Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 248 (2009) 31–46
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Cost Relation Systems: A Language-Independent Target Language for Cost Analysis

Elvira Albert[1] Puri Arenas[1] Samir Genaim[1] Germán Puebla[2]

[1] *DSIC, Complutense University of Madrid,* {elvira,puri}@sip.ucm.es, samir@clip.dia.fi.upm.es

[2] *Technical University of Madrid,* german@clip.dia.fi.upm.es

Abstract

Cost analysis aims at obtaining information about the execution cost of programs. This paper studies *cost relation systems* (CRSs): the sets of recursive equations used in cost analysis in order to capture the execution cost of programs in terms of the size of their input arguments. We investigate the notion of CRS from a general perspective which is independent of the particular cost analysis framework. Our main contributions are: we provide a formal definition of execution cost and of CRS which is not tied to a particular programming language; we present the notion of sound CRS, i.e., which correctly approximates the cost of the corresponding program; we identify the differences with recurrence relation systems, its possible applications and the new challenges that they bring about. Our general framework is illustrated by instantiating it to cost analysis of Java bytecode, Haskell, and Prolog.

Keywords: Cost Analysis, Resource Usage, Static Analysis, Complexity.

1 Introduction

Research about *automatic cost analysis* goes back to the seminal work by Wegbreit in 1975 [22], which proposes to analyze the performance of programs by deriving *closed-form* expressions which capture their execution cost. Also, Cousot and Cousot sketch an approach to performance analysis already in their seminal 1977 paper on abstract interpretation [10]. Since then, a good number of cost analysis frameworks for a wide variety of programming languages have been devised, including for functional [22,16,18,21,19,8], logic [13,17], and imperative [1,2] programming

⋆ This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* and IST-231620 *HATS* projects, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT* and TIN-2008-05624 *DOVES* projects, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project.

doi:10.1016/j.entcs.2009.07.057

languages. An important observation of this paper is that the result of cost analysis in the different languages and programming paradigms can often be uniformly expressed as *Cost Relation Systems* (CRSs for short).

In general, given a program, cost analyzers first compute an approximation of the behaviour of the program by means of static analysis techniques. In most cases, this is done by obtaining an *abstract* version of the program by relying on abstract interpretation techniques. Essentially, the abstraction consists in inferring size relations between the arguments and replacing input arguments (numeric values, arrays, dynamic data structures, etc.) by their corresponding *sizes*. Note that the size of a piece of data is an abstraction of the actual information it contains. For example, the size of an array can be its length, whereas for a linked data structure we can take its size to be the length of the longest reference path. In addition, in order to generate CRSs, iterative constructs (loops and recursion) in the program are transformed into recursion. As a result, CRSs are sets of recursive equations which aim at capturing the cost of a program in terms of the size of its input arguments. CRSs have two features which make them interesting and powerful tools: (1) They are not limited in principle to any complexity class. Therefore, they can be used to infer cost which is polynomial, logarithmic, exponential, etc. (2) 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, bytecode instructions executed, etc.

A first objective of this paper is to characterize the notion of CRSs and motivate its use as a common target language for cost analysis. The intuitive idea is that CRSs abstract away the particular language features and are simply an instrumentation of the abstracted version of the program which allows approximating its cost. Also, we characterize the notion of a CRS being *correct*. To do this, we need to define an evaluation mechanism for CRSs. We will see that CRSs can be formally defined independently of the programming language in which the input programs to cost analysis are written. Therefore, CRSs can be considered a *lingua franca* in the sense that they can be used as the target for cost analysis of any language. Hence, we argue that progress in the study of CRSs is of interest to cost analysis of any programming language.

The second objective of the paper is to present the features and challenges that CRSs bring about. As CRSs resemble *Recurrence Relation Systems* (RRSs) in many aspects, it has been typically assumed that the output of cost analysis are simply RRSs. We clarify the differences between CRSs and RRSs and point out the limitations of existing computer algebra systems (CAS) to handle them. Finally, the usefulness of CRSs is discussed by describing its applications in the context of performance debugging and code certification. The main contributions of this paper are:

i) We provide a unified view of cost analysis which captures the main components of existing analyzers for the different programing paradigms.

ii) A formal definition of CRS is presented, together with a runtime evaluation mechanism which is independent of the language and cost model.

iii) The differences between CRS and RRS are identified, as well as the limitations that existing CAS have in order to handle them.

iv) The challenges that solving and bounding CRSs pose are described and also several applications of CRSs are sketched.

We argue that our work clarifies the notion of CRS and shows that CRSs can be used as a common language within the development of cost analyzers.

The rest of the paper is organized as follows. In Sect. 2 we present a general notion of execution cost. Sect. 3 describes the components that a cost analyzer incorporates. In Sect. 4 we motivate the language independence of CRSs by means of an example in Java bytecode, Haskell, and Prolog. Sect. 5 provides a formal definition of CRS. We highlight the differences between CRSs and RRSs in Sect. 6. A runtime evaluation mechanism of CRSs is presented in Sect. 7 together with the notion of correct CRS. In Sect. 8, we identify the challenges of solving and bounding CRSs. Finally, in Sect. 9 we conclude and review related work.

2 A General Notion of Execution Cost and Cost Model

We start by providing a general notion of *Execution Cost*, which is the feature of executions which CRSs aim at capturing. The cost of executing a program for a given input data is naturally related to the cost of the individual computation steps performed during the computation. Every programming language comes equipped with an operational semantics which describes how to perform computations. In this setting, an execution starts from an initial state s_0, and at each execution step the rules dictated by the operational semantics are used to *expand* every non-final state by computing its successor(s). A common way to rigorously represent an execution is by means of a *state transition system* (STS), which is an abstract machine that consists of a set Σ of states and a binary relation $\rightsquigarrow \subseteq \Sigma \times \Sigma$ which represents transitions between states. We use $s_i \rightsquigarrow s_j$, with $s_i, s_j \in \Sigma$, to denote that there is a transition from s_i to s_j, and we say that s_j is a *successor* of s_i. A state is *final* iff it has no successors. In many programming languages, STSs representing executions consist of only one branch. However, for some programming languages like Prolog, where multiple results for an initial call can be computed on backtracking, it is often convenient to allow STSs to be trees, i.e., some nodes may have multiple successors. Note that, under this operational semantics, Prolog is deterministic since we always compute all possible results. Given an initial call there is just one STS which represents its execution.

It is natural to relate the notion of cost to the transitions performed during execution, i.e., the arcs in the STS. Since not all transitions are necessarily equivalent from the point of view of the cost, we allow the possibility of assigning different *labels* to different transitions. With this aim, we will introduce a set of labels $\mathcal{L}$ and use *labeled transition systems*, which are state transition systems where each transition $s_i \rightsquigarrow s_j$ is marked with a label $l \in \mathcal{L}$, which we denote by $s_i \rightsquigarrow_l s_j$. Therefore, now transitions correspond to a ternary relation $\subseteq \Sigma \times \mathcal{L} \times \Sigma$. Obviously, an unlabeled transition system is equivalent to a labeled transition system with only

one label. The choice of the labels we assign to each transition is important, since it affects the observable information which we can use for obtaining the cost. As an example, in a low-level programming language such as Java bytecode (JBC for short), we can label each transition with the bytecode instruction executed during the transition. We can further refine the label by encoding in it the values of the (possibly implicit) input arguments to the instruction. In the case of Prolog, we can label transitions with (an identifier of) the clause which has been used in the corresponding resolution step. We can optionally encode the input values in the corresponding predicate call. For functional programs, we can annotate transitions with the function definition w.r.t. which the expression is reduced. We can optionally encode the input values in the corresponding function call. A *Cost Model* is a function $\mathcal{M} : \mathcal{L} \to \mathbb{Q}^+$, i.e. which assigns a positive rational number to each label. Different cost models measure different aspects of the execution. Given an STS t, we use $Labels(t)$ to denote the set of labels which appear in the transitions of t. Then the *Cost* of an execution t is defined as the cost of the corresponding labels, namely $Cost(t, \mathcal{M}) = \sum_{l \in Labels(t)} \mathcal{M}(l)$.

Different classes of labels and associated cost models can be used to measure the use of different resources of interest. For instance, the Java bytecode cost analyzer of [2] can be used for observing, among other things, the number of execution steps performed, the amount of heap allocated during execution, and the number of calls to certain relevant methods. In particular, a cost model which counts the number of execution steps can be defined as $\mathcal{M}_{\mathsf{ninst}}(l) = 1$ for any l. The cost model $\mathcal{M}_{\mathsf{heap}}$, which counts the number of bytes allocated on the heap has been defined in [4], where the cost model returns zero for all bytecode instructions which do not allocate memory in the heap and returns the corresponding number of bytes for those instructions which actually allocate heap space.

In cost analysis, new cost models can be directly plugged in by just providing the corresponding definition and CRSs for the provided model can be inferred by the tools, usually without any modification in the analysis engine. In our examples, and in order to keep the presentation simple, we use a cost model which counts the number of execution of steps: for JBC it counts the number of bytecode instructions executed; for Prolog it counts resolution steps; and for functional programs it counts reduction (rewriting) steps.

The direct application of this notion of execution cost is in principle possible for deterministic programming languages, provided that the execution terminates and involves the following steps: (1) given an initial state, produce its corresponding STS t, (2) collect the set of labels in all transitions in t, (3) apply the cost model to each label, and (4) obtain the final result by adding up such figures. From a practical point of view, it is often better to interleave these phases so that the cost is accumulated while executing the program. This can be done by instrumenting the program with additional arguments which accumulate the cost or by instrumenting the operational semantics. Both approaches share the disadvantage that they require running the program in order to compute its costs for each input value of interest. On the other hand, for a given initial state s_0, static approaches aim

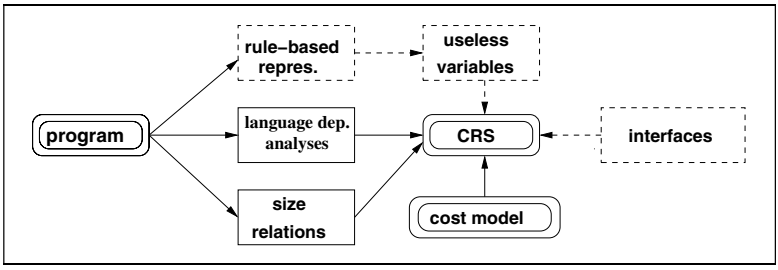

Fig. 1. A general view of cost analysis

at approximating $Cost(t, \mathcal{M})$ s.t. t corresponds to the execution which starts from s_0, but without constructing t, i.e., they allow approximating the cost of a program for some input data *without* having to actually run the program for such data and thus avoiding such overhead. Cost analysis, and within it CRSs, fall into the second approach.

In deterministic languages, given an initial state s_0, there is a unique STS which corresponds to the execution. However, in languages where a non-deterministic choice is possible, an initial state may lead to several possible STSs. This, in fact, is the case for most realistic programming languages, as they provide constructs for random number generation and/or access to environment variables such as date/time. In order to accommodate for truly non-deterministic programming languages, from now on we consider that given an initial state s_0 there is a set of different executions, with their corresponding STSs, which can be built from s_0. We will refer to the set of all possible STSs for s_0 as $\mathsf{Executions}(s_0)$. In deterministic executions, $\mathsf{Executions}(s_0)$ is a singleton.

3 A Unified View of Cost Analysis

Fig. 1 provides a unified view of cost analysis with the main components that it incorporates in order to compute CRSs for different programming paradigms. Within double frames, we show that the analysis receives as input a program and the selected cost model and yields as output a CRS. The analyzer can have a set of predefined cost models and, in some cases, users can define their own cost model [17]. We now describe the main components in more detail.

Rule-based representation. On the top left side of the figure, we see that the incoming program is often transformed into an intermediate *rule-based representation*. The main purpose of this step is to detect loops in the program and represent them by means of recursive rules in order to facilitate the subsequent CRS generation phase. This step can be easily done from the Control Flow Graph (CFG for short) of the program by associating a rule to each basic block. When cost analysis is on a low-level language (see, e.g., [2]), having a rule-based form makes it possible to represent the unstructured control flow of the bytecode into a procedural form (e.g., `goto` statements are transformed into recursion). Naturally, cost analyses of declarative languages do not need this transformation as they are natively in recursive form.

Size analysis. Obtaining *size-relations* between the states at different program points is indispensable for setting up cost relations. These sizes describe how the data change when the program goes through its loops. For this purpose, the notion of *size measure* is crucial. In general, various measures can be used to determine the size of data. For instance, in symbolic languages (see, e.g., [13]), term-depth, term-size and list-length are used as size measures. In object-oriented languages, two size measures have been used. For values which are of integer type, we can take their actual value as their size. For values which are references, their *path-length* [20] can be used as their size. The path length of a reference is defined as the length of the longest reference path reachable from it. A wide range of size analyses exists which compute useful size approximations for different programming languages.

Language dependent analyses. The computational nature of a programming language might require some additional static analyses in order to generate useful and sound CRSs. This is the case in Logic Programing, where additional information, such as determinacy and non-failure, is required in order to obtain CRSs that accurately and correctly approximate the corresponding cost, since such analyses provide valuable information on the shape of the execution tree (see, e.g., [12]). Similar analyses are required in cost analysis of functional programming with lazy evaluation. These problems usually do not occur in imperative languages.

Useless variables. Ideally, cost analyzers are interested in obtaining CRSs where only the program variables and arguments which affect the cost appear as arguments in the equations. The program variables which may have an impact on the cost of a program are those that may affect directly or indirectly the conditional statements (i.e., they can affect the control flow of the program), and those that may be encoded in transition labels (as discussed in Sect. 2 above). The elimination of useless variables can be done by applying well-known *slicing* techniques (see, e.g., [3]).

Interfaces. In order to analyze realistic programs, it is essential to have a modular design which allows handling *external* methods during analysis. By external methods, we mean code which is not accessible to the analyzer, including native libraries written in a different language (and thus not analyzable), methods which are not yet implemented, etc. Cost analyzers can support a modular design by means of *interfaces* which store the required information about external methods. As customary in modular analysis, the information learned from the interfaces is used during the analysis much in the same way as the information inferred by the analyzer itself.

CRS. Once the previous phases have their corresponding information available, the analyzer can set up a CRS for the input program and the selected cost model. Essentially, the recursive representation of the program determines the structure of the CRS: for each equation in the rule-based representation, the cost relation has a corresponding recursive equation. The arguments of the CRS denote the size of the corresponding program variable. Size relations approximate (1) the applicability conditions of each cost equation and (2) how data sizes increase or decrease over the equation. The cost model is applied to define the cost of each block of code that

Prolog	Java (recursive)

merge(This,[ ],R):- !,R = This.
merge([D|Next],[OD|ONext],R):-
D>OD, !, R = [OD|T],
merge([D|Next],ONext,T).
merge([D],O,R):- !,R = [D|O].
merge([D|Next],O,R):-
R = [D|T], merge(Next,O,T).
	public class MLRec {
	private int d;
	private MLRec next;
	public MLRec(int d, MLRec next){
	this.d = d;
	this.next = next;
	}
	public MLRec merge(MLRec o) {
	if (o == null) return this; $(1)_m$
	else if (d>o.d)
	return new MLRec(o.d,merge(o.next)); $(2)_m$
	else if (next = null)
	return new MLRec(d,o); $(3)_m$
	else return new MLRec(d,next.merge(o)); $(4)_m$
	}

Haskell	
merge this [ ] = this merge (d:next) o = if (d> od) then (od: merge (d:next) onext) else if (next==[ ]) then (d:o) else (d:merge next o) where (od:onext) = o	

Fig. 2. Prolog, Haskell and Java implementations of recursive merge

$(1)_m$ $merge(this, o)=k_1$	$\{this \geq 1, o=0\}$
$(2)_m$ $merge(this, o)=k_3+merge(this, o')$	$\{this \geq 1, o \geq 1, o>o', o' \geq 0\}$
$(3)_m$ $merge(this, o)=k_2$	$\{this \geq 1, o \geq 1\}$
$(4)_m$ $merge(this, o)=k_4+merge(this', o)$	$\{this>this', this \geq 2, this' \geq 1, o \geq 1\}$
$\mathcal{M}_{\text{ninst}}$	$k_1=4, k_2=26, k_3=26, k_4=29$

Fig. 3. Structure of the CRS for recursive merge (capturing different implementations)

a cost equation comprises. As a result of this process, a CRS is an instrumented version of the abstracted program aimed at observing its execution cost according to the cost model of interest.

4 Language-Independence of CRSs by Example

By means of a simple example, we illustrate the above components of cost analysis and motivate the notion of CRS as a language-independent target language for cost analysis. We consider the three implementations in Fig. 2 of a method merge which merges two sorted lists. To the right, the implementation in Java merges the list o received as input parameter and the object this and outputs the result in a new list. To the left, we show two implementations, one in Prolog and one in Haskell. In both cases, the lists to be merged are shown as explicit input parameters. The three implementations have the same precondition, which is that the first argument (this in the Java version) is non null.

The CRS depicted in Fig. 3 models a possible output of cost analysis for any of the three implementations of merge shown in Fig. 2, since they have similar operational behaviour. In particular, the CRS shown has been automatically inferred from the bytecode associated to the Java version by using the analyzer of [2]. [1] Depending on the programming language and on the cost model used, the constants

[1] For readability, the CRS is presented after simplifying it by performing *partial evaluation*, which replaces calls by their definitions.

$k_1, \ldots, k_4$ take different values. The values we show at the bottom of the figure correspond to the $\mathcal{M}_{\mathsf{ninst}}$ cost model (see Sect. 2 above).

The purpose of this section is to illustrate the main steps involved in the generation of the CRS from the three implementations. The first important point to note is that, as explained in Sect. 3, the CRS should match the structure of the program such that when the program contains a loop construct, its CRS has a recursion. From the declarative implementations, it can be directly seen that each program rule (or clause) leads to an associated cost equation. In the imperative program, this step requires to go through some form of intermediate, rule-based representation which makes the correspondences between the program constructs and the cost equations explicit. Fig. 4 depicts the CFG for the Java code of merge. One rule (or equation) is obtained for each of the execution paths in the graph. Eq. $(1)_m$ captures the cost of the trace $merge_1$-$merge_3$ when the list o is null (see the Java program). Eq. $(3)_m$ captures the cost of the trace $merge_1$-$merge_2$-$merge_4$-$merge_6$, which occurs when the list this has only one element. Eq. $(2)_m$ captures the cost of the trace $merge_1$-$merge_2$-$merge_5$, which corresponds to the first recursive call. Finally, Eq. $(4)_m$ captures the cost of trace $merge_1$-$merge_2$-$merge_4$-$merge_7$ which corresponds to the second recursive call.

The second important point is that data structures in the program are abstracted to their sizes in the CRS. For instance, in the recursive call of Eq. $(2)_m$, it is ensured that the size of o has decreased ($o{>}o'$), but we do not know how much. This is because the size analysis for pointer based data structures used in [2] is based on path-length analysis [14], where size-relations are expressed using only $>$ and $\geq$. More precise size analysis in logic and functional programming could infer the precise relation, i.e., $o'{=}o{-}1$. Size relations also contain applicability conditions (i.e., *guards*) for the different equations, if any, by providing constraints which only affect (a subset of) the variables in the lhs. Among them, we have e.g. $o{=}0$ which requires that the list o is null.

The third important point is that the condition d>o.d in the program does not appear in the size relation of equation $(2)_m$ (resp. d$\leq$o.d in $(3)_m$ and $(4)_m$). This is because this condition is not observable in our CRS, as the lists this and o have been abstracted to their length, and hence the values in this.d and o.d are unknown. This is indeed the case in the three different languages.

The final conclusion is that, regardless of the language in which the program is written, cost analysis produces a cost relation system in the same form. This observation has motivated us to formalize the notion of CRS, its evaluation mechanism, its main properties and challenges, which are the subject of the remaining of the paper.

5 Cost Relation Systems

This section presents formally the notion of cost relation system. We use x, y, z, possibly subscribed, to denote variables, v, w denote integer values from $\mathbb{Z}$, a, b natural numbers from $\mathbb{N}$, and q rational numbers from $\mathbb{Q}$. We use $\overline{x}$ to denote

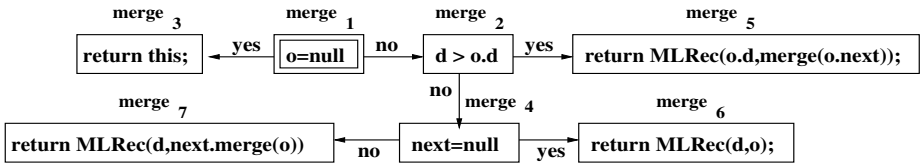

Fig. 4. Control flow graph for method merge

sequence of variables $x_1, \ldots, x_n$, for some $n>0$. Similarly, $\overline{v}$ denotes a sequence of integer values. For simplicity, we sometimes interpret these sequences as sets. Given a sequence $\overline{x}$, we say that an *assignment* for $\overline{x}$ is a sequence $\overline{v}$ of integer values (the actual assignment is denoted $[\overline{x}/\overline{v}]$). Given any entity A, $A[\overline{x}/\overline{v}]$ stands for the result of replacing in A each occurrence of variable x_i by v_i. Also, we use $vars(A)$ to refer to the set of variables occurring in A. A *linear expression* has the form $q_0+q_1x_1+\cdots+q_nx_n$. A *linear constraint* has the form l_1 *op* l_2 where l_1 and l_2 are linear expressions and $op \in \{=, \leq, <, >, \geq\}$. A set of linear constraints is used to represent the conjunction of the corresponding constraints. *Size relations* are sets of linear constraints. We now define the notion of *cost expression*, which syntactically characterizes the kind of expressions which CRSs contain.

Definition 5.1 [cost expression] A *cost expression* exp is a symbolic expression of the form:

$$\texttt{exp} ::= q \mid x^q \mid \texttt{exp}\ op\ \texttt{exp} \mid \texttt{exp}^{\texttt{exp}} \mid \log_{\texttt{a}}(\texttt{exp}) \mid \max(S) \mid \min(S)$$

where $op \in \{+, -, /, *\}$ and S is a non empty set of cost expressions.

Cost expressions are the basic elements of CRSs. They are used to indicate the resources we are accumulating; they are also used to represent the bounds of CRSs. As CRSs can be used to capture any complexity class, cost expressions must cover polynomial, logarithmic and exponential expressions and we must be able to bound its solution (by using functions max and min). We now present a language-independent definition of the notion of CRS.

Definition 5.2 A *cost relation system* $\mathcal{S}$ is a set of cost equations of the form $\langle C(\overline{x})=\texttt{exp}+\sum_{i=1}^{k} D_i(\overline{y}_i), \varphi\rangle$ with $k\geq 0$ where

1. All variables $\overline{x}$, $vars(\texttt{exp})$ and $\overline{y}_i$ are distinct variables,
2. exp is a cost expression,
3. φ is a size relation between the variables $\overline{x} \cup vars(\texttt{exp}) \bigcup_{i=1}^{k} \overline{y}_i$.

The CRS depicted in Fig. 3 contains simple cost equations with only one recursive call. Also, all expressions exp are constants, as we are giving the same constant value "1" to the cost of steps. This is clearly not the case in all cost models. For instance, if we measure heap consumption, the operation of creating an array of integers costs $4*n$ heap cells, where n is the length of the array and 4 is the number of bytes required to represent an integer.

Example 5.3 Let us illustrate the use of non-constant expressions. We include the

following method insertSort which sorts the input list by using the previous method merge. The CRS appears to the right. We can observe in $(3)_s$ the call to the cost of merge in which the first parameter is always a list of one element. Also, the size relations appear attached to the equations describing the relations for the variables in the head of the rules and between the head and the body. As expected, the size of the list l decreases over the loop.

public static MLRec insertSort(MLRec l){	
MLRec acu=null;	$(1)_s$ $insertSort(l)=4+loop(l,0)$ $\{l\geq 0\}$
while (l!=null) {	$(2)_s$ $loop(l,acu)=2$ $\{l=0, acu\geq 0\}$
MLRec node = new MLRec(l.d,null);	$(3)_s$ $loop(l,acu)=26+merge(1,acu)+$
acu=node.merge(acu);	$loop(l',acu')$
l=l.next;}	$\{l>l', l'\geq 0, acu\geq 0, acu'>acu\}$
return acu;}	

We can safely assume the following upper bound (see Def. 8.1) for the cost of merge: $merge(l_1,l_2) = 26+29*(l_1+l_2)$. In this case the equation $(3)_s$ takes the form: $loop(l,acu)=52+29*(1+acu)+loop(l',acu')$. We can observe here the use of non-constant cost expressions and, in particular, that they must cover the complexity classes that the CRS can be bounded to. □

6 CRSs vs. Recurrence Relation Systems

CRSs have several important features which are a consequence of being obtained by automatic program analysis, and which are not present in traditional Recurrence Relation Systems (RRSs):

Non-deterministic relations. In contrast to RRSs, cost equations for the same relation do not need to be mutually exclusive. The reason for allowing this is because cost analysis needs to use *size abstractions*. Unavoidably, the use of abstraction introduces a loss of precision: some guards which make the execution of the original program deterministic may not be observable when using the size of arguments instead of their actual values. In our example, this happens with the guard d>o.d between equations $(2)_m$ and $(4)_m$ as it talks about the concrete values of the elements in the list, which are clearly not observable as the lists have been abstracted to their length.

Inexact size relations. CRSs can have size relations which contain inequality constraints. This is essential in order to handle cost relations automatically obtained from the analysis of realistic programs with complex data structures, for which size analysis may lose precision. For instance, analysis may be able to infer that a given data structure strictly decreases in size from one iteration to another, but it may be unable to provide the precise reduction. This happens in our example in equations $(2)_m$, $(4)_m$, $(3)_s$, where we only know that $o>o'$, $this>this'$, $l>l'$.

Multiple arguments. Cost relations can have several arguments that may increase or decrease at each iteration. Importantly, the number of times a given relation is executed can depend, or be a combination of, several of its arguments. For

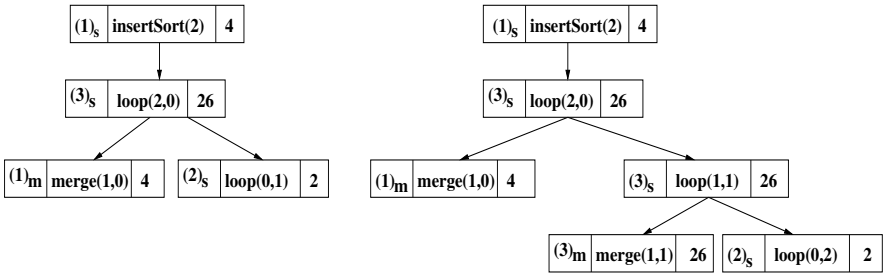

Fig. 5. Two evaluation trees for *insertSort*(2)

instance, function *merge* is executed $min(this, o)$ times. In contrast, most RRS solvers assume that the number of times a function is executed only depends on one argument (often a decreasing variable).

As a result of the first two points above, CRSs are not required to define functions, but rather relations, in the sense that, given input values $\overline{v}$, there may exist multiple results for $C(\overline{v})$. This raises the questions: is it practical to evaluate a CRS at runtime, i.e., does it make sense to run an instrumentation of the abstracted program? Are existing CAS (Maple, Mathematica, etc.) sufficient to solve CRSs, as it had been assumed typically by the cost analysis community? The next sections address these issues and the challenges that CRSs bring about.

7 Evaluation of CRS

We now provide a formal semantics for CRSs. This semantics is in terms of calls and answers. Calls are of the form $C(\overline{v})$, where C is a cost relation and $\overline{v}$ are integer values. A call $C(\overline{v})$ is evaluated in $\mathcal{S}$ by repeatedly replacing calls to relations by appropriate instantiations of the rhs of applicable equations until a cost expression (i.e., call-free) is reached. This *evaluated* expression consists of the sum of a series of cost expressions. Note that CRSs are potentially non-deterministic, which means that there may be different ways of evaluating a call and which result in different *answers* to the call. The process of obtaining an answer can be represented graphically using trees defined as (possibly nested) terms of the form $node(Call, Local\_Cost, Children)$.

Example 7.1 Fig. 5 depicts two evaluation trees for the call *insertSort*(2). We can observe that each node in the tree contains a call (middle box) and its local cost (right box) and it is linked by arrows to its children. In the figure, for clarity, each call is annotated with a number (left box) which indicates the equation which was selected for evaluating the corresponding call. The leftmost tree is an evaluation tree of minimal cost, which results in 36 cost units, and the rightmost one is an evaluation tree of maximal cost, which results in 88 cost units. □

Definition 7.2 [evaluation tree] Given a CRS $\mathcal{S}$ and a call $C(\overline{v})$, an *evaluation tree* of $C(\overline{v})$ in $\mathcal{S}$, denoted $\mathsf{Tree}(C(\overline{v}), \mathcal{S})$, is $node(C(\overline{v}), e, \langle t_1, \ldots, t_k \rangle)$, where:

(1) there is a renamed apart equation $\langle C(\overline{x}) = \mathtt{exp} + \sum_{i=0}^{k} D_i(\overline{y}_i),\ \varphi \rangle \in \mathcal{S}$ s.t. φ' is

satisfiable in $\mathbb{Z}$, where $\varphi'=\varphi[\overline{x}/\overline{v}]$, and

(2) there exist assignments $\overline{w}, \overline{v}_i$ for $vars(\texttt{exp}), \overline{y}_i$ respectively s.t. $\varphi'[vars(\texttt{exp})/\overline{w}, \overline{y}_i/\overline{v}_i]$ is satisfiable in $\mathbb{Z}$, and

(3) $e=\texttt{exp}[vars(\texttt{exp})/\overline{w}]$, t_i is an evaluation tree $\mathsf{Tree}(D_i(\overline{v}_i), \mathcal{S})$ with $i=0,\ldots,k$.

In step 1 we look for an equation $\mathcal{E}$ which is applicable for solving $C(\overline{v})$. Note that there may be several equations which are applicable. In step 2 we look for assignments for the variables in the rhs of $\mathcal{E}$ which are compatible with the size relations associated to $\mathcal{E}$. This is another non-deterministic step as there may be (infinitely many) different assignments which satisfy all size relations. Finally, in step 3 we apply the assignment to the expression `exp` and continue recursively evaluating the call.

Example 7.3 In the CRS of Fig. 3, whenever equation $(4)_m$ is applicable, equation $(2)_m$ is also applicable, since $this \geq 2$ implies $this \geq 1$. Also note, that in the recursive call to *loop* we are allowed to pick any values l', acu' such that $l'<l$, $acu'>acu$. The rightmost tree in Fig. 5 corresponds to the maximal real cost, where we assign $l'=l-1$ and $acu'=acu+1$ in the recursive call. This is what happens in actual executions of the program. In the rightmost tree we assign $l'=l-2$ and $acu'=acu+1$ in the recursive call to *loop*, this results in a minimal approximation, however, it does not correspond to any actual execution. This is a side effect of the safe approximations computed by static analysis: it allows obtaining correct information, but it may be imprecise sometimes. It is interesting to observe that we can compute an infinite number of evaluation trees, as the instantiation step 2 can give an infinite number of assignments to variable acu' satisfying the condition $acu'>acu$. □

Since multiple evaluation trees can be obtained for a given call, we use the notation $Trees(C(\overline{v}), \mathcal{S})$ to refer to the set of all evaluation trees for $C(\overline{v})$ in $\mathcal{S}$. Then, we can define $answers(C(\overline{v}), \mathcal{S})=\{\mathsf{Sum}(t) \mid t \in Trees(C(\overline{v}), \mathcal{S})\}$, where $\mathsf{Sum}(t)$ traverses all nodes in t and computes the sum of the cost expressions in them. The following definition presents the notion of *correct* CRS.

Definition 7.4 [correct CRS] Let m be a method with n input parameters and $\mathcal{M}$ a cost model. A CRS $\mathcal{S}$ is *correct* for m and $\mathcal{M}$ iff for any $\overline{v}\in\mathbb{Z}^n$ and a corresponding initial state s_0, we have $Cost(t, \mathcal{M}) \in answers(C_m(\overline{v}), \mathcal{S})$ for any $t \in \mathsf{Executions}(s_0)$.

Intuitively, a CRS is correct if it safely approximates the actual execution cost in the sense that such cost must be a possible solution to the equations. In a given cost analysis framework, the correctness of the CRS is usually entailed from the correctness of the particular size analysis used in the framework.

8 Solving CRSs: Overview of Existing Tools

In the previous section, we have seen that due to the recursive nature of CRSs, directly evaluating calls to a relation defined in a CRS requires to perform an iterative computation following the recursive calls. As there are often multiple

(possibly infinite) possible evaluations of a call, trying to obtain a solution (or an upper bound for it) in this way is impractical. Besides, it can difficult to glean immediate information about the cost of a program by looking at its associated CRS, especially when there are multiple relations involved. As a result, CRSs have rather limited applicability unless *closed form* (i.e., non-recursive) solutions or bounds for them are obtained.

An important point to note is that the actual cost of executing the program for a given goal is necessarily a solution of the CRS. In principle, this makes CRSs valid tools for computing upper and lower bounds of the cost. We start by recalling the definition of *upper bound* of relations.

Definition 8.1 [upper bound] Let $\langle P, \leq \rangle$ be a partially ordered set, and let S be a subset of P. A value $a \in P$ is an *upper bound* of S iff $\forall s \in S$ we have that $s \leq a$. Also, let C be a relation over $\mathbb{Z}^n \times \mathbb{R}$. A function $U : \mathbb{Z}^n \to \mathbb{R}$ is an *upper bound* of C iff $\forall \overline{v} \in \mathbb{Z}^n$ we have that $U(\overline{v})$ is an upper bound of $answers(C(\overline{v}), \mathcal{S})$.

Most of existing analyzers try to use computer algebra systems (CAS) for solving RRSs, like Mathematica, Maple, Maxima, etc. Essentially, there are two ways in which cost analyzers use them 1) directly trying to apply them to solve CRSs (if they have the form of RRSs) or 2) converting CRSs to RRSs.

In our experience with the analyzer in [2], applying directly CAS on CRSs is not a realistic choice. The main problem is that CAS admit only a form of RRSs which does not cover the essential features of CRSs that distinguish them from RRSs, as pointed out in Sect. 6. This happens even for our simple running example; there exists no RRS solver that we can use directly. With this, we do not mean to say that existing CAS are not powerful. In fact, from an algebraic perspective, if we ignore the additional features that CRSs have, cost relations would be casted as a simple class of recurrence equations. Indeed, the recurrence equations solved by CAS can present a much more complex structure. For instance, they support equations with coefficients to function calls which can be polynomials. However, this power is not really needed for the equations in CRSs, since the equations generated from cost analysis of programs are not usually in such complicated form, as their structure is obtained from the structure of the program. On the other hand, existing CAS fall short in order to attack some of the features of CRSs which are not present in RRSs.

The second approach is to obtain closed form *upper bounds* by converting CRSs into RRSs and then using CAS. This requires, among other things, removing non-determinacy while preserving the worst-case solution. For this, we need to remove equations from the CRS as well as sometimes to replace inexact size relations by exact ones. This transformation would be easy to do in our example, as we can take the most expensive recursive case $(4)_m$ and the most expensive base case $(3)_m$. However, neither of these transformations can be safely done in all cases. In particular, this is not possible when the maximum cost might be a result of interleaving between the different equations. For instance, if we are interested in obtaining an upper bound solution, there are cases, where if we remove any of the equations in

the CRS in order to obtain determinacy, we no longer obtain the worst case and the resulting closed form is not guaranteed to correspond to a correct upper bound. Therefore, though this approach can be applied in simple cases, is not a sound alternative either.

A CRS solver which computes solutions or upper/lower bounds for CRS output by automatic cost analysis must be able to:

(i) Bound the number of iterations of the cost relation when the (maximum) number of times that a given relation is executed can be a combination of several of its arguments.

(ii) Handle inequalities in a general way. Previous systems either cannot handle them or they only allow inequalities which relate variables to constant terms, but not inequalities between variables (e.g., $this > this'$).

(iii) Provide a general and sound way to compute the maximum (or minimum) over non-deterministic equations. This is complicated in general as the maximum (or minimum) cost might be a result of interleaving between the different equations.

We are only aware of three tools which aim at solving or bounding CRSs. One of the first existing systems was CASLOG [13]. It is though limited to rather simple CRS and in particular it cannot handle the first two features above. Another relevant work in this direction is PURRS [7], which has been the first system to provide, in a fully automatic way, non-asymptotic upper and lower bounds for a wide class of recurrences. Unfortunately, it requires CRSs to be deterministic (item 3 above) and does not handle inequalities (item 2). The PUBS system [5] has been recently developed (it is available from `http://www.cliplab.org/Systems/PUBS`). It is an important step in this direction as, in principle, it is able to handle the three items above. We believe that the recent development of automatic solving tools will be of importance for the practical use of cost analysis.

9 Discussion and Related Work

We have presented a general notion of cost and a unified view of cost analysis which includes the main components of state-of-the-art resource usage analyzers. In this context, we have motivated and formalized the notion of CRSs as a language-independent means to target the analysis output. There is an important point which has remained unclear in the area of cost analysis, and which is a contribution of this paper, which is to state the differences between CRSs and RRSs. Indeed with the development of advanced CAS (such as Mathematica®, MAXIMA, MAPLE, etc.), recent cost analyses [8,15] try to use them. This paper has clarified the differences between CRSs and RRSs. The important consequence of such differences is that existing CAS do not cover the distinguishing aspects of CRSs and there is a need to develop practical solvers which are able to directly handle them, as we have done in [5] as a first attempt to solve this problem.

To conclude, note that given a solution for a CRS, or an upper/lower bound

approximation for it, CRSs bring about very interesting applications in the following fields:

- *Resource Bound Certification [11,6,9].* It proposes the use of safety properties involving cost requirements, i.e., that the untrusted code adheres to specific bounds on resource consumption. CRSs enable to express arbitrary resource bounds certificates. Approaches based on type systems are usually restricted to polynomial bounds [11,6]. An example of this application is *mobile code*, where the *code consumer* receives code to be executed. The receiver of the code may want to infer cost information in order to reject code which has too large cost requirements in terms of computing resources (in time and/or space).
- *Performance Debugging and Validation.* This is a direct application of cost analysis, where the analyzer tries to verify or falsify assertions about the efficiency of the program which are written by the programmer. The role of CRSs is essential in order to infer the performance, either as an exact solution or upper bound.
- *Granularity Control [13].* Parallel computers with multicore processors are currently becoming mainstream. In parallel systems, knowledge about the cost of different procedures can be used in order to guide the partitioning, allocation and scheduling of parallel processes.

References

[1] A. Adachi, T. Kasai, and E. Moriya. A theoretical study of the time analysis of programs. In J. Becvár, editor, *MFCS*, volume 74 of *Lecture Notes in Computer Science*, pages 201–207. Springer, 1979.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In Rocco De Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, March 2007.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Removing Useless Variables in Cost Analysis of Java Bytecode. In *ACM Symposium on Applied Computing (SAC) - Software Verification Track (SV08)*, pages 368–375, Fortaleza, Brasil, March 2008. ACM Press, New York.

[4] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap Space Analysis for Java Bytecode. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 105–116, New York, NY, USA, October 2007. ACM Press.

[5] Elvira Albert, Puri Arenas, Samir Genaim, and German Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In María Alpuente and Germán Vidal, editors, *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 15-17, 2008, Proceedings*, volume 5079 of *Lecture Notes in Computer Science*, pages 221–237. Springer-Verlag, July 2008.

[6] D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile Resource Guarantees for Smart Devices. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Proc. of Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS)*, volume 3362 of *LNCS*, pages 1–27. Springer, 2005.

[7] 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/`.

[8] R. Benzinger. Automated Higher-Order Complexity Analysis. *Theor. Comput. Sci.*, 318(1-2), 2004.

[9] Ajay Chander, David Espinosa, Nayeem Islam, Peter Lee, and George C. Necula. Enforcing resource bounds via static verification of dynamic checks. In *European Symposium on Programming (ESOP)*, number 3444 in LNCS, pages 311–325. Springer-Verlag, 2005.

[10] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[11] K. Crary and S. Weirich. Resource bound certification. In *POPL'00*. ACM Press, 2000.

[12] S. K. Debray and N.-W. Lin. Automatic complexity analysis for logic programs. In *Eighth International Conference on Logic Programming*, pages 599–613, Paris, France, June (1991). MIT Press.

[13] 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.

[14] Patricia M. Hill, Etienne Payet, and Fausto Spoto. Path-length analysis of object-oriented programs. In *Proc. International Workshop on Emerging Applications of Abstract Interpretation (EAAI)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2006.

[15] J. Jouannaud and W. Xu. Automatic Complexity Analysis for Programs Extracted from Coq Proof. *ENTCS*, 2006.

[16] D. Le Metayer. ACE: An Automatic Complexity Evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, April 1988.

[17] J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *International Conference on Logic Programming (ICLP)*, volume 4670 of *LNCS*, pages 348–363. Springer-Verlag, September 2007.

[18] M. Rosendahl. Automatic Complexity Analysis. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, pages 144–156. ACM, New York, 1989.

[19] D. Sands. A naïve time analysis and its theory of cost equivalence. *J. Log. Comput.*, 5(4), 1995.

[20] F. Spoto, P. M. Hill, and E. Payet. Path-length analysis for object-oriented programs. In *Proc. International Workshop on Emerging Applications of Abstract Interpretation (EAAI)*, 2006.

[21] P. Wadler. Strictness analysis aids time analysis. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, pages 119–132. ACM Press, 1988.

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