TLP **16** (5–6): 849–865, 2016. © Cambridge University Press 2016
doi:10.1017/S1471068416000442

*A general framework for static profiling of parametric resource usage**

P. LOPEZ-GARCIA[1,2], M. KLEMEN[1], U. LIQAT[1]
and M. V. HERMENEGILDO[1,3]

[1]*IMDEA Software Institute*
(*e-mail:* {pedro.lopez, maximiliano.klemen, umer.liqat, manuel.hermenegildo}@imdea.org)
[2]*Spanish Council for Scientific Research (CSIC)*
[3]*Technical University of Madrid (UPM)*

submitted 6 May 2016; revised 8 July 2016; accepted 22 August 2016

Abstract

For some applications, standard resource analyses do not provide the information required. Such analyses estimate the *total* resource usage of a program (without executing it) as functions on input data sizes. However, some applications require knowing how such total resource usage is *distributed* over selected parts of a program. We propose a novel, general, and flexible framework for setting up cost equations/relations which can be instantiated for performing a wide range of resource usage analyses, including both *static profiling* and the inference of the standard notion of cost. We extend and generalize standard resource analysis techniques, so that the relations generated include additional Boolean control variables for switching on or off different terms in the relations, as required by the desired resource usage profile. We also instantiate our framework to perform *static profiling of accumulated cost* (also parameterized by input data sizes). Such information is much more useful to the software developer than the standard notion of cost: it identifies the parts of the program that have the greatest impact on the total program cost, and which therefore should be optimized first. We also report on an implementation of our framework within the CiaoPP system, and its instantiation for accumulated cost, and provide some experimental results. In addition to generality, our new method brings important advantages over our previous approach based on a program transformation, including support for non-deterministic programs, better and easier integration in the compiler, and higher efficiency.

KEYWORDS: Static Profiling, Static Analysis, Resource Usage Analysis, Complexity Analysis

1 Introduction

Resources are numerical properties about the execution of a program, such as number of resolution steps, execution time, energy consumption, number of calls to a particular predicate, number of network accesses, number of transactions in a

* This research has received funding from EU FP7 agreement no 318337 *ENTRA*, Spanish MINECO TIN2012-39391 *StrongSoft* and TIN2015-67522-C3-1-R *TRACES* projects, and the Madrid M141047003 *N-GREENS* program. Special thanks are due to John Gallagher for many fruitful and inspiring discussions and to the anonymous reviewers for their detailed and useful comments.

https://doi.org/10.1017/S1471068416000442 Published online by Cambridge University Press

database, etc. The goal of automatic static cost analysis is estimating the resource usage of the execution of a program without running it, as a function of input data sizes and possibly other (environmental) parameters. The significant body of work on static analysis for logic programs has actually also been applied to the analysis of other programming paradigms, including imperative programs. This is achieved via a transformation of the program into *Horn Clauses* (Méndez-Lojo *et al.* 2007). In this paper we concentrate on the analysis of Horn Clause programs, independently of whether they are the result of a translation or the actual program source.

Given a program $\mathscr{P}$ and a predicate $\mathtt{p} \in \mathscr{P}$ of arity k and a set Π of k-tuples of actual arguments to p, we refer to the *standard cost* of a call $\mathtt{p}(\bar{e})$ (i.e., a call to p with actual data $\bar{e} \in \Pi$), as the resource usage (under a given cost metric) of the complete execution of $\mathtt{p}(\bar{e})$. Thus, the *standard cost* is a per-call cost formalized as a function $\mathscr{C}_{\mathtt{p}} : \Pi \rightarrow \mathscr{R}_{\infty}$, where $\mathscr{R}_{\infty}$ is the set of real numbers augmented with the special symbol ∞ (which is used to represent non-termination). *Standard cost*, and, in general, resource usage information, is very useful for a number of applications, such as automatic program optimization, verification of resource-related specifications, detection of performance bugs, or helping developers make resource-related design decisions. In the latter case, the analysis has to show which parts of the program are the most resource-consuming, i.e., which predicates would bring the highest overall improvement if they were optimized, so that programming efforts can be focused more productively. The standard cost information only partially meets these objectives. For example, often predicates with the highest (standard) cost are not the ones whose optimization is most profitable, since predicates which have lower costs but which are called more often may be responsible for a larger part of the overall resource usage. The input data sizes to such calls are also relevant. Thus, rather than the global costs provided by standard cost analyses, what is really needed in many such applications is the results of a *static profiling* of the program that helps identify the parts of a program responsible for highest fractions of the cost, or, more generally, how the total resource usage of the execution of a program is *distributed* over selected parts of it. By *static profiling* we mean the static inference of the kinds of information that are usually obtained at run-time by profilers.

For this reason, herein we are more interested in what we refer to as *accumulated cost*. To give an intuition of this concept, we first explain our notion of *cost centers*, which is similar to the one we use in (Haemmerlé *et al.* 2016), and was inspired from (Sansom and Jones 1995; Morgan and Jarvis 1998): they are user-defined program points (predicates, in our case) to which execution costs are assigned during the execution of a program. Data about computational events is accumulated by the cost center each time the corresponding program point is reached by the program execution control flow. Assume for example that predicate p calls another predicate q (either directly or indirectly), and that we declare that both predicates are cost centers. In this case, the cost of a (single) call $\mathtt{p}(\bar{e})$ *accumulated in* cost center q, denoted $\mathscr{C}^{\mathtt{q}}_{\mathtt{p}}(\bar{e})$, expresses how much of the standard cost of $\mathtt{p}(\bar{e})$ is attributed to q, and is the sum of the costs of the computations performed "under the scope" of all the calls to q generated during the complete execution of $\mathtt{p}(\bar{e})$. We say that a computation is "under the scope" of a call to cost center q if the closest ancestor of

https://doi.org/10.1017/S1471068416000442 Published online by Cambridge University Press

such computation in the call stack that is a cost center is q. The *accumulated cost* is formalized as a function $\mathscr{C}_{\mathtt{p}}^{\mathtt{q}} : \Pi \rightarrow \mathscr{R}_{\infty}$. We refer the reader to (Haemmerlé *et al.* 2016) for a formal definition of accumulated cost.[1]

The goal of static analysis is to infer approximations (i.e., abstractions) of the concrete functions $\mathscr{C}_{\mathtt{p}}^{\mathtt{q}}$ and $\mathscr{C}_{\mathtt{p}}$ (or, more precisely, of the extensions of such functions to the powerset of Π) that represent the *accumulated* and *standard* cost respectively. In this paper we propose a novel, general, and flexible framework for setting up cost equations/relations which can be instantiated for performing a wide range of static resource usage analyses, including both *accumulated cost* and standard cost. Our starting point is the well-developed technique of setting up recurrence relations representing resource usage functions parameterized by input data sizes (Wegbreit 1975; Rosendahl 1989; Debray *et al.* 1990; Debray and Lin 1993; Debray *et al.* 1997; Navas *et al.* 2007; Albert *et al.* 2011; Serrano *et al.* 2014), which are then solved to obtain (exact or safely approximated) closed-forms of such functions (i.e., functions that provide upper or lower bounds on resource usage in general).[2] Our proposal extends and generalizes these standard resource analysis techniques by introducing into the derived relations extra Boolean control variables whose value is 0 or 1. A particular resource profile can be analyzed by assigning values to the control variables, effectively switching on or off different terms in the relations. The standard resource analysis is obtained by assigning 1 to all variables. We also define a concrete Boolean variable assignment that instantiates our framework so that it performs *static profiling of accumulated cost*, similarly to (Haemmerlé *et al.* 2016), where the results are also parameterized by input data sizes. However, the approach we present in this paper is quite different from our previous approach (Haemmerlé *et al.* 2016), which was based on a program transformation. The main contributions of this paper and the differences and advantages over that work can be summarized as follows:

- We propose a novel, general, and flexible framework for setting up cost relations which can be instantiated for performing a wide range of resource usage analyses. Is more general than (Haemmerlé *et al.* 2016), which is limited to accumulated cost analysis.
- Our new approach can deal with non-deterministic/multiple-solution predicates, unlike (Haemmerlé *et al.* 2016). This is obviously a requirement for analyzing logic programs and is also useful for dealing with certain aspects of imperative programs, such as multiple dispatch; see (Méndez-Lojo *et al.* 2007). While our previous approach could conceivably be extended to deal with such programs, it would certainly result in a more complicated and indirect solution.
- Our new approach and its implementation are based on a direct application of abstract interpretation and integration into the Ciao preprocessor,

[1] In (Haemmerlé *et al.* 2016) we use the notation $\mathtt{C}_{\mathtt{q}}^{\mathtt{p}}(\bar{\mathtt{e}})$ instead of $\mathscr{C}_{\mathtt{p}}^{\mathtt{q}}(\bar{\mathtt{e}})$.

[2] In addition, recently many other approaches have been proposed for resource analysis (Vasconcelos and Hammond 2003; Hoffmann *et al.* 2012; Grobauer 2001; Igarashi and Kobayashi 2002; Nielson *et al.* 2002; Giesl *et al.* 2012; Albert *et al.* 2011; Gulwani *et al.* 2009). While based on different techniques, all these analyses are aimed at inferring the *standard* notion of cost. Please see (Haemmerlé *et al.* 2016) for a further discussion of related work.

https://doi.org/10.1017/S1471068416000442 Published online by Cambridge University Press

CiaoPP (Hermenegildo *et al.* 2005), rather than on a program transformation. As a result, many useful CiaoPP features are inherited for free, such as *multivariance* (being able to infer separate cost functions for different abstract call patterns for the same predicate), communication with the other required analyses, integrated treatment of special control features (such as, e.g., the cut), assertion-based verification and user interaction, efficient fixpoint, etc. Also, for this integration we define a novel abstract domain for resource analysis that keeps track of the *environment*.

- Furthermore, this direct implementation avoids the disadvantages of the transformation-based approach, such as making it more difficult to relate the results (and warnings/errors) to the original program, and complicating the task of the auxiliary analyses needed for cost analysis (types, modes, determinism, non-failure, etc.). This is because if the analyses are performed on the original program, then the results need to be transferred to the transformed program; and if the analyses are performed on the transformed program, then there is always the risk of loss of precision. Also, the transformation required by our previous approach is global, which is problematic for modular compilation. In general, this new approach allows much better and easier integration in a real-world compilation infrastructure.
- The integration also inherits the capability of CiaoPP's analyzers of *analyzing for several resources* at the same time. While it might be possible to define a new transformation capable of keeping track of several resources, this would further complicate the transformed program, and in any case requires additional work.
- Finally, as our experimental results show, our new approach is more efficient than the transformation-based approach. This is not only due to its implementation as a direct abstract interpretation, but also to the inclusion and use of reachability information, performed automatically by the abstract interpretation framework.

2 The standard parametric cost relations framework

We start by describing the kind of functions inferred by the standard cost analysis that we generalize for static profiling. Consider the function $\mathscr{C}_{\mathtt{p}} : \Pi \to \mathscr{R}_\infty$ introduced in the previous section. We extend it to the powerset of Π, i.e., $\hat{\mathscr{C}}_{\mathtt{p}} : 2^{\Pi} \to 2^{\mathscr{R}_\infty}$, where $\hat{\mathscr{C}}_{\mathtt{p}}(E) = \{\mathscr{C}_{\mathtt{p}}(\bar{\mathtt{e}}) \mid \bar{\mathtt{e}} \in E\}$. Our goal is to abstract (safely approximate, as accurately as possible) $\hat{\mathscr{C}}_{\mathtt{p}}$ (note that $\mathscr{C}_{\mathtt{p}}(\bar{\mathtt{e}}) = \hat{\mathscr{C}}_{\mathtt{p}}(\{\bar{\mathtt{e}}\})$). Intuitively, this abstraction is the composition of two abstractions: a size abstraction and a cost abstraction. The goal of the analysis is to infer two functions $\hat{\mathscr{C}}^{\downarrow}_{\mathtt{p}}$ and $\hat{\mathscr{C}}^{\uparrow}_{\mathtt{p}} : \mathscr{N}^m_\top \to \mathscr{R}_\infty$ that give lower and upper bounds respectively on the cost function $\hat{\mathscr{C}}_{\mathtt{p}}$, where $\mathscr{N}^m_\top$ is the set of m-tuples whose elements are natural numbers or the special symbol $\top$, meaning that the size of a given term under a given size metric is *undefined*. Such bounds are given as a function of tuples of data sizes (representing the concrete tuples of data of the concrete function $\hat{\mathscr{C}}_{\mathtt{p}}$). Typical size metrics are the actual value of a number,

https://doi.org/10.1017/S1471068416000442 Published online by Cambridge University Press

the length of a list, the size (number of constant and function symbols) of a term, etc. (Navas *et al.* 2007; Serrano *et al.* 2014).

Our starting point for static analysis is the standard general framework described in (Navas *et al.* 2007) for setting up parametric relations representing the resource usage (and size relations) of programs and predicates.[3] The analysis infers size relations for each predicate in a program: arithmetic expressions that provide the size of output arguments of the predicate as a function of its input data sizes. It also infers size relations for each clause, which give the input data sizes of the body literals as functions of the input data sizes to the clause head. Such size relations are instrumental for setting up cost relations. This work generalizes the approach of (Debray *et al.* 1990; Debray and Lin 1993; Debray *et al.* 1997) to infer *user-defined resources* (by using an extension of the Ciao assertion language (Hermenegildo *et al.* 2012)). The framework is doubly parametric: first, the costs inferred are parametric (they are functions of input data sizes), and second, the framework itself is parametric with respect to the resources being tracked and the type of approximation made (upper or lower bounds). Each concrete resource r to be tracked is defined by two sets of (user-provided) functions, some of which can be constant functions:

1. *Head cost* $\varphi(P)$*:* a function that returns the amount of resource r used by the unification of the calling literal (subgoal) P and the head of a clause matching P, plus any preparation for entering a clause (i.e., call and parameter passing cost).
2. *Predicate cost* $\Psi(\mathtt{p})$: it is also possible to define the *full cost* for a particular predicate p for resource r, i.e., the function $\Psi(\mathtt{p}) : \mathcal{N}_{\top}^{m} \rightarrow \mathcal{R}_{\infty}$ (with the sizes of p's input data as parameters, $\bar{\mathtt{x}}$) that returns the usage of resource r made by a call to this predicate. This is specially useful for built-in or external predicates, i.e., predicates for which the source code is not available and thus cannot be analyzed, or for providing a more accurate function than analysis can infer.[4] $\Psi(\mathtt{p})$ is expressed using the Ciao assertion language "trust" assertions (Hermenegildo *et al.* 2012).

Thus, for a clause $C \equiv \mathtt{p}(\bar{x}) :- \mathtt{q}_1(\bar{x}_1), \ldots, \mathtt{q}_n(\bar{x}_n)$, defining predicate p, the cost relation expressing the cost (for resource r) of the complete execution of a single call to p for input data sizes $\bar{\mathtt{x}}$ (obtaining all solutions), represented as $\mathtt{C}_{\mathtt{p}}(\bar{\mathtt{x}})$ is:

$$\mathtt{C}_{\mathtt{p}}(\bar{\mathtt{x}}) = \varphi(\mathtt{p}(\bar{\mathtt{x}})) + \sum_{i=1}^{lim(C,\bar{\mathtt{x}})} sols_i \times \mathtt{C}_{\mathtt{q}_i}(\bar{\mathtt{x}}_i) \tag{1}$$

[3] We give equivalent but simpler descriptions than in (Navas *et al.* 2007), which are allowed by assuming that programs are the result of a normalization process that makes all unifications explicit in the clause body, so that the arguments of the clause head and the body literals are all unique variables. We also omit the resource and approximation identifiers, r and ap respectively, since they are assumed to be arguments of all expressions that yield a resource usage.

[4] Note that sometimes approximations have to be used when solving recurrence relations, and there are other potential sources of loss of precision in the intervening analyses, which can accumulate in larger programs. In these cases trust assertions can be used in key places to recover precision. While this implies a burden, it is certainly always better than having to do all the cost analysis of the program by hand.

https://doi.org/10.1017/S1471068416000442 Published online by Cambridge University Press

where $sols_i$ represents the product of the number of solutions produced by the ancestor literals of $\mathtt{q}_i(\bar{x}_i)$ in the clause body:

$$sols_i = \prod_{j=1}^{i-1} s_{pred}(\mathtt{q}_j(\bar{x}_j)) \tag{2}$$

where $s_{pred}(\mathtt{q}_j(\bar{x}_j))$ gives the number of solutions produced by $\mathtt{q}_j(\bar{x}_j)$, and $lim(C, \bar{\mathtt{x}})$ gives the index of the last body literal that is called in the execution of clause C.

The (*standard*) cost of a body literal $\mathtt{q}_i(\bar{x}_i)$, i.e., $\mathtt{C}_{\mathtt{q}_i}(\bar{\mathtt{x}}_i)$, is obtained from the costs of all clauses applicable to it that are executed, by using an aggregation operator $\bigodot$. The resulting set of cost relations can be considered a definition of the resource usage semantics of a program. Ideally, we would like to find solutions to such relations, i.e., closed-form functions that give the resource usage of the programs and all of its predicates. However, this is impossible to do statically for all cases, and we then seek approximations, both upper and lower bounds. For this reason, we use a parametric operator $\bigodot(ap)$ that depends on the approximation *ap* being performed. For example, if *ap* is the identifier for lower bounds approximation (*lb*), then $\bigodot(ap)$ is the *min* function. If *ap* is the identifier for upper bound approximation (*ub*), then a possible conservative definition for $\bigodot(ap)$ is the $\sum$ function. In this case, and since the number of solutions generated by a predicate that will be demanded is generally not known in advance, a conservative upper bound on the computational cost of a predicate can be obtained by assuming that all solutions are needed, and that all clauses are executed. Then, the cost of the predicate is assumed to be the sum of the costs of all of its clauses. However, it is straightforward to take mutual exclusion into account to obtain a more precise estimate of the cost of a predicate, using the maximum of the costs of mutually exclusive groups of clauses, as done in (Serrano *et al.* 2014). Similarly, we use safe approximations of the function $lim(C, \bar{\mathtt{x}})$ in Expression 1 by introducing the function $lim(C, \bar{\mathtt{x}}, ap)$ that returns the index of a literal in the clause body depending on the approximation identifier *ap*. For example, $lim(C, \bar{\mathtt{x}}, ub) = n$ (the index of the last body literal) and $lim(C, \bar{\mathtt{x}}, lb)$ is the index of the leftmost body literal that could possibly fail. [5] If the cost of a $\mathtt{q}_i$ is given by a *trust assertion* as a function $\Psi(\mathtt{q}_i)(\bar{\mathtt{y}})$ then the closed-form $\Psi(\mathtt{q}_i)(\bar{\mathtt{x}})$ is used directly instead of the symbolic $\mathtt{C}_{\mathtt{q}_i}(\bar{\mathtt{x}})$ appearing in the set of cost relations to be solved.

Example 1

Consider the following program that checks whether a number n is prime based on Wilson's theorem: *any integer* $n > 1$ *is prime iff* $(n-1)! \equiv -1 \pmod{n}$. Equivalently, n is prime iff $(n-1)! + 1$ is a multiple of n.

```
1 prime(X):- X > 1, X1 is X - 1, fact(X1,F1), F is F1 + 1, multiple(F,X).
2
3 fact(X,1):- X = 1.
4 fact(X,Y):- X > 1, X1 is X - 1, fact(X1,Y1), Y is Y1*X.
```

[5] CiaoPP implements analyses like coverage, non-failure, cardinality, reachability, modes, shapes, treatment of cut, etc. that are instrumental in this context; see (Hermenegildo *et al.* 2005) and its references.

https://doi.org/10.1017/S1471068416000442 Published online by Cambridge University Press

Assume that `multiple` is a naively implemented library predicate, so that its resource usage, in number of resolution steps, is linear on the size of the input: $\mathtt{C}_{\mathtt{multiple}}(n,m) = n+1$ if $n > 1$ (given by using a trust assertion). Assume that we want to infer the standard cost of this predicate in resolution steps, i.e., we define $\varphi(\mathtt{p}(\bar{\mathtt{x}})) = 1$ for all predicates $\mathtt{p} \in \mathscr{P}$. For brevity, we also assume that we are only interested in inferring upper bounds on resource usages, so that the expression $\mathtt{C}_{\mathtt{p}}(\bar{\mathtt{x}})$ appearing in Equation 1 is understood to represent an upper bound, and, assuming no definite failure information, then $lim(C, \bar{\mathtt{x}})$ is the index of the last body literal of clause C. Finally, we also assume that size relations have been inferred for the different arguments in a clause, and that the size metric used is the actual value of an argument, since all arguments are numeric. Such relations are obvious in this example, so that we focus only on cost relations. However, as already stated, CiaoPP is able to infer and deal with a rich set of size metrics, and also infer such size relations. The size of the *kth* output argument of predicate `p`, given as a function of the input data sizes $\bar{n}$ to that predicate is represented as $Sz^k_{\mathtt{p}}(\bar{n})$. It is important also to mention the modes of these predicates (again, inferred automatically by CiaoPP): `prime` has one input argument and no output; `multiple` has two input arguments and no output; and `fact` has one input and one output, whose size we have assumed is already inferred in terms of the size of the input by the size analysis. This size is represented by $Sz^2_{\mathtt{fact}}(n)$, and is obtained from the setting up of the following size relation:

$$Sz^2_{\mathtt{fact}}(n) = 1 \text{ if } n = 1\,, \quad Sz^2_{\mathtt{fact}}(n) = n \times Sz^2_{\mathtt{fact}}(n-1) \quad \text{if } n > 1$$

By solving this recurrence, the size analysis obtains the closed-form $Sz^2_{\mathtt{fact}}(n) = n!$. Regarding the number of solutions, in this example all the predicates generate at most one solution, thus $\forall i : sols_i = 1$ in Equation 1. Now we have all the necessary elements to set up the cost relations for `prime`, `fact`, and `multiple`:

$$\begin{aligned} \mathtt{C}_{\mathtt{fact}}(n) &= 1 && \text{if } n = 1 \\ \mathtt{C}_{\mathtt{fact}}(n) &= 1 + \mathtt{C}_{\mathtt{fact}}(n-1) && \text{if } n > 1 \\ \mathtt{C}_{\mathtt{multiple}}(n,m) &= n+1 && \text{if } n > 1 \end{aligned}$$

$$\mathtt{C}_{\mathtt{prime}}(n) = 1 + \mathtt{C}_{\mathtt{fact}}(n-1) + \mathtt{C}_{\mathtt{multiple}}(Sz^2_{\mathtt{fact}}(n-1)+1, n) \text{ if } n > 1$$

Note that in this program, the size of the input of the call to `multiple` is given by the size of the output of `fact`, represented by $Sz^2_{\mathtt{fact}}(n)$. After solving these equations and composing the closed forms, we obtain the following closed form functions:

$$\begin{aligned} \mathtt{C}_{\mathtt{fact}}(n) &= n && \text{if } n > 1 \\ \mathtt{C}_{\mathtt{multiple}}(n,m) &= n+1 && \text{if } n > 1 \\ \mathtt{C}_{\mathtt{prime}}(n) &= (n-1)! + n + 3 && \text{if } n > 1 \end{aligned}$$

Example 2

Consider the following program $\mathscr{P}$:

https://doi.org/10.1017/S1471068416000442 Published online by Cambridge University Press

```
1  p(X,Y):- h(X), q(X,Y), w(Y),
       s(X).
2
3  q(0,_).
4  q(X,Y):- X > 0, X1 is X - 1,
5           m(Y), q(X1,Y), s(X).
```

```
8  m(0).
9  m(X):- X > 0, w(X), X1 is X - 1,
       m(X1).
10
11 s(0).
12 s(X):- X > 0, X1 is X - 1, w(X),
       s(X1).
13
14 h(2).
15 h(3).
```

Assume as in the previous example that we want to infer upper bounds of the standard costs of all the predicates in resolution steps, i.e., $\varphi(\mathtt{p}(\bar{\mathtt{x}})) = 1$ for all predicates $\mathtt{p} \in \mathscr{P}$. Assume also that `w` is a library predicate and that its (standard) cost is given as a predicate cost function (by using a trust assertion):

$$\Psi(\mathtt{w})(n) = 2n + 1 \tag{3}$$

We assume again that the size metric used is the actual value of the arguments, since they are all numeric, and that size relations, again obvious, have been inferred for all clause arguments, which are all inputs, and we focus only on cost relations. The cost relation for the recursive clause of predicate `s`, according to Expression 1 is (for simplicity, $sols_i = 1$ for all predicates in this example):

$$\mathtt{C}_\mathtt{s}(n) = 1 + \mathtt{C}_\mathtt{w}(n) + \mathtt{C}_\mathtt{s}(n-1) \quad \text{if } n > 0$$

Since $\mathtt{C}_\mathtt{w}(n)$ is given by a trust assertion as $\Psi(\mathtt{w})(n) = 2\,n+1$ (Expression 3), this cost relation, together with the one for the non-recursive clause, form the system:

$$\begin{aligned} \mathtt{C}_\mathtt{s}(n) &= 1 && \text{if } n = 0 \\ \mathtt{C}_\mathtt{s}(n) &= 1 + 2n + 1 + \mathtt{C}_\mathtt{s}(n-1) && \text{if } n > 0 \end{aligned}$$

and its closed-form solution is $\mathtt{C}_\mathtt{s}(n) = n^2 + 3n + 1$ for $n \geqslant 0$. The same cost relations correspond to predicate `m`, therefore its closed form is $\mathtt{C}_\mathtt{m}(n) = n^2 + 3n + 1$ for $n \geqslant 0$. For predicate `h`, the following non-recursive system of cost relations is set up:

$$\mathtt{C}_\mathtt{h}(n) = 1, \text{ if } n = 2 \quad \text{and} \quad \mathtt{C}_\mathtt{h}(n) = 1, \quad \text{if } n = 3$$

obtaining $\mathtt{C}_\mathtt{h}(n) = 1$, since the clauses of `h` are mutually exclusive. Now, the cost relations for `q` are:

$$\begin{aligned} \mathtt{C}_\mathtt{q}(m,n) &= 1 && \text{if } m = 0 \\ \mathtt{C}_\mathtt{q}(m,n) &= 1 + \mathtt{C}_\mathtt{m}(n) + \mathtt{C}_\mathtt{q}(m-1,n) + \mathtt{C}_\mathtt{s}(m) && \text{if } m > 0 \end{aligned}$$

Replacing $\mathtt{C}_\mathtt{m}(n)$ and $\mathtt{C}_\mathtt{s}(n)$ with their corresponding closed-form functions obtained before, and solving the recurrence, we obtain $\mathtt{C}_\mathtt{q}(m,n) = \frac{1}{3}m^3 + mn^2 + 2m^2 + 3mn + \frac{14}{3}m + 1$. Finally, the cost relations for the main predicate `p` result in:

$$\mathtt{C}_\mathtt{p}(m,n) = 1 + \mathtt{C}_\mathtt{h}(m) + \mathtt{C}_\mathtt{q}(m,n) + \mathtt{C}_\mathtt{w}(n) + \mathtt{C}_\mathtt{s}(m)$$

and its closed form is: $\mathtt{C}_\mathtt{p}(m,n) = \frac{1}{3}m^3 + mn^2 + 3m^2 + 3mn + \frac{23}{3}m + 2n + 4$.

3 Generalizing the standard cost relations approach

Our proposal extends and generalizes the approach described in Sect. 2. We introduce a new concept of cost, $\mathtt{C}^{\mathtt{c}}_{\mathtt{p},e}(\bar{\mathtt{x}})$, representing the (part of the) cost of the complete

https://doi.org/10.1017/S1471068416000442 Published online by Cambridge University Press

execution of a single call $\mathtt{p}(\bar{\mathtt{x}})$ (i.e., $\mathtt{C_p}(\bar{\mathtt{x}})$ in Sect. 2), performed in an *environment* e, that is attributed/assigned to *cost center* $\mathtt{c}$ of the program. The parameter e is used to capture a broad notion of *environment*. For example, it can be just the name of a predicate that is an ancestor of $\mathtt{p}$ in the call stack. In a more complex setting, for example when inferring hardware-dependent resources, such as energy (Navas *et al.* 2008; Liqat *et al.* 2014; Liqat *et al.* 2016), e can also include information about the state of the hardware (or the whole system, including the running software environment), e.g., the last instruction executed (useful for modeling the *switching cost* of instructions), temperature, voltage, cache state, and pipeline state. There is of course a trade-off between the amount of information in e and analysis efficiency and accuracy.

As already said, and similarly to (Haemmerlé *et al.* 2016), in this paper we assume that a *cost center* is a predicate in the program. Conceptually, we can say that we extend the notion of resource so that it is now a pair $(\mathtt{c}, r)$, where r is a resource identifier as before (e.g., resolution steps, execution time, energy, etc.), and $\mathtt{c}$ is the cost center (predicate) that the resource usage is attributed/assigned to.

We also introduce *Boolean functions* $B_\varphi(\mathtt{p}, \mathtt{c}, e)$ and $B(\mathtt{p}, \mathtt{c}, e, \mathtt{q})$ to control which terms of the cost relation should be considered. To this end, Expression 1 is generalized as:

$$\mathtt{C}^{\mathtt{c}}_{\mathtt{p},e}(\bar{\mathtt{x}}) = B_\varphi(\mathtt{p}, \mathtt{c}, e) \times \varphi(\mathtt{p}(\bar{\mathtt{x}})) + \textstyle\sum_{i=1}^{lim(C,\bar{\mathtt{x}})} sols_i \times B(\mathtt{p}, \mathtt{c}, e, \mathtt{q}_i) \times \mathtt{C}^{\mathtt{c}}_{\mathtt{q}_i,e'}(\bar{\mathtt{x}}_i) \qquad (4)$$

where $e' = \mathscr{E}(\mathtt{p}, \mathtt{c}, e, \mathtt{q}_i(\bar{\mathtt{x}}_i))$, and $\mathscr{E}$ is the *environment change* function, which obtains the new environment for $\mathtt{q}_i$. If the cost of $\mathtt{p}$ is given (by using a trust assertion) as a function $\Psi(\mathtt{p})(\bar{\mathtt{x}})$, then:

$$\mathtt{C}^{\mathtt{c}}_{\mathtt{p},e}(\bar{\mathtt{x}}) = B_\varphi(\mathtt{p}, \mathtt{c}, e) \times \Psi(\mathtt{p})(\bar{\mathtt{x}}) \qquad (5)$$

Again, this equational framework can be instantiated to obtain the standard cost by defining $B_\varphi(\mathtt{p}, \mathtt{c}, e) = B(\mathtt{p}, \mathtt{c}, e, \mathtt{q}) \equiv 1$, and defining $\mathscr{E}$ so that it does not change the environment and always returns the input environment, i.e., $\mathscr{E}(\mathtt{p}, \mathtt{c}, e, \mathtt{q}_i(\bar{\mathtt{x}}_i)) = e$. The *standard* cost $\mathtt{C_p}(\bar{\mathtt{x}})$ is then given by $\mathtt{C}^{\mathtt{p}}_{\mathtt{p},\perp}(\bar{\mathtt{x}})$, where $\perp$ is the *null* environment, in which no information about the environment is tracked, and the only cost center that the cost of a single call to $\mathtt{p}$ is attributed to is the predicate $\mathtt{p}$ itself.

4 Instantiation for parametric accumulated-cost static profiling

We now instantiate the general approach described in Sect. 3 for the static inference of accumulated cost. The advantages of this approach with respect to our previous approach to accumulated cost inference (Haemmerlé *et al.* 2016) were already discussed in Sect. 1.

Assume we are given a set of (user-defined) cost centers $\diamond$, which, as mentioned before, in our approach are program predicates. Assuming that $\mathtt{p}$ is a cost center, the standard cost of a single call $\mathtt{p}(\bar{\mathtt{x}})$ (as defined in Sect. 1, and whose inference was discussed in Sect. 2) is the sum of its accumulated costs in all the cost centers in the program, or, equivalently in all the cost centers that are descendants (in the call stack) of $\mathtt{p}$. This is formally expressed in (Haemmerlé *et al.* 2016) Theorem

https://doi.org/10.1017/S1471068416000442 Published online by Cambridge University Press

1, and, intuitively, the proof is based on the fact that, according to the definition of accumulated cost, the cost of any computation performed during the complete execution of $p(\bar{x})$ is uniquely attributed to a cost center (predicate): the closest ancestor of such computation in the call stack that is a cost center.

Given a predicate p, we refer to the computations performed by a call $p(\bar{x})$ that are not under the scope of any cost center that is a descendant (in the call stack) of p, as the *residual computations* of p. We refer to the cost of such computations as the *residual cost* of p. Note that such computations include the computations performed by calls to non-cost-center predicates that are descendants of p and that are not under the scope of any cost center that is a descendant of p. Assume that the analysis is inferring accumulated costs on a given cost center c. When analyzing a call to a non-cost-center predicate p, its residual cost must be attributed to c only if the call $p(\bar{x})$ is under the scope of c (i.e., is a descendant of c). When analyzing a call to a cost-center predicate p, its residual cost must be attributed to c only if $p = c$. Thus, in the expression $C^c_{p,e}(\bar{x})$ (where necessarily $c \in \diamond$) the environment e is just a Boolean value representing whether the (single) call to p is in the scope of cost center c ($e = 1$) or not ($e = 0$).

To this end, we define the *environment change* function as follows: $\mathscr{E}(p, c, e, _) \equiv (p = c \vee (p \notin \diamond \wedge e))$.

Knowing that a given predicate cannot be called by another during program execution allows the analysis to ignore some parts not affecting the cost to be inferred. We define a simple *calls* relation between predicates as: p calls q, denoted $p \rightsquigarrow_\alpha q$, if and only if a literal with predicate symbol q appears in the body of a clause defining p; $\rightsquigarrow^\star_\alpha$ is the reflexive transitive closure of $\rightsquigarrow_\alpha$. This $\rightsquigarrow_\alpha$ relation is an abstraction (over-approximation) of the concrete $\rightsquigarrow$ relation (a more precise abstraction is computed by CiaoPP).

The Boolean assignment functions (appearing in Expression 4) are defined as follows:

$$B_\varphi(p, c, e) \equiv (p = c \vee (p \notin \diamond \wedge e)) \tag{6}$$

$$B(p, c, e, q) \equiv B_\varphi(p, c, e) \vee (q \rightsquigarrow^\star_\alpha p) \tag{7}$$

Note that the analysis of the accumulated cost of a given non-cost-center predicate p in a given cost center c can create at most two versions of $C^c_{p,e}(\bar{x})$ for the same input (calling pattern) $\bar{x}$ (and hence, there will be at most two versions of the cost relations for p): the version $C^c_{p,1}(\bar{x})$ created if there is a (direct or indirect) call to p in the scope of c, e.g., if such call is in the body of a clause defining c (in which case the φ *cost* is added to the cost relations for p), and the variant $C^c_{p,0}(\bar{x})$ created if there is a call to p not in the scope of c (in which case the φ *cost* is not added).

Lemma 1
$\forall p, q \in \diamond$, $\forall e \in \{0, 1\}$, it holds that $\mathscr{E}(p, q, e, _) \equiv (p = q)$ and $B_\varphi(p, q, e) \equiv (p = q)$.

This implies that:

Lemma 2
$\forall p, q \in \diamond$ it holds that $C^q_{p,0}(\bar{x}) = C^q_{p,1}(\bar{x})$.

https://doi.org/10.1017/S1471068416000442 Published online by Cambridge University Press

Thus, if $\mathtt{p} \in \diamond$ we omit the environment e and write $\mathtt{C}^{\mathtt{q}}_{\mathtt{p}}(\bar{\mathtt{x}})$. Note that necessarily $\mathtt{q} \in \diamond$.

Lemma 3
$\forall \mathtt{p}, \mathtt{q} \in \diamond$, if $\mathtt{p} \not\rightsquigarrow^{\star}_{\alpha} \mathtt{q}$ then $\mathtt{C}^{\mathtt{q}}_{\mathtt{p}}(\bar{\mathtt{x}}) = 0$.

Lemma 4
$\forall \mathtt{p} \notin \diamond, \forall \mathtt{q} \in \diamond$, if $\mathtt{p} \not\rightsquigarrow^{\star}_{\alpha} \mathtt{q}$ then $\mathtt{C}^{\mathtt{q}}_{\mathtt{p},0}(\bar{\mathtt{x}}) = 0$.

Note also that in the standard cost relation-based static analysis, cost relations are set up for each predicate in the program. In the approach we propose here for accumulated cost, cost relations are set up for each cost center and for each predicate in the program.

Example 3
In Example 1, predicate `prime` was found too expensive in terms of resolution steps to be practical, since $\mathtt{C}_{\mathtt{prime}}(n) \in \mathcal{O}(n!)$. However, the standard cost inferred for all the predicates called from `prime` is linear, and it is not easy to detect at first glance where the resource is really consumed. To locate the culprit, traditionally this would be attempted using a dynamic profiling tool, executing the program with several test cases –commonly known as *hot spot detection*. However, as with the standard cost analysis, we want to detect such hot spots *statically*, in order to have sound information for *any* possible input. For this purpose, we perform the accumulated cost analysis declaring that all predicates are cost centers (i.e, $\diamond = \{\mathtt{prime}, \mathtt{fact}, \mathtt{multiple}\}$). Based on the equational framework instantiation above and Lemma 2, consider the cost of a single call to `prime` accumulated in `fact`, $\mathtt{C}^{\mathtt{fact}}_{\mathtt{prime}}(n)$, for an input size n. As already stated, the number of solutions of all these predicates is 1, and the output sizes have already been inferred. For the sake of conciseness, from now on we refer to `prime`, `fact` and `multiple` as p, f and m respectively. The cost relations for the accumulated costs in cost center `fact` are:

$$\mathtt{C}^f_p(n) = B_\varphi(p, f, _) \times \varphi(p(n)) + \mathtt{C}^f_f(n-1) + \mathtt{C}^f_m(Sz^2_f(n-1)+1)$$

$$\begin{array}{ll} \mathtt{C}^f_f(n) = B_\varphi(f, f, _) \times \varphi(f(n)) & \text{if } n = 1 \\ \mathtt{C}^f_f(n) = B_\varphi(f, f, _) \times \varphi(f(n)) + \mathtt{C}^f_f(n-1) & \text{if } n > 1 \end{array}$$

$$\mathtt{C}^f_m(n) = B_\varphi(m, f, _) \times \Psi(m)(n) = B_\varphi(m, f, _) \times (n+1) \quad \text{if } n > 1$$

We have that $\varphi(_) = 1$, and, according to Expression 6, $B_\varphi(p, f, _) = B_\varphi(m, f, _) = 0$ and $B_\varphi(f, f, _) = 1$. Using these values, the cost relations defining $\mathtt{C}^f_p(n)$ are:

$$\mathtt{C}^f_p(n) = \mathtt{C}^f_f(n-1)$$

$$\begin{array}{ll} \mathtt{C}^f_f(n) = 1 & \text{if } n = 1 \\ \mathtt{C}^f_f(n) = 1 + \mathtt{C}^f_f(n-1) & \text{if } n > 1 \end{array}$$

Solving this system of cost relations, we finally obtain: $\mathtt{C}^f_p(n) = n$.
Analogously, we obtain the closed-form functions for $\mathtt{C}^m_p(n)$ and $\mathtt{C}^p_p(n)$:

$$\begin{array}{ll} \mathtt{C}^p_p(n) = 1 & \text{if } n > 1 \\ \mathtt{C}^m_p(n) = (n-1)! + 2 & \text{if } n > 1 \end{array}$$

https://doi.org/10.1017/S1471068416000442 Published online by Cambridge University Press

Now, it is clear that the most expensive part of this program is the call to `multiple`. Even though the standard cost of `multiple` is linear, its input size is the output size of the call to `fact` (plus 1), which is the factorial of the input to `prime` minus 1. In this case the problem can really only be fixed by using a better implementation of `multiple` ($\mathcal{O}(1)$) or of `prime`, to achieve the expected polynomial resource usage.

This example illustrates how the accumulated cost is more useful than the standard cost. Neither the standard cost of `multiple` ($n + 1$) nor the number of calls to this predicate from `prime` (since it is called just once) gives a direct hint that this predicate is responsible for most of the resource consumption of `prime`.

Example 4

Consider again the program in Example 2. Assume that we declare that predicates `p`, `q`, `m` and `h` are cost centers, i.e., $\diamond = \{\mathtt{p}, \mathtt{q}, \mathtt{m}, \mathtt{h}\}$, and `s` and `w` are not. For space reasons, we will only illustrate the inference of upper bounds on accumulated costs in all cost centers.

The accumulated costs in cost center `q` are inferred as follows. Consider the clause defining predicate `p`. Since $\mathtt{p} \in \diamond$, by Lemma 1 the current environment e is irrelevant for the computation of the new environment e' (i.e., $e' = \mathscr{E}(\mathtt{p}, \mathtt{q}, 0, _) = \mathscr{E}(\mathtt{p}, \mathtt{q}, 1, _) \equiv (\mathtt{p} = \mathtt{q}) \equiv 0$), and for the computation of the head cost, i.e., $B_\varphi(\mathtt{p}, \mathtt{q}, 0) = B_\varphi(\mathtt{p}, \mathtt{q}, 1) \equiv (\mathtt{p} = \mathtt{q}) \equiv 0$. Thus, the cost relation for `p` according to Equation 4 is $\mathtt{C}^{\mathtt{q}}_{\mathtt{p}}(\mathtt{x}, \mathtt{y}) = \mathtt{C}^{\mathtt{q}}_{\mathtt{h}}(\mathtt{x}) + \mathtt{C}^{\mathtt{q}}_{\mathtt{q}}(\mathtt{x}, \mathtt{y}) + \mathtt{C}^{\mathtt{q}}_{\mathtt{w},0}(\mathtt{y}) + \mathtt{C}^{\mathtt{q}}_{\mathtt{s},0}(\mathtt{x})$. Consider predicate `q` now. Since $B_\varphi(\mathtt{q}, \mathtt{q}, e) \equiv (\mathtt{q} = \mathtt{q}) \equiv 1$ and $\mathscr{E}(\mathtt{q}, \mathtt{q}, e, _) \equiv (\mathtt{q} = \mathtt{q}) \equiv 1$ for $e \in \{0, 1\}$, the cost relations for the base case and recursive clause of `q` respectively are:

$$\begin{aligned} \mathtt{C}^{\mathtt{q}}_{\mathtt{q}}(\mathtt{x}, \mathtt{y}) &= B_\varphi(\mathtt{q}, \mathtt{q}, _) \times 1 = 1 \times 1 = 1 && \text{if } \mathtt{x} = 0 \\ \mathtt{C}^{\mathtt{q}}_{\mathtt{q}}(\mathtt{x}, \mathtt{y}) &= 1 + \mathtt{C}^{\mathtt{q}}_{\mathtt{m}}(\mathtt{y}) + \mathtt{C}^{\mathtt{q}}_{\mathtt{q}}(\mathtt{x} - 1, \mathtt{y}) + \mathtt{C}^{\mathtt{q}}_{\mathtt{s},1}(\mathtt{x}) && \text{if } \mathtt{x} > 0 \end{aligned}$$

For expression $\mathtt{C}^{\mathtt{q}}_{\mathtt{s},1}(\mathtt{x})$ appearing in the recursive cost relation for `q` above (i.e., the version of the cost of `s` when called in the scope of cost center `q`), the cost relations are:[6]

$$\begin{aligned} \mathtt{C}^{\mathtt{q}}_{\mathtt{s},1}(\mathtt{x}) &= 1 && \text{if } x = 0 \\ \mathtt{C}^{\mathtt{q}}_{\mathtt{s},1}(\mathtt{x}) &= 1 + \mathtt{C}^{\mathtt{q}}_{\mathtt{w},1}(\mathtt{x}) + \mathtt{C}^{\mathtt{q}}_{\mathtt{s},1}(\mathtt{x} - 1) && \text{if } \mathtt{x} > 0 \end{aligned}$$

We now need to infer the function represented by expression $\mathtt{C}^{\mathtt{q}}_{\mathtt{w},1}(\mathtt{x})$ appearing in the recursive cost relation for `s` above. Since the cost function for `w` is given by a trust assertion (see Expression 3) and $B_\varphi(\mathtt{w}, \mathtt{q}, 1) = 1$, we have that $\mathtt{C}^{\mathtt{q}}_{\mathtt{w},1}(\mathtt{x}) = B_\varphi(\mathtt{w}, \mathtt{q}, 1) \times (2\ \mathtt{x} + 1) = 2\ \mathtt{x} + 1$. Using this function, the closed-form solution for $\mathtt{C}^{\mathtt{q}}_{\mathtt{s},1}(\mathtt{x})$ is $\mathtt{x}^2 + 3\mathtt{x} + 1$ for $\mathtt{x} \geqslant 0$. For expression $\mathtt{C}^{\mathtt{q}}_{\mathtt{w},0}(\mathtt{y})$ appearing in the equation for `p` above, we have that $\mathtt{C}^{\mathtt{q}}_{\mathtt{w},0}(\mathtt{y}) = B_\varphi(\mathtt{w}, \mathtt{q}, 0) \times (2\mathtt{y} + 1) = 0 \times (2\mathtt{y} + 1) = 0$. Now, for expression $\mathtt{C}^{\mathtt{q}}_{\mathtt{s},0}(\mathtt{x})$ appearing in the cost relation for `p` above (i.e., the version of the cost of `s` when it is called not in the scope of cost center `q`), we have that $\mathtt{C}^{\mathtt{q}}_{\mathtt{s},0}(\mathtt{x}) = 0$ (Lemma 4). For expression $\mathtt{C}^{\mathtt{q}}_{\mathtt{m}}(\mathtt{y})$ appearing in the second cost relation for `q` above, we have that $\mathtt{C}^{\mathtt{q}}_{\mathtt{m}}(\mathtt{y}) = 0$ (Lemma 3), and no cost relation is set up for

[6] Since $\mathtt{s} \notin \diamond$, the environment is needed in this case.

https://doi.org/10.1017/S1471068416000442 Published online by Cambridge University Press

predicate m. Now, the accumulated costs in cost center h are inferred as follows. The accumulated cost in h for a call to p is given by:

$$C_p^h(x, y) = C_h^h(x) + C_q^h(x, y) + C_{w,0}^h(y) + C_{s,0}^h(x)$$

We have that:

$$C_q^h(x, y) = 0 \text{ and } C_m^h(y) = 0 \text{ (by Lemma 3)}$$

and:

$$C_{s,0}^h(x) = 0 \text{ (by Lemma 4)}$$

and $C_{w,0}^h(y) = B_\varphi(w, h, 0) \times \Psi(m)(y) = 0$. Then, the cost relations for the accumulated cost in h for a call to h are:

$$C_h^h(x) = B_\varphi(h, h, _) \times 1 = 1$$
$$C_h^h(x) = B_\varphi(h, h, _) \times 1 = 1$$

Therefore, $C_h^h(x) = 1$ and $C_p^h(x, y) = 1$. For cost center m we have:

$$C_p^m(x, y) = xy^2 + 3xy + 2y + \tfrac{1}{3}x^3 + \tfrac{5}{2}x^2 + \tfrac{25}{6}x + 1$$
$$C_q^m(x, y) = xy^2 + 3xy + \tfrac{1}{3}x^3 + \tfrac{3}{2}x^2 + \tfrac{13}{6}x$$
$$C_m^m(x) = x^2 + 3x + 1$$

Finally, for cost center p we have:

$$C_p^p(x, y) = 1\ C_q^p(x, y) = 0\ C_m^p(x) = 0\ C_h^p(x) = 0$$

Note that the large complexity of $C_p^m(x, y)$ makes us realize that if we move the call m(y) from the recursive clause of q to the clause of p:

```
1 p(X, Y):- h(X), m(Y), q(X, Y), w(Y), s(X).
2
3 q(0, _).
4 q(X, Y):- X > 0, X1 is X - 1, q(X1, Y), s(X).
```

then, the standard cost of p will be reduced. In particular, it is reduced from $C_p(x, y) = \frac{1}{3}x^3 + xy^2 + 3x^2 + 3xy + \frac{23}{3}x + 2y + 4$ to $C_p(x, y) = y^2 + 5y + \frac{1}{3}x^3 + 3x^2 + \frac{20}{3}x + 8$.

5 Implementation and experimental results

We have implemented the proposed approach within the CiaoPP system, by extending the implementation of (Serrano *et al.* 2014). The latter improved on (Navas *et al.* 2007) by defining the resource analysis itself as an *abstract domain* that is integrated into the PLAI abstract interpretation framework (Muthukumar and Hermenegildo 1992; Puebla and Hermenegildo 1996) of CiaoPP, inheriting features such as multivariance, efficient fixpoints, and assertion-based verification and user interaction. A significant additional improvement brought about by (Serrano *et al.* 2014) is its use of a *sized types* abstract domain, which allows the inference of non-trivial cost bounds when these depend on the sizes of parts of input terms at any position and depth. The resulting abstract interpretation-based implementation builds the cost equations described in Sect. 3. Separate equations are built for each

https://doi.org/10.1017/S1471068416000442 Published online by Cambridge University Press

procedure *version* thanks to the built-in multivariance in PLAI. Other optimizations include not building equations for unreachable program parts.

Table 1 shows the results of the comparison between the proposed approach and our previous, program transformation-based approach (Haemmerlé *et al.* 2016) –**New** and **Prev** respectively from now on. Column **Bench** shows, for each program, the entry predicate (marked with a *star*, e.g., *sublist**) and the predicates that are declared as cost centers (which always include the entry predicate). **Acc. Cost** shows the parametric accumulated cost functions inferred for each cost center, which depend on the input data sizes of the entry predicate. For conciseness, we only show upper bound functions, although in the experiments both upper and lower bounds were inferred. The resource inferred in these tests is the number of resolution steps (i.e., each clause body is assumed to have unitary cost). The symbols in Column **C** compare **New** and **Prev**: $\times$ means that it is a non-deterministic program that produces multiple solutions and **New** is able to obtain non-trivial bounds while **Prev** fails to obtain a correct bound (as mentioned before, **Prev** is not applicable for these programs).= indicates that **New** obtains the same bounds as **Prev**. Only these two symbols are required because all the results coincide except for the non-deterministic programs. **AvD** is the average deviation of the accumulated costs obtained by evaluating the functions in Column **Acc. Cost**, with respect to the costs measured with a dynamic profiler (Mera *et al.* 2011). The input data for dynamic profiling was selected to exhibit the worst case execution,[7] in order to compare with upper bound functions. **Time (s)** lists the analysis times of **New** in seconds (Ciao/CiaoPP version 1.15-4048-g6bd1569, MacBook Pro, 2.4GHz Intel Core i7 CPU, 8 GB 1333 MHz DDR3 memory, MAC OS X Lion 10.7.5) and, between brackets, how efficient **New** is with respect to **Prev** ($\frac{\textbf{New}-\textbf{Prev}}{\textbf{Prev}} \times 100$). **New** is more efficient than **Prev** in all programs, with one exception (hanoi). Times are quite encouraging in any case, specially considering the currently inefficient implementation of the Mathematica interface, one of the solvers used for the recurrence equations.

Std. Cost shows the cost functions inferred using the standard notion of cost (in particular, the cost functions inferred by (Serrano *et al.* 2014)) for comparison with the accumulated cost functions (**Acc. Cost**). The latter clearly signal hot spots that are not visible from the standard cost functions. Note also that in all cases the sum of the functions for all the cost centers is the standard cost of the entry predicate. Due to space limitations we do not include analysis times for obtaining the standard costs in Column **Std. Cost**, but while the analysis times of **New** are higher, as expected, it is only by 20% on average. **#Calls** shows the number of times each predicate is called, as a function of input data sizes of the entry predicate. These functions are inferred using the standard analysis by defining explicitly a **#Calls** *resource* for each cost center predicate. A large complexity order in the number of calls to a predicate (in relation to that of a single call) suggests that it could be profitable to optimize the program to reduce the number of calls to this predicate, to effectively reduce its impact on the overall cost of the program. More interestingly, since both resources

[7] Except for *queens*: the *queens* program was simply run for 8 queens. The selection of input data that can make a program exhibit worst case execution is non-trivial.

https://doi.org/10.1017/S1471068416000442 Published online by Cambridge University Press

Table 1. *Experimental results (static profiling of accumulated cost)*

Bench	Acc. Cost	C	AvD	Time (s)	Std. Cost	#Calls	Acc.BigO
*sublist**	n_2+3	×	5%	4.7	$n_1n_2+3n_2+2$	2	$\mathcal{O}(n_2)$
append	$n_1n_2+2n_2-1$		40%	(NA)	$2n-1$	$n_1n_2+2n_2-1$	$\mathcal{O}(n_1n_2)$
*is_prime**	1	=	0%		$(n-1)!+n+3$	1	$\mathcal{O}(1)$
fact	n		0%	1.6	n	n	$\mathcal{O}(n)$
mult	$(n-1)!+2$		0%	(−24%)	$n+1$	$(n-1)!+2$	$\mathcal{O}(n!)$
*queens**	$n+2$	×	7%		$\mathcal{O}(n^n)^\dagger$	1	$\mathcal{O}(n)$
consistent	$\frac{((n-1)n-1)n^{n+1}+n}{(n-1)^2}$		10^4%	4.7	$2n+1$	$\frac{((n-1)n-1)n^{n+1}+n}{(n-1)^2}$	$\mathcal{O}(n^n)$
choose	$\frac{(2n-1)(n^n-1)}{(n-1)}$		10^4%	(NA)	$2n-1$	$\frac{(2n-1)(n^n-1)}{n-1}$	$\mathcal{O}(n^n)$
noattack	$\frac{(n-2)n^{(n+2)}+n^2}{(n-1)^2}$		10^4%		1	$\frac{(n-2)n^{n+2}+n^2}{(n-1)^2}$	$\mathcal{O}(n^n)$
*search**	1	×	0%	1.4	$2n+2$	1	$\mathcal{O}(1)$
member	$2n+1$		0.1%	(NA)	$2n+1$	$2n+1$	$\mathcal{O}(n)$
*appAll2**	b_1	=	0%	5.3	$\mathcal{O}(b_1b_2b_3)^\dagger$	1	$\mathcal{O}(b_1)$
appAll	b_1b_2		0%	(-16%)	b_1b_2	b_1	$\mathcal{O}(b_1b_2)$
append	$2b_1b_2b_3$		0%		n	$b_1b_2+b_1$	$\mathcal{O}(b_1b_2b_3)$
*hanoi**	2^n-1	=	0%	1.6	$2^{n+1}-2$	1	$\mathcal{O}(2^n)$
move	2^n-1		0%	(-19%)	1	2^n-1	$\mathcal{O}(2^n)$
*coupled**	1	=	0%	2.4	$n+2$	1	$\mathcal{O}(1)$
p	$\frac{n}{2}+\frac{(-1)^n}{4}+\frac{3}{4}$		1.2%	(-14%)	$n+1$	$\frac{n}{2}-\frac{(-1)^n}{4}+\frac{1}{4}$	$\mathcal{O}(n)$
q	$\frac{n}{2}-\frac{(-1)^n}{4}+\frac{1}{4}$		0%		$n+1$	$\frac{n}{2}+\frac{(-1)^n}{4}-\frac{1}{4}$	$\mathcal{O}(n)$
*isort**	$n+1$	=	0%	3	n^2+n+1	$n+1$	$\mathcal{O}(n)$
insert	n^2		71%	(-19%)	$2n+1$	n^2	$\mathcal{O}(n^2)$
*minsort**	$n+1$	=	0%	3.5	$\frac{(n+1)^2}{2}+\frac{n+1}{2}$	1	$\mathcal{O}(n)$
findmin	$\frac{(n+1)^2}{2}+\frac{n-1}{2}$		7%	(-27%)	n	$n+1$	$\mathcal{O}(n^2)$
*dyade**	n_1	=	0%	3.2	$n_1(n_2+1)$	1	$\mathcal{O}(n_1)$
mult	n_1n_2		0%	(-20%)	n	n_1	$\mathcal{O}(n_1n_2)$
*variance**	1	=	0%	3.6	$2n^2$	1	$\mathcal{O}(1)$
sq_diff	$n-1$		0%	(-39%)	$2n_2n_1-2n_2$	$n-1$	$\mathcal{O}(n)$
mean	$2n^2-n$		0%		$2n+1$	n	$\mathcal{O}(n^2)$
*variance2**	1	=	0%	3.1	$5n+3$	1	$\mathcal{O}(1)$
sq_diff	n		0%	(-40%)	n	n	$\mathcal{O}(n)$
mean	$4n+2$		0%		$2n+1$	2	$\mathcal{O}(n)$
*listfact**	b_1	=	0%	1.9	$b_1(b_2+2)$	1	$\mathcal{O}(b1)$
fact	$b_1b_2+b_1$		0%	(-23%)	n	b_1	$\mathcal{O}(b_1b_2)$

† For space limitations only the complexity order is shown.

- $n_1, n_2, \ldots, n_k$ represent the sizes of k input arguments. For a single input argument, the subscript is dropped.
- $b_1, b_2, \ldots, b_k$ represent the sizes of the nested structures of an input argument, where b_1 represents the size of the outer most structure and b_k the inner most. In cases, where the cost only depends on the outer most structure, the previous representation is used.

https://doi.org/10.1017/S1471068416000442 Published online by Cambridge University Press

Acc. Cost and **#Calls** of a predicate p are expressed as functions of input data sizes of the entry predicate, their quotient (**Acc. Cost**/ **#Calls**) is meaningful and will give an approximation of the cost of a single call to p as a function of the input data sizes *of the entry predicate*. Note that the standard analysis (Column **Std. Cost**) also provides an upper-bound approximation of this cost but as a function of the input data sizes of q. Finally, Column **Acc.BigO** shows the *actual* asymptotic resource usage of the accumulated cost in different cost centers.

6 Conclusions

We presented a novel, general, and flexible framework for setting up cost equation-s/relations which can be instantiated for performing a wide range of resource usage analyses, including both *accumulated cost* and standard cost. We have also reported on an implementation of this general framework within the CiaoPP system, and its instantiation for accumulated cost, and provided some experimental results. The results show that the resulting accumulated cost analysis, in addition to providing results for non-deterministic programs, is also more efficient than our previous approach based on program transformation, and has a good number of additional advantages. We argue that our approach is quite general, and it can be easily applied to other paradigms, including imperative programs, functional programs, CHR, etc., using the strategy based on compilation to Horn Clauses, as in our previous work with Java or XC.

Supplementary material

For supplementary material for this article, please visit http://dx.doi.org/10.1017/S1471068416000442

References

Albert, E., Arenas, P., Genaim, S. and Puebla, G. 2011. Closed-form upper bounds in static cost analysis. *Journal of Automated Reasoning 46,* 2, 161–203.

Albert, E., Genaim, S. and Masud, A. N. 2011. More precise yet widely applicable cost analysis. In *Proc. of VMCAI'11*, LNCS, vol. 6538. Springer, 38–53.

Debray, S. K. and Lin, N. W. 1993. Cost analysis of logic programs. *ACM TOPLAS 15,* 5 (November), 826–875.

Debray, S. K., Lin, N.-W. and Hermenegildo, M. 1990. Task granularity analysis in logic programs. In *Proc. PLDI'90*. ACM, 174–188.

Debray, S. K., López-García, P., Hermenegildo, M. and Lin, N.-W. 1997. Lower bound cost estimation for logic programs. In *ILPS'97*. MIT Press, 291–305.

Giesl, J., Ströder, T., Schneider-Kamp, P., Emmes, F. and Fuhs, C. 2012. Symbolic evaluation graphs and term rewriting: a general methodology for analyzing logic programs. In *Proceedings of PPDP'12*. ACM, 1–12.

Grobauer, B. 2001. Cost recurrences for DML programs. In *Proceedings of ICFP '01*. ACM, New York, NY, USA, 253–264.

Gulwani, S., Mehra, K. K. and Chilimbi, T. M. 2009. SPEED: Precise and efficient static estimation of program computational complexity. In *The 36th Symposium on Principles of Programming Languages (POPL'09)*. ACM, 127–139.

Haemmerlé, R., Lopez-Garcia, P., Liqat, U., Klemen, M., Gallagher, J. P. and Hermenegildo, M. V. 2016. A transformational approach to parametric

https://doi.org/10.1017/S1471068416000442 Published online by Cambridge University Press

accumulated-cost static profiling. In *FLOPS'16*, LNCS, vol. 9613. Springer, 163–180.

Hermenegildo, M., Puebla, G., Bueno, F. and García, P. L. 2005. Integrated program debugging, verification, and optimization using abstract interpretation (and The Ciao System Preprocessor). *Science of Computer Programming 58,* 1–2, 115–140.

Hermenegildo, M. V., Bueno, F., Carro, M., López, P., Mera, E., Morales, J. and Puebla, G. 2012. An overview of ciao and its design philosophy. *Theory and Practice of Logic Programming 12,* 1–2, 219–252. http://arxiv.org/abs/1102.5497.

Hoffmann, J., Aehlig, K. and Hofmann, M. 2012. Multivariate amortized resource analysis. *ACM TOPLAS 34,* 3, 14:1–14:62.

Igarashi, A. and Kobayashi, N. 2002. Resource usage analysis. In *Symposium on Principles of Programming Languages*. ACM, 331–342.

Liqat, U., Georgiou, K., Kerrison, S., Lopez-Garcia, P., Hermenegildo, M. V., Gallagher, J. P. and Eder, K. 2016. Inferring Parametric Energy Consumption Functions at Different Software Levels: ISA vs. LLVM IR. In *Proc. of FOPARA*, LNCS. Springer. In press.

Liqat, U., Kerrison, S., Serrano, A., Georgiou, K., Lopez-Garcia, P., Grech, N., Hermenegildo, M. and Eder, K. 2014. Energy consumption analysis of programs based on XMOS ISA-level models. In *Proceedings of LOPSTR'13*, LNCS, vol. 8901. Springer, 72–90.

Méndez-Lojo, M., Navas, J. and Hermenegildo, M. 2007. A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *LOPSTR 2007*. Number 4915 in LNCS. Springer-Verlag, 154–168.

Mera, E., Trigo, T., López-García, P. and Hermenegildo, M. 2011. Profiling for run-time checking of computational properties and performance debugging. In *PADL*. LNCS, vol. 6539. 38–53.

Morgan, R. G. and Jarvis, S. A. 1998. Profiling Large-Scale Lazy Functional Programs. *Journal of Functional Programing 8,* 3, 201–237.

Muthukumar, K. and Hermenegildo, M. 1992. Compile-time derivation of variable dependency using abstract interpretation. *JLP 13,* 2/3 (July), 315–347.

Navas, J., Méndez-Lojo, M. and Hermenegildo, M. 2008. Safe upper-bounds inference of energy consumption for Java Bytecode Applications. In *NASA LFM'08*. 29–32.

Navas, J., Mera, E., López-García, P. and Hermenegildo, M. 2007. User-definable resource bounds analysis for logic programs. In *Proc. of ICLP'07*. LNCS, vol. 4670. Springer, 348–363.

Nielson, F., Nielson, H. and Seidl, H. 2002. Automatic complexity analysis. In *Programming Languages and Systems*, LNCS. Springer, 243–261.

Puebla, G. and Hermenegildo, M. 1996. Optimized algorithms for the incremental analysis of logic programs. In *SAS'96*. Springer LNCS 1145, 270–284.

Rosendahl, M. 1989. Automatic complexity analysis. In *Proc. of FPCA'89*. ACM Press, 144–156.

Sansom, P. M. and Jones, S. L. P. 1995. Time and space profiling for non-strict, higher-order functional languages. In *Proc. of POPL'95*. ACM, New York, NY, USA, 355–366.

Serrano, A., Lopez-Garcia, P. and Hermenegildo, M. 2014. Resource usage analysis of logic programs via abstract interpretation using sized types. *Theory and Practice of Logic Programming, ICLP'14 Special Issue 14,* 4-5, 739–754.

Vasconcelos, P. and Hammond, K. 2003. Inferring cost equations for recursive, polymorphic and higher-order functional programs. In *IFL'03*, LNCS, vol. 3145. Springer, 86–101.

Wegbreit, B. 1975. Mechanical program analysis. *Comm. of the ACM 18,* 9, 528–539.

https://doi.org/10.1017/S1471068416000442 Published online by Cambridge University Press