \\ (ii) & \Gamma, \{\mathcal{U}_A\}(b \wedge Inv \wedge dec \doteq d0) \ \Rightarrow \\ & \quad \{\mathcal{U}_A\}\langle \mathsf{body}\rangle(Inv \wedge dec < d0 \wedge dec \geq 0), \Delta \\ (iii) & \Gamma, \{\mathcal{U}_A\}(\neg b \wedge Inv) \Longrightarrow \{\mathcal{U}_A\}\langle \mathsf{rest}\rangle\phi, \Delta \end{array}}{\Gamma \Longrightarrow \langle \mathsf{while\ (b)\ \{\ body\ \}\ rest}\rangle\phi, \Delta}$$

Inv and dec are obtained, respectively, from the loop_invariant and decreasing JML annotations generated by COSTA. Premise (i) ensures that invariant Inv is valid just before entering the loop and that the variant dec is non-negative. Premise (ii) ensures that Inv is preserved by the loop body and that the variant term decreases strictly monotonic while remaining non-negative. Premise (iii) continues symbolical execution upon loop exit. The integer-typed variant term ensures loop termination as it has a lower bound (0) and is decreased by each loop iteration. Using COSTA's derived ranking function as variant term obviously verifies that the ranking function is correct. The update $\mathcal{U}_A$ assigns to all locations whose values are potentially changed by the loop a fixed, but unknown value. This allows using the values of locations that are unchanged in the loop during symbolic execution of the body.

Generated Proofs. A single proof for each method is sufficient to verify the correctness of the derived loop invariants, ranking functions and size relations. The reason is that the contracts capturing the input-output size relations are not more restrictive w.r.t. the precondition than the default contracts are. Hence, with the verification of the input-output size relation contracts, we analyze all feasible execution paths and prove correctness of all loop invariants, ranking functions and JML assertion annotations. We stress that the proofs run fully automatic. Much of the time is needed to derive specific instances of arithmetic properties. As future work, we plan to do proof profiling and to reduce the search time by hashing frequently occuring normalisation steps.

4. Implementation and Experiments

The implementation of our approach has required the following non-trivial extensions to COSTA and KeY (note that COSTA works on Java bytecode, and KeY on Java source): (1) output the proof obligations using the original variable names (at the bytecode level, operand stack variables are often used); (2) place the obligations in the Java source at the precise program points where they must be verified (entry points of loops); (3) finding a suitable JML format for representing proof obligations on UBs has required a considerable number of iterations (defining ghost variables, introducing assert constructs, etc.); (4) implement the JML assert construct in KeY which was not supported hitherto. To express assertions which have to hold before a method call but after parameter binding support for a second assertion construct invocAssert has been added.

Eclipse plugins for both the extended COSTA and KeY systems are available from `http://pepm2011.hats-project.eu`. Source code for the tools (under GPL) is planned in the near future.

Table 1 shows some preliminary experiments using a set of representative programs, available from the above website, which include sorting algorithms, namely bubble sort (*bubsort*), insert sort (*inssort*), and selection sort (*selsort*); a method to generate a Pascal Triangle (*pastri*); simple (*slm*) and nested loops (*nlf*). Columns $\mathbf{T}_{size}$, $\mathbf{T}_{inv}$, $\mathbf{T}_{rf}$, $\mathbf{T}_{ana}$ and $\mathbf{T}_{jml}$ show, respectively, the times taken by COSTA to obtain the size relations, loop invariants, ranking functions, the whole analysis (which includes the previous times) and generate the JML annotations. Column $\mathbf{T}_{ver}$ shows the time taken by KeY in order to verify the JML annotations generated by COSTA. As time measurements for Java are imprecise we state in addition the number of nodes and branches of the generated proof to provide some insight on the proof complexity. Column **Total** shows the time taken by the whole process. All times are measured in ms and were obtained using an Intel Core2 Duo P8700 at 2.53GHz with 4Gb of RAM running a Linux 2.6.32 (Ubuntu Desktop). A notable result of our experiments is that KeY was able to spot a bug in COSTA, as it failed to prove correct one invariant which was incorrect. In addition, KeY could provide a concrete counterexample that helped understand, locate and fix the bug, which was related to a recently added feature of COSTA.

Bench	COSTA					KeY			**Total**
	$\mathbf{T}_{size}$	$\mathbf{T}_{inv}$	$\mathbf{T}_{rf}$	$\mathbf{T}_{ana}$	$\mathbf{T}_{jml}$	**Nodes**	**Branches**	$\mathbf{T}_{ver}$	
slm	22	20	26	112	4	3641	36	6700	6816
nlf	30	16	24	106	6	5665	37	2800	2912
bubsort	38	24	144	296	14	14890	230	57800	58110
inssort	30	12	46	142	6	9875	167	29300	29448
selsort	40	20	112	232	8	12564	209	40700	40940
pastri	66	38	138	394	14	29723	337	110100	110508

Table 1. Statistics about the Analysis and Verification Process

5. Conclusions and Future Work

We have demonstrated that automatic verification of the upper bounds inferred by COSTA using KeY is feasible. Instead of verifying the correctness of the underlying static analysis, we take the alternative approach of verifying the correctness of their results. Interestingly, this approach, though weaker in principle than verification of the analyzer, has advantages in the context of mobile code. Following proof-carrying-code [6] principles, code originating from an untrusted *producer* can be bundled together with the proof generated by KeY for its declared resource consumption. This way, the code *consumer* can check locally and automatically using KeY whether the claimed resource guarantees are verified. As future work, we plan to extend our approach to support programs that manipulate data structures other than arrays.

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 TIN-2008-05624 *DOVES*, by UCM-BSCH-GR58/08-910502 (GPD-UCM) and S2009TIC-1465 *PROMETIDOS* project.

References

[1] E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination Analysis of Java Bytecode. In *FMOODS'08*, volume 5051 of *LNCS*, pages 2–18. Springer, 2008.

[2] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 2010. To appear.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *ESOP'07*, volume 4421 of *LNCS*, pages 157–172. Springer, 2007.

[4] B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2006.

[5] K. Crary and S. Weirich. Resource Bound Certification. In *POPL'05*, pages 184–198. ACM Press, 2000.

[6] G. Necula. Proof-Carrying Code. In *POPL 1997*. ACM Press, 1997.

[7] A. Pnueli, M. Siegel, and E. Singerman. Translation Validation. In *TACAS'98*, volume 1384 of *LNCS*, pages 151–166. Springer, 1998.

[8] A. Podelski and A. Rybalchenko. A Complete Method for the Synthesis of Linear Ranking Functions. In *VMCAI'04*, LNCS. Springer, 2004.

Appendix G

Closed-Form Upper Bounds in Static Cost Analysis

The paper "Closed-Form Upper Bounds in Static Cost Analysis" [9] follows.

J Autom Reasoning (2011) 46:161-203
DOI 10.1007/s10817-010-9174-1

Closed-Form Upper Bounds in Static Cost Analysis

Elvira Albert · Puri Arenas
Samir Genaim · Germán Puebla

Abstract The classical approach to automatic cost analysis consists of two phases. Given a program and some measure of cost, the analysis first produces *cost relations* (*CRs*), i.e., recursive equations which capture the cost of the program in terms of the size of its input data. Second, *CRs* are converted into *closed-form*, i.e., without recurrences. Whereas the first phase has received considerable attention, with a number of cost analyses available for a variety of programming languages, the second phase has been comparatively less studied. This article presents, to our knowledge, the first practical framework for the generation of closed-form upper bounds for *CRs* which (1) is fully automatic, (2) can handle the distinctive features of *CRs* originating from cost analysis of realistic programming languages, (3) is not restricted to simple complexity classes, and (4) produces reasonably accurate solutions. A key idea in our approach is to view *CRs* as programs, which allows applying semantic-based static analyses and transformations to bound them, namely our method is based on the inference of *ranking functions* and *loop invariants* and on the use of *partial evaluation*.

Keywords Cost analysis, closed-form upper bounds, resource analysis, automatic complexity analysis, static analysis, abstract interpretation, programming languages.

1 Introduction

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

Elvira Albert · Puri Arenas · Samir Genaim
DSIC, Complutense University of Madrid (UCM), E-28040 Madrid, Spain
E-mail: {elvira,puri}@sip.ucm.es,samir.genaim@fdi.ucm.es

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

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

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

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

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

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

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

Though cost relations are simpler than the programs they originate from, since all variables are of integer type, in several respects they are not as static as one would expect from the result of a static analysis. One reason is that they are recursive and thus we may need to iterate for computing their value for concrete input values. Another reason is that even for deterministic programs, it is well known that the loss of precision introduced by the size abstraction may result in cost relations which are non-deterministic. This happens in the above example: since the array v has been abstracted to its length v, the values of v[i] are unknown statically. Hence, the last

two equations (c) and (d) become non-deterministic choices. In general, for finding the worst-case cost we may need to compute and compare (infinitely) many results. For both reasons, it is clear that it is interesting to compute *closed-form* upper bounds for the cost relation, whenever this is possible, i.e., upper bounds which are not in recursive form. For instance, for the above example, we aim at inferring the closed-form upper bound $k_1+k_2+v*max(\{k_3+C_{m1}, k_4+C_{m2}\})$ where C_{m1} and C_{m2} are in turn closed-form upper bounds for the corresponding methods.

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

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

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

1.1 Applications of Upper Bounds of Cost Relations

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

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

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

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

2 Cost Relations vs. Recurrence Relations

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

2.1 Cost Relations for the Running Example

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

Example 1 The system COSTA [7] is an abstract interpretation-based COSt and Termination Analyzer for Java bytecode. It receives as input a bytecode program and (a

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

Fig. 1 Java code of running example

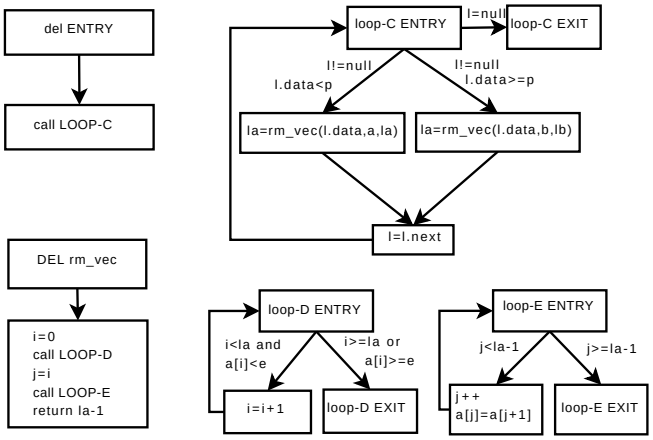


Fig. 2 Control flow graphs for running example

choice of) a resource of interest in the form of a cost model, and tries to obtain an upper bound of the resource consumption of the program. In Figure 2, we show the control flow graphs (CFG) constructed by COSTA in order to generate automatically the *CRs*. Such CFGs correspond to the graphs for the two methods (del and rm_vec) and separate CFGs for the loops, as in COSTA loop extraction is performed mainly for efficiency issues (see [2]). Although [6] analyzes Java bytecode and not Java source, we show the source for clarity of the presentation.

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

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

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

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

construct, its *CR* has a recursion. To carry out this step, analyzers usually build CFGs. In our example, we have three recursive cost relations C, D and E which correspond to the three CFGs for the loops in Figure 2:

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

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

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

After applying the above steps, the analyzer can set up the *CRs* shown in Figure 3 which we explain below. Equation (1) defines the cost of method del as 1 bytecode instruction plus the cost of the call to C. Observe also that the set of constraints contain

applicability conditions (i.e., *guards*) for each equation, if any, by providing constraints which only affect a subset of the variables in the left hand side (lhs for short). For clarity, we have inlined equality constraints (e.g., inlining equality $lb' = lb - 1$ is done by replacing all occurrences of lb' by $lb - 1$). The constraints attached to (1) are the (abstract) preconditions of the program. Among them, we have $a \geq la$ (resp. $b \geq lb$), which requires that the number of elements occupied in each array is less or equal than its length. Such preconditions are propagated properly to the rest of the equations.

In addition to Del, we have three recursive relations. As regards E, Equation (8) is its base case and it corresponds to the exit from the for loop, whereas Equation (9) counts the cost of each iteration in the loop. As expected, the value of j is increased by one at the recursive call to E. As regards the cost relation D, we have two base cases, Equations (5) and (6), which correspond to the exits from the loop because $\mathsf{i} \geq \mathsf{la}$ and because $\mathsf{a}[\mathsf{i}] \geq \mathsf{e}$, respectively. The important point here is that the second condition does not appear in the constraints of Equation (6) because this condition is not observable after abstracting the array a to its length, i.e., the value in $\mathsf{a}[\mathsf{i}]$ is *unknown*. For the selected cost model, we count 3 bytecode instructions in the first base case and 8 in the second one. The cost of executing an iteration of the loop is captured by (7), where the condition $i < la$ must be satisfied and variable i is increased by one at each recursive call.

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

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

2.2 Why Cost Relations are not Recurrence Relations ?

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

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

Fig. 4 Replacing multiple arguments with a single one

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

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

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

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

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

only depends on $y-x$ the latter takes different values depending on the initial value of x. E.g., $C(0,10) = C(1000,1010)$ but $D(0,10) \neq D(1000,1010)$.

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

Given a (non-deterministic) cost relation, it is sometimes useful to define a cost *function*. A relatively straightforward way of obtaining a cost function from non-deterministic *CRs* would be to introduce a maximization operator. Unfortunately, the cost functions thus produced are not very useful since existing CAS do not support the maximization operator. Adding it is far from trivial, since computing the maximum when the equations are not mutually exclusive requires taking into account multiple possibilities, which results in a highly combinatorial problem. This combinatorial explosion also affects the use of such cost-bound function in dynamic approaches, i.e., those based on executing cost-bound functions, such as [28].

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

Finally, let us observe that the properties listed above are all evident properties of constraint programs whose arguments are integer values. This explains the fact that we treat *CR* as programs and apply analysis and transformations delevoped for programming languages on them.

3 Cost Relations: Syntax and Semantics

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

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

over $\bar{x}$ as $\bar{x} = \bar{v}$, therefore we might write $\sigma \models \varphi$ for $\bar{x} = \bar{v} \models \varphi$. We use $\sigma(x)$ to refer to the value of x in σ, and $\sigma(\bar{x})$ for $\langle \sigma(x_1), \ldots, \sigma(x_n) \rangle$. The projection operator $\exists \bar{x}.\varphi$ (resp. $\bar{\exists} \bar{x}.\varphi$) projects the polyhedron defined by φ on the space $vars(\varphi) \setminus \bar{x}$ (resp. $\bar{x}$).

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

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

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

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

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

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

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

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

In contrast to standard definitions of *RRs*, in *CRSs*, the variables which occur in the rhs of the equations do not need to be related to those in the left hand side (lhs for short) by equality constraints. Other constraints such as $\leq$ and $<$ can also be used. We denote by $rel(\mathcal{S})$ the set of cost relation symbols which are defined in $\mathcal{S}$, i.e., which appear in the lhs of some equation in $\mathcal{S}$. Given a *CRS* $\mathcal{S}$ and a cost relation symbol C, the definition of C in $\mathcal{S}$, denoted $def(\mathcal{S}, C)$, is the subset of the equations in $\mathcal{S}$ whose

lhs is of the form $C(\bar{x})$. Without loss of generality, we assume that all equations in $def(\mathcal{S}, C)$ have the same variable names in the lhs, and that $\mathcal{S}$ is self-contained in the sense that all cost relation symbols which appear in the rhs of an equation in $\mathcal{S}$ must be in $rel(\mathcal{S})$.

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

The next definition provides a formal (denotational) semantics for *CRSs* which maps a call $C(\bar{v})$ to the set of all possible evaluation trees, and therefore the set of all possible answers. We will represent evaluation trees using nested terms of the form $node(Call, Local\_Cost, Children)$, where $Local\_Cost$ is a constant in $\mathbb{R}_+$ and $Children$ is a sequence of evaluation trees.

Definition 3 Given a cost relation system $\mathcal{S}$, the set of evaluation trees induced by an initial query $C(\bar{v})$ is defined as:

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

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

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

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

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

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

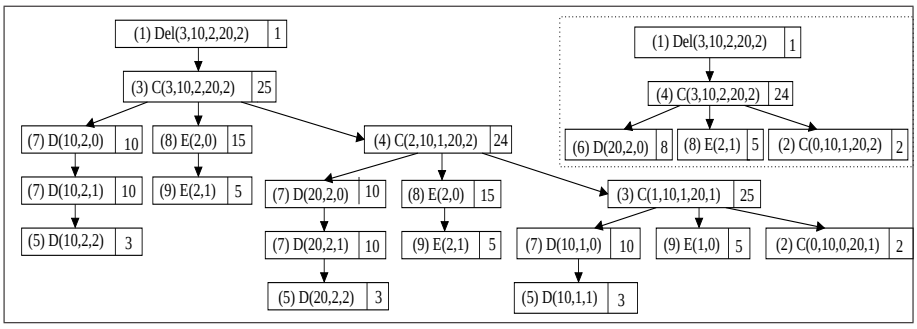


Fig. 5 Two evaluation trees for $Del(3, 10, 2, 20, 2)$

the equation which was selected for evaluating such call. The part on the right contains the local cost associated to the call, 1 in both root nodes. Nodes are linked by arrows to their children, if any.

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

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

4 Closed-Form Upper-Bounds for Cost Relations

After providing a suitable semantics for *CRs*, we now study how to obtain closed-form upper bounds for them. In what follows, we are only interested in upper-bound

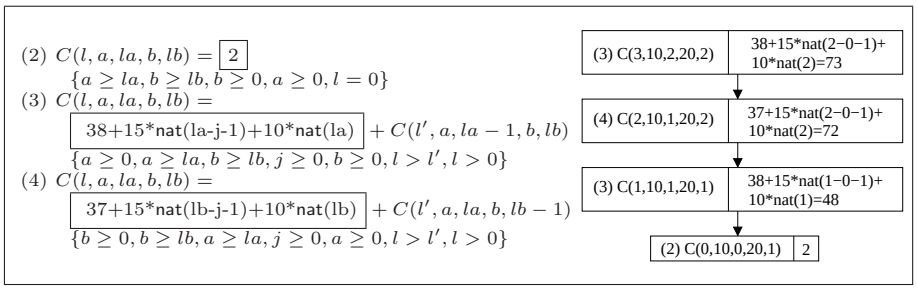


Fig. 6 Standalone *CR* for relation C and a corresponding evaluation tree

functions which are in closed-form. Therefore, for brevity, we often just write 'upper bound' instead of 'closed-form upper bound'.

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

4.1 Standalone Cost Relations

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

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

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

The compositionality principle only results in an effective mechanism if all recursions are *direct* (i.e., all cycles are of length one). In that case we can start by computing upper bounds for cost relations which do not depend on any other relations, which we refer to as *standalone cost relations* and continue by replacing the computed upper bounds on the equations which call such relations. In the following, we formalize our method by assuming standalone cost relations and, in Section 8, we provide a mechanism for obtaining direct recursion automatically.

4.2 Approximating Evaluation Trees

Existing approaches to compute upper bounds and asymptotic complexity of *RRs*, usually applied by hand, are based on reasoning about evaluation trees in terms of their size, depth, number of nodes, etc. They typically consider two categories of nodes: (1) *internal* nodes, which correspond to applying recursive equations, and (2) *leaves* of the tree(s), which correspond to the application of a base (non-recursive) case. The central idea then is to count (or obtain an upper bound on) the number of leaves and the number of internal nodes in the tree separately and then multiply each of these by an upper bound on the cost of the base case and of a recursive step, respectively. For instance, in the evaluation tree in Figure 6 for the standalone cost relation C, there are three internal nodes and one leaf. The values in the internal nodes, once performed the evaluation of the expressions are 73, 72, and 48, therefore 73 is the worst case. In the case of leaves, the only value is 2. Therefore, the tightest upper bound we can find using this approximation is $3 \times 73 + 1 * 2 = 221 \ \geq \ 73 + 72 + 48 + 2 = 193$.

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

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

$$C_+(\bar{x}) = \mathit{internal}_+(\bar{x}) * \mathit{costr}_+(\bar{x}) + \mathit{leaf}_+(\bar{x}) * \mathit{costnr}_+(\bar{x})$$

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

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

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

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

5 Upper Bounds on the Number of Nodes

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

branching factor and leaves are at the same depth. Clearly, complete trees provide an upper bound of the number of nodes of any tree with such height and branching factor. Therefore, we define $internal_+(\bar{x})$ and $leaf_+(\bar{x})$ as follows:

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

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

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

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

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

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

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

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

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

□

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

Definition 5 (ranking function for a loop) A function $f : \mathbb{Z}^n \mapsto \mathbb{Z}_+$ is a *ranking function* for a loop $\langle C(\bar{x}) \to C(\bar{y}), \varphi \rangle$ if $\varphi \models f(\bar{x}) > f(\bar{y})$.

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

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

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

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

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

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

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

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

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

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

Example 6 Consider the following two loops:

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

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

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

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

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

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

6 Bounding the Cost per Node

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

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

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

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

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

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

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

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

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

6.1 Invariants

Computing an *invariant*, in terms of linear constraints, that holds between the arguments at the initial call and at each call during the evaluation, can be done by using $Loops(C)$. Intuitively, if we know that a linear constraint ψ holds between the arguments of the initial call $C(\bar{x}_0)$ and the arguments of a specific recursive call $C(\bar{x})$ during the evaluation, denoted $\langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \psi \rangle$, and we have a loop $\langle C(\bar{x}) \to C(\bar{y}), \varphi \rangle \in Loops(C)$, then we can apply the loop one more step and get the new *calling context* (or *context* for short) $\langle C(\bar{x}_0) \rightsquigarrow C(\bar{y}), \exists \bar{x}.(\psi \wedge \varphi) \} \rangle$. The following definition describes how from a set of contexts I we learn more contexts by applying

all loops in a relation. We denote by $\mathcal{R}$ the set of all possible contexts for C, and by $\wp(\mathcal{R})$ all subsets of C that include $I_0 = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{\bar{x}_0 = \bar{x}\}\rangle$.

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

$$\mathcal{T}_C(X) = \left\{ \langle C(\bar{x}_0) \rightsquigarrow C(\bar{y}), \psi' \rangle \;\middle|\; \begin{array}{l} \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \psi \rangle \in X \\ \langle C(\bar{x}) \rightarrow C(\bar{y}), \varphi \rangle \in \mathit{Loops}(C) \\ \psi' = \bar{\exists} \bar{x_0} \cup \bar{y}.(\psi \wedge \varphi) \end{array} \right\}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.2 Upper Bounds on Cost Expressions

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

Consider, for example, the expression $\mathsf{nat}(la - j - 1)$ which appears in Equation (3) of Figure 6. We want to infer an upper bound of the values that it can be evaluated to in terms of the input values $\langle l_0, a_0, la_0, b_0, lb_0 \rangle$. We have inferred that $\langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \psi \rangle$ where $\psi = \{l \leq l_0, a = a_0, \underline{la \leq la_0}, b = b_0, lb \leq lb_0\}$, is a safe approximation of the loop invariant I_C, from which we can observe that the maximum value that la can take is la_0. In addition, from the local constraints φ of Equation (3) we know that $j \geq 0$. Since $la - j - 1$ takes its maximal value when la is maximal and j is minimal, the expression $la_0 - 1$ is an upper bound for $la - j - 1$. In practice, this inference method can be done in a fully automatic way using linear constraints tools (e.g. [13]) as follow:

1. compute $\phi = \bar{\exists} l_0, a_0, la_0, b_0, lb_0, r.(\psi \wedge \varphi \wedge y = la - j - 1)$, where y is a new variable;
2. *syntactically* look in ϕ for an expression that can be rewritten to $y \leq f'$, where f' is a linear expression which (obviously) contains only variables from $\{l_0, a_0, la_0, b_0, lb_0\}$.

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

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

This function computes an upper bound f' for each expression f which occurs inside a nat function and then replaces in exp all such f expressions with their corresponding upper bounds (line 5). If it cannot find an upper bound, the method returns ∞ (line 6).

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

$$\texttt{mexp}_3 = 38 + 15 * \mathsf{nat}(la_0 - 1) + 10 * \mathsf{nat}(la_0)$$
$$\texttt{mexp}_4 = 37 + 15 * \mathsf{nat}(lb_0 - 1) + 10 * \mathsf{nat}(lb_0)$$

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

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

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

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

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

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

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

6.3 Concluding Remarks

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

Theorem 1 *Let $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$ be a cost relation where $\mathcal{S}_1$ and $\mathcal{S}_2$ are respectively the sets of non-recursive and recursive equations for C. Let*

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

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

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

Proof Follows from Lemmata 2 and 3. □

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

	h_+	$costnr_+$	$costr_+$	Upper Bound
$E(la_0, j_0)$	$\texttt{nat}(la_0 - j_0 - 1)$	5	15	$5 + 15 * \texttt{nat}(la_0 - j_0 - 1)$
$D(a_0, la_0, i_0)$	$\texttt{nat}(la_0 - i_0)$	8	10	$8 + 10 * \texttt{nat}(la_0 - i_0)$

These upper bounds can then be substituted in the Equations (3) and (4) which results in the cost relation for C depicted in Figure 6. We have already computed a ranking function for C in Example 4, and $costnr_+$ and $costr_+$ in Example 8, which are then combined into:

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

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

7 Improving Accuracy in Divide and Conquer Programs

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

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

7.1 Level-count upper bound

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

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

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

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

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

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

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

7.2 Divide and Conquer Programs

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Proof It follows from Lemmata 3 and 5. □

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

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

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

8 Direct Recursion using Partial Evaluation

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

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

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

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

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

8.1 Binding Time Classification

We now recall some standard terminology on graphs. A *directed graph* G is a pair $\langle N, A\rangle$ where N is the set of *nodes* and $A \subseteq N \times N$ is the set of *arcs*. Given a graph $G = \langle N, A\rangle$, a set of nodes $S \subseteq N$ is *strongly connected* if $\forall n, n' \in S$ we have that n' is reachable from n. The *strongly connected components* of $G = \langle N, A\rangle$ is a partition of N into the largest possible strongly connected sets. Given a graph G we write $SCC(G)$ to denote its strongly connected components. Given a graph $G = \langle N, A\rangle$ and a set $S \subseteq N$, the *subgraph* of G w.r.t. S, denoted $G|_S$, is defined as $G|_S = \langle S, A \cap (S \times S)\rangle$. Also, given a strongly connected component S, a node $n \in S$ is a *covering point* for $G|_S$ if $G|_{S\setminus\{n\}}$ is an acyclic graph, i.e., n is a covering point of $G|_S$ if n is part of all cycles in $G|_S$. The problem of finding a minimal set of nodes to delete from a cyclic graph in order to convert it into an acyclic graph is also known as the *feedback vertex set* problem in computational complexity theory. The *feedback vertex set* decision problem is NP-complete in general, but for reducible graphs it is linear [50].

As explained in [50], control flow graphs originating from structured programming languages are often reducible, since usually there are no jumps to the the middle of a loop. Moreover, since our interest is only in checking if there exists a feedback set of size 1, when the graph is not reducible, we can solve it in quadratic time simply by removing a node n from $G|_S$ and checking if $G|_{S\setminus\{n\}}$ is acyclic.

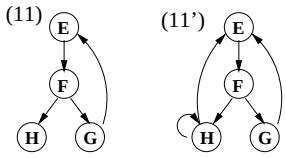
Note that, when the *CRS* originates from a structured program (i.e., without jumps), it is not common to have SCCs without covering points. This is due to: (1) As done in [6], each structured loop (e.g., while, for, etc.) can be transformed to a separated method in tail recursive form, and the loop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

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

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

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

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



The strongly connected component which could be problematic as regards minimal coverage (more than one element) is $\{E, F, G\}$. Since there is just one cycle, any of

the nodes is a covering point and therefore G is of minimal coverage. However, if we replace Equation (11) in Example 12 with Equation (11') below:

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

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

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

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

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

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

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

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

8.2 Partial Evaluation of Cost Relations

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

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

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

where we have that:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1. *Soundness.* The soundness condition ensures that the all answers in the partially evaluated program are also answers in the original program. It is proven by demonstrating that each unfolding step in the partially evaluated program corresponds to a sequence of equivalent steps in the original one. In our context, it amounts to ensuring that the operator Unfold of Definition 11 preserves the answers.
2. *Completeness.* Completeness guarantees that all answers in the original program are also found in the partially evaluated one. It can be ensured when the set of terms

to be partially evaluated meets the so-called *closedness* condition [39]. The role of this condition is to ensure that all possible calls that raise during the execution of a *CRS* will find a matching relation. In our context, we need to ensure that the set btc enforces the closedness condition, i.e., answers are not lost.

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

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

9 Incompleteness in Cost Analysis

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

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

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

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

   Importantly, this phase is complete for *CRs* extracted from structured loops and from the recursive patterns found in most programs. The use of features like `break`

and `continue` in languages like Java or C have do not pose any problem, since the control flow graph of the program can be constructed and the program can thus be turned into recursive form. As it can be seen in the example, incompleteness might occur in certain types of mutually recursive relations.

2. The second source of incompleteness in our method is in finding ranking functions. Currently, we use a complete procedure for inferring linear ranking functions [45]. However, there are *CRSs* which do not have a linear ranking function as explained in Example 6. Integrating other more sophisticated ranking functions is possible, but it is probably not required in practice.
3. The third one is finding useful invariants. Sometimes this is not possible by using linear constraints. This happens for example in this example:

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

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

10 Experimental Evaluation

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

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

Now we briefly describe the programs considered, which are listed in increasing complexity order and range from constant to exponential complexity, going through polynomial and divide and conquer. Polynomial is a method for copying polynomials and has a constant upper bound (on memory consumption). DivByTwo is a loop which iterates a logarithmic number of times, as its counter is decremented by half in each iteration. ArrayReverse produces a reversed copy of an array of integers. Concat concatenates two arrays of integers into a new array. Incr has a loop which iterates a linear number of times that depends on the run-time type of an input argument. ListReverse is an in-place reversal of a list represented as a linked list. MergeList merges two sorted lists implemented as linked lists. Power recursively computes the power operation. Cons copies a linked list. MergeSort sorts an array using the Merge Sort algorithm. EvenDigits is a simple `for` loop with a call to the DivbyTwo method inside the loop body. ListInter computes the intersection of two unsorted linked lists. SelectSort sorts an array by Selection Sort. FactSum adds up the factorial of all naturals from 0 to the input value n.

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

Table 1 Upper bounds computed automatically

Delete is the running example in Figure 1. MatMult multiplies two matrices. Hanoi has a doubly recursive structure, as the well-known Towers of Hanoi problem. Fibonacci is a naive doubly recursive implementation of Fibonacci numbers. Finally, BST is a method for recursively copying a binary search tree. In addition, in the experiments, we have used three different cost models:

- the heap consumption (in bytes), in those benchmarks marked with "*",
- the number of executed comparison instructions, in the benchmark marked with "n", and
- the number of executed bytecode instructions, in the rest of benchmarks.

10.1 Accuracy of the Upper-Bounds Obtained

The first set of experiments performed aims at evaluating the applicability of PUBS and the accuracy of the closed-form upper bounds thus obtained. Table 1 shows the upper bounds generated by PUBS for the benchmarks described above. The column **Properties** shows the properties of the corresponding *CR*, in such a way that a, b and c indicate, respectively, that the *CR* is non-deterministic, that it has inexact size constraints, and multiple arguments (Section 2.2). As can be seen, most of the benchmarks have one or more of such properties. If we handle the complete semantics of programs, including exceptions, even simple programs such as ArrayReverse are non-deterministic since accesses to arrays may in principle throw array-out-of-bounds exceptions. As a result, only the purely numerical programs, i.e., Power, FactSum, Hanoi, and Fibonacci are in a format syntactically acceptable by Mathematica® or other CAS. In contrast, PUBS has been able to automatically find upper bounds for all benchmarks considered. This clearly shows that CAS have rather restricted applicability in *CRs* obtained from

Benchmark	Input	Estimated	Actual	Accuracy
Polynomial*	copy_pol(10)	216	216	100
DivByTwo	divByTwo(10)	49	38	76
ArrayReverse	arrayReverse(10)	152	152	100
Concat	concat(10,10)	245	245	100
Incr	add(10,10)	218	218	100
ListReverse	listReverse(10)	138	138	100
MergeList	merge(5,5)	316	279	88
Power	power(10)	104	104	100
Cons*	cons(10)	222	222	100
MergeSort^n	ms_sort(_,_,0,5)	52	32	62
EvenDigits	evenDigits(10)	462	345	75
ListInter	listInter(5,5)	486	486	100
SelectSort	selectSort(6)	417	315	76
FactSum	doSum(10)	1172	677	58
Delete	delete(3,_,_,3,_,3)	297	256	86
MatMult	multiply(3,3)	866	866	100
Hanoi	hanoi(10)	20463	20463	100
Fibonacci	fibonacciMethod(10)	9203	1589	17
BST*	copy(4)	180	132	73

Table 2 Estimated versus actual maximal value

real programs. Column **Upper Bound** shows the closed-form upper bound obtained by PUBS. As can be seen, they are relatively syntactically simple. This is important since, as already mentioned in [54], one of the problems of cost analysis is that the cost functions produced can grow considerably large. This can hinder the success of cost analysis since large cost functions are hard to understand by humans and also difficult to automatically handle in applications such as resource certification [9], where it is required to compare cost functions [3].

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

Then, column **Estimated** provides the value obtained by evaluating the upper bound computed by PUBS on the given input data. Column **Actual** provides the actual value obtained by evaluating the cost-bound function discussed in Section 3, which is defined as $C_+(\bar{v}) = \max(\mathit{Answers}(C(\bar{v}), \mathcal{S}))$. For this we have implemented an evaluator for *CRSs* which given a *CRS* $\mathcal{S}$ and an initial call $C(\bar{v})$ produces all answers corresponding to all evaluation trees for $C(\bar{v})$ in $\mathcal{S}$ and then obtains the maximum of them. The evaluator has been implemented in *Constraint Logic Programming* [32] in order to efficiently handle the size constraints which are accumulated when obtaining the evaluation trees. It is important to note that due to the highly non-deterministic nature of many of the *CRs*, this evaluation often results in a combinatorial explosion which makes evaluation of most *CRs* unfeasible except for very small input values. This is why in some cases the input values are smaller than 10, which was the originally attempted input value for all arguments. We also use underscore to indicate arguments which do not affect the evaluation of the *CR*.

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

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

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

10.2 Efficiency and Scalability of the Approach

Table 3 aims at studying the efficiency of our system by showing the results of two different experiments. In the first experiment, we analyze each of the benchmarks in isolation. Column $\#_{eq}$ shows the number of equations before PE (in brackets after PE). Note that PE greatly reduces $\#_{eq}$ in all benchmarks. Column **T** shows the total runtime in milliseconds. The experiments have been performed on an Intel Core 2 Quad Q9300 at 2.50GHz with 1.95GB of RAM, running Linux 2.6.24-21. We argue that analysis times are acceptable. In the case of MergeSort analysis time is higher because its equations contain a large number of variables when compared to those of the other examples. This affects the efficiency when computing the ranking function and also when maximizing expressions.

Benchmark	$\#_{eq}$	T	$\#^c_{eq}$	$\mathbf{T}_{pe}$	$\mathbf{T}_{ub}$	Rat.
Polynomial*	23(3)	10	385(97)	388	1190	4.1
DivByTwo	9(3)	2	362(94)	402	1173	4.3
ArrayReverse	9(3)	2	344(88)	387	1122	4.4
Concat	14(5)	10	335(85)	386	1102	4.4
Incr	28(5)	23	321(80)	384	1046	4.5
ListReverse	9(3)	4	293(75)	374	943	4.5
MergeList	21(4)	17	284(72)	374	925	4.6
Power	8(2)	2	262(67)	366	898	4.8
Cons*	22(2)	6	253(64)	376	912	5.1
MergeSort^n	39(12)	499	230(61)	354	805	5.0
EvenDigits	18(5)	7	191(49)	130	290	2.2
ListInter	37(9)	48	173(44)	126	246	2.2
SelectSort	19(6)	22	136(35)	115	169	2.1
FactSum	17(5)	8	117(29)	109	143	2.2
Delete	33(9)	106	100(24)	102	130	2.3
MatMult	19(7)	17	67(15)	69	34	1.5
Hanoi	9(2)	5	48(8)	67	16	1.7
Fibonacci	8(2)	4	39(6)	63	11	1.9
BST*	31(4)	36	31(4)	64	8	2.3

Table 3 Scalability of upper bounds inference

The second experiment aims at studying how analysis time increases when larger *CRs* are used as benchmarks, i.e., the scalability of our approach. In order to do so, we have connected together the *CRs* for the different benchmarks by introducing a call from each *CR* to the one appearing immediately below it in the table. Such call is always introduced in a recursive equation. The results of this second experiment are shown in the last four columns of the table. Column $\#^c_{eq}$ shows the number of equations we want to solve in each case (in brackets after PE). Reading this column bottom-up, we can see that when we analyze BST in the second experiment we have the same number of equations as in the first experiment. Then, for Fibonacci we have its 8 equations plus 31 which have been previously accumulated. Progressively, each benchmark adds its own number of equations to $\#^c_{eq}$. Thus, in the first row we have a *CRS* with all the equations connected, i.e., we compute a closed-form upper bound of a *CRS* with at least 20 nested loops and 385 equations. In this experiment, the analysis time is split into $\mathbf{T}_{pe}$ and $\mathbf{T}_{ub}$, where $\mathbf{T}_{pe}$ is the time of PE and $\mathbf{T}_{ub}$ is the time of all other phases. The results show that even though PE is a *global* transformation, its time efficiency is linear with the number of equations, since PE operates on strongly connected components. Our system solves 385 equations in $388 + 1190ms$.

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

are currently implementing standard techniques for simplification of arithmetic expressions.

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

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

11 Related Work

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

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

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

The transformational view consists in regarding *CRs* as (functional) programs. In this view, closed-form upper bounds are produced by applying (general-purpose) pro-

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

The problem with all the approaches mentioned above is that, though they can be successfully applied for obtaining closed-forms for *CRs* generated from simple programs, they do not fulfill the initial expectations in that they are not of general applicability to *CRs* generated from real programs. The essential features which neither the algebraic nor the transformational approaches can handle are discussed in Section 2.2. The main motivation for this work was our own experience in trying to apply the algebraic approach on the *CRs* generated by [6]. We argue that automatically converting *CRs* into the format accepted by CASs is unfeasible. Furthermore, even in those cases where CASs can be used, the solutions obtained are so complicated that they become useless for most practical purposes. In contrast, our approach can produce correct and comparatively simple results even in the presence of non-determinism.

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

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

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

12 Conclusions

We have proposed an approach to the automatic inference of non-asymptotic closed-form upper bounds of *CRs* produced by automatic cost analysis. For this, we have formally defined *CRs* as a target language for cost analysis. Hence, our method for

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

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

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

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
2. E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination Analysis of Java Bytecode. In *10th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'08)*, volume 5051 of *Lecture Notes in Computer Science*, pages 2–18. Springer, 2008.
3. E. Albert, P. Arenas, S. Genaim, I. Herraiz, and G. Puebla. Comparing cost functions in resource analysis. In *1st International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA'09)*, Lecture Notes in Computer Science. Springer, 2009. To appear.
4. 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.
5. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Cost Relation Systems: a Language–Independent Target Language for Cost Analysis. In *8th Spanish Conference on Programming and Computer Languages (PROLE'08)*, volume 17615 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2008.
6. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *16th European Symposium on Programming, (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 2007.
7. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *6th International Symposioum on Formal Methods for Components and Objects (FMCO'08)*, number 5382 in Lecture Notes in Computer Science, pages 113–133. Springer, 2007.

8. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Experiments in Cost Analysis of Java Bytecode. In *2nd Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'07)*, volume 190, Issue 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2007.
9. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Resource usage analysis and its application to resource certification. In *9th International School on Foundations of Security Analysis and Design (FOSAD'09)*, number 5705 in Lecture Notes in Computer Science, pages 258–288. Springer, 2009.
10. E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap Space Analysis of Java Bytecode. In *6th International Symposium on Memory Management (ISMM'07)*, pages 105–116. ACM Press, 2007.
11. E. Albert, S. Genaim, and M. Gómez-Zamalloa. Live Heap Space Analysis for Languages with Garbage Collection. In *8th International Symposium on Memory management (ISMM'09)*. ACM Press, 2009.
12. D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile Resource Guarantees for Smart Devices. In *Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *Lecture Notes in Computer Science*, pages 1–27. Springer, 2005.
13. R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.
14. R. Bagnara, A. Pescetti, A. Zaccagnini, and E. Zaffanella. PURRS: Towards Computer Algebra Support for Fully Automatic Worst-Case Complexity Analysis. Technical report, 2005. `arXiv:cs/0512056` available from http://arxiv.org/.
15. Paul M. Batchelder. *An Introduction to Linear Difference Equations*. Dover Publications, 1967.
16. Amir M. Ben-Amram. Size-Change Termination with Difference Constraints. *ACM Transactions on Programming Languages and Systems*, 30(3), 2008.
17. Amir M. Ben-Amram, Neil D. Jones, and Lars Kristiansen. Linear, Polynomial or Exponential? Complexity Inference in Polynomial Time. In *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, (CiE'08)*, volume 5028 of *Lecture Notes in Computer Science*, pages 67–76. Springer, 2008.
18. R. Benzinger. Automated Higher-Order Complexity Analysis. *Theoretical Computer Science*, 318(1-2), 2004.
19. G. Bonfante, J-Y. Marion, and J-Y. Moyen. Quasi-Interpretations and Small Space Bounds. In *16th International Conference on Rewriting Techniques and Applications (RTA'05)*, volume 3467 of *Lecture Notes in Computer Science*, pages 150–164, 2005.
20. M. Braverman. Termination of Integer Linear Programs. In *18th Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 372–385. Springer, 2006.
21. A. Chander, D. Espinosa, N. Islam, P. Lee, and G. Necula. Enforcing Resource Bounds via Static Verification of Dynamic Checks. In *14th European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 311–325. Springer, 2005.
22. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *ACM Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252. ACM Press, 1977.
23. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *ACM Symposium on Principles of Programming Languages (POPL'78)*, pages 84–97. ACM Press, 1978.
24. Stephen-John Craig and Michael Leuschel. Self-Tuning Resource Aware Specialisation for Prolog. In *7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'05)*, pages 23–34. ACM Press, 2005.
25. K. Crary and S. Weirich. Resource Bound Certification. In *27 th ACM Symposium on Principles of Programming Languages (POPL'05)*, pages 184–198. ACM Press, 2000.
26. S. K. Debray and N. W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.
27. R. W. Floyd. Assigning Meanings to Programs. In *Proceedings of Symposium in Applied Mathematics*, volume 19, Mathematical Aspects of Computer Science, pages 19–32. American Mathematical Society, Providence, RI, 1967.

28. G. Gómez and Y. A. Liu. Automatic Time-Bound Analysis for a Higher-Order Language. In *Proceedings of the ACM SIGPLAN 2002 Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 75–88. ACM Press, 2002.
29. M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.
30. T. Hickey and J. Cohen. Automating Program Analysis. *Journal of the ACM*, 35(1), 1988.
31. M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *30th Symposium on Principles of Programming Languages (POPL'03)*, pages 185–197. ACM Press, 2003.
32. J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
33. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, 1993.
34. J. Komorovski. An Introduction to Partial Deduction. In *Meta Programming in Logic (META'92)*, volume 649 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 1992.
35. Lars Kristiansen and Neil D. Jones. The Flow of Data and the Complexity of Algorithms. In *1st Conference on Computability in Europe (CiE'05)*, volume 3526 of *Lecture Notes in Computer Science*, pages 263–274, 2005.
36. D. Le Metayer. ACE: An Automatic Complexity Evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, April 1988.
37. M. Leuschel. A Framework for the Integration of Partial Evaluation and Abstract Interpretation of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 26(3):413 – 463, May 2004.
38. M. Leuschel and M. Bruynooghe. Logic Program Specialisation through Partial Deduction: Control Issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.
39. J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11(3–4):217–242, 1991.
40. Beatrice Luca, Stefan Andrei, Hugh Anderson, and Siau-Cheng Khoo. Program transformation by solving recurrences. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '06)*, pages 121–129. ACM, 2006.
41. J-Y. Marion and R. Péchoux. Sup-Interpretations, a Semantic Method for Static Analysis of Program Resources. *ACM Transactions on Computational Logic*, 10(4), 2009.
42. J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *23rd International Conference on Logic Programming (ICLP'07)*, volume 4670 of *LNCS*, pages 348–363. Springer, 2007.
43. G. Necula. Proof-Carrying Code. In *ACM Symposium on Principles of programming languages (POPL 1997)*, pages 106–119. ACM Press, 1997.
44. K-H. Niggl and H. Wunderlich. Certifying Polynomial Time and Linear/Polynomial Space for Imperative Programs. *SIAM Journal on Computing*, 35(5):1122–1147, 2006.
45. 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.
46. G. Puebla and C. Ochoa. Poly-Controlled Partial Evaluation. In *8th ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'06)*, pages 261–271. ACM Press, 2006.
47. M. Rosendahl. Automatic Complexity Analysis. In *4th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*, pages 144–156. ACM Press, 1989.
48. M. Rosendahl. Simple Driving Techniques. In T. Mogensen, D. Schmidt, and I. Hal Sudborough, editors, *The Essence of Computation*, volume 2566 of *Lecture Notes in Computer Science*, pages 404–419. Springer, 2002.
49. D. Sands. A Naïve Time Analysis and its Theory of Cost Equivalence. *Journal of Logic and Computation*, 5(4), 1995.
50. Adi Shamir. A Linear Time Algorithm for Finding Minimum Cutsets in Reducible Graphs. *SIAM Journal on Computing*, 8(4):645–655, 1979.

51. Fausto Spoto, Patricia M. Hill, and Etienne Payet. Path-Length Analysis of Object-Oriented Programs. In *1st International Workshop on Emerging Applications of Abstract Interpretation (EAAI'06)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2006.
52. V. F. Turchin. The Concept of a Supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
53. P. Wadler. Strictness Analysis Aids Time Analysis. In *ACM Symposium on Principles of Programming Languages (POPL'88)*, pages 119–132. ACM Press, 1988.
54. B. Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9), 1975.