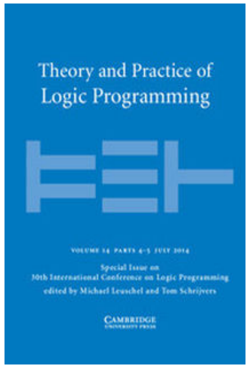

Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types

A. SERRANO, P. LOPEZ-GARCIA and M. V. HERMENEGILDO

Theory and Practice of Logic Programming / Volume 14 / Special Issue 4-5 / July 2014, pp 739 - 754
DOI: 10.1017/S147106841400057X, Published online: 21 July 2014

TLP **14** (4–5): 739–754, 2014. © Cambridge University Press 2014
doi:10.1017/S147106841400057X

*Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types**

A. SERRANO[1]†, P. LOPEZ-GARCIA[2,3] and M. V. HERMENEGILDO[3,4]
[1]*Dept. of Information and Computing Sciences, Utrecht University*
(*e-mail:* `A.SerranoMena@uu.nl`)
[2]*IMDEA Software Institute*
(*e-mail:* `pedro.lopez@imdea.org, manuel.hermenegildo@imdea.org`)
[3]*Spanish Council for Scientific Research (CSIC)*
[4]*Technical University of Madrid (UPM)*
(*e-mail:* `herme@fi.upm.es`)

submitted February 4, 2014; revised March 18, 2014; accepted May 1, 2014

Abstract

We present a novel general resource analysis for logic programs based on sized types. Sized types are representations that incorporate structural (shape) information and allow expressing both lower and upper bounds on the size of a set of terms and their subterms at any position and depth. They also allow relating the sizes of terms and subterms occurring at different argument positions in logic predicates. Using these sized types, the resource analysis can infer both lower and upper bounds on the resources used by all the procedures in a program as functions on input term (and subterm) sizes, overcoming limitations of existing resource analyses and enhancing their precision. Our new resource analysis has been developed within the abstract interpretation framework, as an extension of the sized types abstract domain, and has been integrated into the Ciao preprocessor, CiaoPP. The abstract domain operations are integrated with the setting up and solving of recurrence equations for inferring both size and resource usage functions. We show that the analysis is an improvement over the previous resource analysis present in CiaoPP and compares well in power to state of the art systems.

1 Introduction

Resource usage analysis infers the aggregation of some numerical properties (named *resources*), like memory usage, time spent in computation, or bytes sent over a wire, throughout the execution of a piece of code. The expressions giving the usage of resources are usually functions of the sizes of some input arguments to procedures.

Our starting point is the methodology outlined by (Debray *et al.* 1990; Debray and Lin 1993; Debray *et al.* 1997), characterized by the setting up of recurrence equations. In that methodology, the size analysis is the first of several other analysis steps that include, e.g., cardinality analysis (that infers lower and upper bounds on

* This research was supported in part by projects EU FP7 318337 *ENTRA*, Spanish MINECO TIN2012-39391 *StrongSoft* and TIN2008-05624 *DOVES*, and Madrid TIC/1465 *PROMETIDOS-CM*.
† A. Serrano performed this work during his former affiliation to the IMDEA Software Institute.

the number of solutions computed by a predicate), and which ultimately obtain the resource usage bounds. One drawback of these proposals, as well as most of their subsequent derivatives, is that they are able to cope with size information about subterms in a very limited way. This is an important limitation, which causes the analysis to infer trivial bounds for a large class of programs. For example, consider a predicate which computes the factorials of a list:

```
% listfact(+L, -FL).           % fact(+N, -F).
listfact([],    []).           fact(0,1).
listfact([E|R],[F|FR]) :-      fact(N,M) :- N1 is N - 1,
  fact(E, F),                               fact(N1, M1),
  listfact(R, FR).                          M is N * M1.
```

Intuitively, the best bound for the running time of this program for a list L is $c_1 + \sum_{e \in L} \left(c_2 + time_{fact}(e)\right)$, where c_1 and c_2 are constants related to unification and calling costs. But with no further information, the upper bound for the elements of L must be ∞ to be on the safe side, and then the returned overall time bound must also be ∞. In a previous paper (Serrano *et al.* 2013) we focused on a proposal to improve the size analysis based on *sized types*. While in that paper we already hinted at the fact that the application of our sized types in resource analysis could result in considerable improvement, no description was provided of the actual resource analysis. This paper is complementary and fills this gap by describing a new resource usage analysis that can *take advantage of the new information contained in sized types*. Furthermore, the resource analysis we propose is *based fully on abstract interpretation* (Cousot and Cousot 1992). Previously, the auxiliary analyses used this technique, but the core resource analysis did not use it directly. Our approach formulates the resource analysis as an *abstract domain* that can be integrated within a standard, parametric abstract interpreter. In particular, we integrate it into the PLAI abstract interpretation framework (Muthukumar and Hermenegildo 1992; Puebla and Hermenegildo 1996) of CiaoPP, obtaining features such as *multivariance*, efficient fixpoints, and assertion-based verification and user interaction for free. We also perform an assessment of the accuracy and efficiency of the resulting overall system.

In Section 2 we give a high-level view of the approach. In the following section we review the abstract interpretation approach to size analysis using sized types. Section 4 gets deeper into the resource usage analysis, our main contribution. Experimental results are shown in Section 5. Finally we review some related work and discuss future directions.

2 Overview of the Approach

We give now an overview of our approach to resource usage analysis, and present the main ideas in our proposal using the classical `append/3` predicate as a running example:

```
append([],    S, S).
append([E|R], S, [E|T]) :- append(R, S, T).
```

The process starts by performing the regular type analysis present in the CiaoPP system (Vaucheret and Bueno 2002). In our example, the system infers that for any call to the predicate `append(X, Y, Z)` with `X` and `Y` bound to lists of numbers and `Z` a free variable, if the call succeeds, then `Z` also gets bound to a list of numbers. The set of "list of numbers" is represented by the regular type *listnum*, defined as follows:

```
listnum := [] | [num | listnum].
```

From this regular type definition, sized type schemas are derived. The sized type schema *listnum-s* is derived from *listnum*. This schema corresponds to a list whose length is between α and β, containing numbers between γ and δ.

$$listnum\text{-}s \rightarrow listnum^{(\alpha,\beta)}(num^{(\gamma,\delta)})$$

From now on, in the examples we will use *ln* and *n* instead of *listnum* and *num* for the sake of conciseness. The next phase involves relating the sized types of the different arguments to the `append/3` predicate using recurrence (in)equations. Let $size_X$ denote the sized type schema for argument `X` in a call `append(X, Y, Z)` (from the regular type inferred by a previous analysis). We have that $size_X$ denotes $ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)})$. Similarly, the sized type schema for the output argument `Z` is $ln^{(\alpha_Z,\beta_Z)}(n^{(\gamma_Z,\delta_Z)})$, denoted by $size_Z$. We are interested in expressing bounds on the length of the output list `Z` and the values of its elements as a function of size bounds for the input lists `X` and `Y` (and their elements). For this, we set up a system of inequations. For instance, the inequations that are set up to express a lower bound on the length of the output argument `Z`, denoted α_Z, as a function on the size bounds of the input arguments `X` and `Y`, and their subarguments (α_X, β_X, γ_X, δ_X, α_Y, β_Y, γ_Y, and δ_Y) are:

$$\alpha_Z\begin{pmatrix}\alpha_X,\beta_X,\gamma_X,\delta_X,\\ \alpha_Y,\beta_Y,\gamma_Y,\delta_Y\end{pmatrix} \geqslant \begin{cases}\alpha_Y & \text{if } \alpha_X = 0\\ 1+\alpha_Z\begin{pmatrix}\alpha_X-1,\beta_X-1,\gamma_X,\delta_X,\\ \alpha_Y,\beta_Y,\gamma_Y,\delta_Y\end{pmatrix} & \text{if } \alpha_X > 0\end{cases}$$

Note that in the recurrence inequation set up for the second clause of `append/3`, the expression $\alpha_X - 1$ (respectively $\beta_X - 1$) represents the size relationship that a lower (respectively upper) bound on the length of the list in the first argument of the recursive call to `append/3` is one unit less than the length of the first argument in the clause head.

As the number of size variables grows, the set of inequations becomes too large. Thus, we propose a compact representation, which allows us to grasp all the relations in one view. The first change in our proposal is to write the parameters to size functions directly as sized types. Now, the parameters to the α_Z function are the sized type schemas corresponding to the arguments `X` and `Y` of the `append/3` predicate:

$$\alpha_Z \begin{pmatrix} ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)}) \\ ln^{(\alpha_Y,\beta_Y)}(n^{(\gamma_Y,\delta_Y)}) \end{pmatrix} \geqslant \begin{cases} \alpha_Y & \text{if } \alpha_X = 0 \\ 1 + \alpha_Z \begin{pmatrix} ln^{(\alpha_X-1,\beta_X-1)}(n^{(\gamma_X,\delta_X)}) \\ ln^{(\alpha_Y,\beta_Y)}(n^{(\gamma_Y,\delta_Y)}) \end{pmatrix} & \text{if } \alpha_X > 0 \end{cases}$$

In a second step, we group together all the inequalities of a single sized type. As we always alternate lower and upper bounds, it is always possible to distinguish the type of each inequality. We do not write equalities, so that we do not use the symbol =. However, we always write inequalities of both signs ($\geqslant$ and $\leqslant$) for each size function, since we compute both lower and upper size bounds. Throughout this paper we use a representation using $\lesseqgtr$ for the symbols $\geqslant$ and $\leqslant$ that are always paired. For example, the expression $ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)}) \lesseqgtr ln^{(e_1,e_2)}(n^{(e_3,e_4)})$ represents the conjunction of the following size constraints: $\alpha_X \geqslant e_1$, $\beta_X \leqslant e_2$, $\gamma_X \geqslant e_3$, $\delta_X \leqslant e_4$. In the implementation, constraints for each variable are kept apart and solved separatedly.

After setting up the corresponding system of inequations for the output argument `Z` of `append/3`, and solving it, we obtain the following expression:

$$size_Z\,(size_X, size_Y) \lesseqgtr ln^{(\alpha_X+\alpha_Y,\beta_X+\beta_Y)}(n^{(\min(\gamma_X,\gamma_Y),\max(\delta_X,\delta_Y))})$$

that represents, among others, the relation $\alpha_z \geqslant \alpha_X + \alpha_Y$ (resp. $\beta_z \leqslant \beta_X + \beta_Y$), expressing that a lower (resp. upper) bound on the length of the output list `Z`, denoted α_z (resp. β_z), is the addition of the lower (resp. upper) bounds on the lengths of `X` and `Y`. It also represents the relation $\gamma_Z \geqslant \min(\gamma_X, \gamma_Y)$ (resp. $\delta_Z \leqslant \max(\delta_X, \delta_Y)$), which expresses that a lower (resp. upper) bound on the size of the elements of the list `Z`, denoted γ_z (resp. δ_z), is the minimum (resp. maximum) of the lower (resp. upper) bounds on the sizes of the elements of the input lists `X` and `Y`.

Resource analysis builds upon the sized type analysis and adds recurrence equations for each resource we want to analyze. Apart from that, when considering logic programs, we have to take into account that they can fail or have multiple solutions when executed, so we need an auxiliary *cardinality analysis* to get correct results.

Let us focus on cardinality analysis. Let s_L and s_U denote lower and upper bounds on the number of solutions for `append/3`. Following the program structure we can infer:

$$\begin{aligned} s_L\left(ln^{(0,0)}(n^{(\gamma_X,\delta_X)}), size_Y\right) &\geqslant 1 \\ s_L\left(ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)}), size_Y\right) &\geqslant s_L\left(ln^{(\alpha_X-1,\beta_X-1)}(n^{(\gamma_X,\delta_X)}), size_Y\right) \end{aligned}$$

$$\begin{aligned} s_U\left(ln^{(0,0)}(n^{(\gamma_X,\delta_X)}), size_Y\right) &\leqslant 1 \\ s_U\left(ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)}), size_Y\right) &\leqslant s_U\left(ln^{(\alpha_X-1,\beta_X-1)}(n^{(\gamma_X,\delta_X)}), size_Y\right) \end{aligned}$$

Since $s_L \leqslant s_U$, the solution to these inequations must be $(s_L, s_U) = (1, 1)$. Thus, we have inferred that `append/3` has at least (and at most) one solution: it behaves like a function. When setting up the equations, we use the result of the non-failure analysis to see that append/3 cannot fail when given lists as arguments. If not, the lower bound is 0.

Now we move forward to analyzing the number of resolution steps performed by a call to append/3 (we will only focus on upper bounds, r_U, for brevity). For the first clause, we know that only one resolution step is needed, so:

$$r_U\left(ln^{(0,0)}(n^{(\gamma_X,\delta_X)}), ln^{(\alpha_Y,\beta_Y)}(n^{(\gamma_Y,\delta_Y)})\right) \leqslant 1$$

The second clause performs one resolution step plus all the resolution steps performed by all possible backtrackings over the call in the body of the clause. This number can be bounded as a function of the number of solutions. Thus, the equation reads:

$$\begin{aligned} r_U\left(ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)}), size_Y\right) &\leqslant 1 + s_U\left(ln^{(\alpha_X-1,\beta_X-1)}(n^{(\gamma_X,\delta_X)}), size_Y\right) \\ &\quad \times r_U\left(ln^{(\alpha_X-1,\beta_X-1)}(n^{(\gamma_X,\delta_X)}), size_Y\right) \\ &= 1 + r_U\left(ln^{(\alpha_X-1,\beta_X-1)}(n^{(\gamma_X,\delta_X)}), size_Y\right) \end{aligned}$$

Solving these equations we infer that an upper bound on the number of resolution steps is the (upper bound on) the length of the input list X plus one. This is expressed as:

$$r_U\left(ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)}), ln^{(\alpha_Y,\beta_Y)}(n^{(\gamma_Y,\delta_Y)})\right) \leqslant \beta_X + 1$$

3 Sized Types Review

As shown in the append example, the variables that we relate in our inequations come from sized types, which are ultimately derived from the regular types previously inferred for the program. Among several representations of regular types used in the literature, we use one based on *regular term grammars*, equivalent to (Dart and Zobel 1992) but with some adaptations. A *type term* is either a *base type* η_i (taken from a finite set), a *type symbol* τ_i (taken from an infinite set), or a term of the form $f(\phi_1,\ldots,\phi_n)$, where f is a n-ary function symbol (taken from an infinite set) and $\phi_1,\ldots,\phi_n$ are *type terms*. A *type rule* has the form $\tau \rightarrow \phi$, where τ is a *type symbol* and ϕ a *type term*. A *regular term grammar* Υ is a set of *type rules*.

To devise the abstract domain we focus specifically on the PLAI (Muthukumar and Hermenegildo 1989; Muthukumar and Hermenegildo 1992) framework, integrated within CiaoPP (Hermenegildo *et al.* 2012) (see the on-line Appendix A), where we have incorporated our implementation. The PLAI algorithm abstracts execution AND-OR trees similarly to (Bruynooghe 1991) but represents the abstract executions *implicitly* and computes fixpoints efficiently using memo tables, dependency tracking, etc. It takes as input a pair (L, λ_c) representing an entry point (predicate) along with an abstraction of the call patterns (in the chosen *abstract domain*) and produces an abstraction which overapproximates information at all program points (for all procedure versions).

The formal concept of *sized type* is an abstraction of a set of Herbrand terms which are a subset of some regular type τ and meet some lower- and upper-bound size constraints on the number of *type rule applications* needed to generate the terms.

A grammar for the new sized types follows:

$$
\begin{array}{rll}
\textit{sized-type} ::= & \eta^{bounds} & \eta \text{ base type} \\
 | & \tau^{bounds}(\textit{sized-args}) & \tau \text{ recursive type symbol} \\
 | & \tau(\textit{sized-args}) & \tau \text{ non-recursive type symbol} \\
\textit{bounds} ::= & nob \ | \ (n, m) & n, m \in \mathbb{N}, m \geqslant n \\
\textit{sized-args} ::= & \epsilon \ | \ \textit{sized-arg}, \ \textit{sized-args} & \\
\textit{sized-arg} ::= & \textit{sized-type}_{position} & \\
\textit{position} ::= & \epsilon \ | \ \langle f, n \rangle & f \text{ functor}, 0 \leqslant n \leqslant \text{arity of } f
\end{array}
$$

However, in our abstract domain we need to refer to sets of sized types which satisfy certain constraints on their bounds. For that purpose, we introduce *sized type schemas*: a schema is just a sized type with variables in bound positions, i.e., where n and m in the pair (n,m) defining the symbol *bounds* in the grammar above are variables (called bound variables), along with a set of constraints over those variables. We call such variables *bound variables*. We will denote $sized(\tau)$ the sized type schema corresponding to a regular type τ where all the bound variables are fresh.

The full abstract domain is an extension of sized type schemas to several predicate variables. Each abstract element is a triple $\langle t, d, r \rangle$ such that:

1. t is a set of $v \to (sized(\tau), c)$, where v is a variable, τ its regular type and c is its classification. Subgoal variables can be classified as *output*, *relevant*, or *irrelevant*. Variables appearing in the clause body but not in the head are classified as *clausal*;
2. d (the *domain*) is a set of constraints over the relevant variables;
3. r (the *relations*) is a set of relations among bound variables.

For example, the final abstract elements corresponding to the clauses of the `listfact` example can be found below. The equations have already been normalized into their simplest form, and the variables refer to the predicate arguments in normal form. $listfact$ refers implicitly to the solution of the joint equations: it is the recurrence we need to solve. In order to enhance readability, we have dropped the position element $\langle ., 1 \rangle$ from ln.

$$
\lambda'_1 = \left\langle \begin{array}{c} \{L \to (ln^{(\alpha_1,\beta_1)}(n^{(\gamma_1,\delta_1)}), rel.), FL \to (ln^{(\alpha_2,\beta_2)}(n^{(\gamma_2,\delta_2)}), out.)\} \\ \{\alpha_1 = 1, \beta_1 = 1\}, \{ln^{(\alpha_2,\beta_2)}(n^{(\gamma_2,\delta_2)}) \lesseqgtr ln^{(1,1)}(n^{nob})\} \end{array} \right\rangle
$$

$$
\lambda'_2 = \left\langle \begin{array}{c} \left\{ \begin{array}{c} L \to (ln^{(\alpha_1,\beta_1)}(n^{(\gamma_1,\delta_1)}), rel.), FL \to (ln^{(\alpha_2,\beta_2)}(n^{(\gamma_2,\delta_2)}), out.), \\ E \to (n^{(\gamma_3,\delta_3)}, cl.), R \to (ln^{(\alpha_4,\beta_4)}(n^{(\gamma_4,\delta_4)}), cl.), \\ F \to (n^{(\gamma_5,\delta_5)}, cl.), FR \to (ln^{(\alpha_6,\beta_6)}(n^{(\gamma_6,\delta_6)}), cl.) \end{array} \right\} \\ \{\alpha_1 > 0, \beta_1 > 0\}, \\ \left\{ \begin{array}{c} ln^{(\alpha_2,\beta_2)}(n^{(\gamma_2,\delta_2)}) \lesseqgtr ln^{(\alpha'+1,\beta'+1)}(n^{(\min(\gamma_1!,\gamma'),\max(\delta_1!,\delta'))}) \\ ln^{(\alpha',\beta')}(n^{(\gamma',\delta')}) \lesseqgtr listfact\left(ln^{(\alpha_1-1,\beta_1-1)}(n^{(\gamma_1,\delta_1)})\right) \end{array} \right\} \end{array} \right\rangle
$$

4 The Resources Abstract Domain

We take advantage of the added power of sized types to develop a better resource analysis which infers upper and lower bounds on the amount of resources used by each predicate as a function of the sized type schemas of the input arguments (which encode the sizes of the terms and subterms appearing in such input arguments). For this reason, the novel abstract domain for resource analysis that we have developed is tightly integrated with the sized types abstract domain. Following (Navas *et al.* 2007), we account for two places where the resource usage can be abstracted:

- When entering a clause: some resources may be needed during unification of the call (subgoal) and the clause head, the preparation of entering that clause, and any work done when all the literals of the clause have been processed. This cost, dependent on the head h, is called *head cost*, $\varphi(h)$.
- Before calling a literal q: some resources may be used to prepare a call to a body literal (e.g., constructing the actual arguments). The amount of these resources is known as *literal cost* and is represented by $\omega(q)$.

We first consider the case of estimating upper bounds on resource usages. For simplicity, assume first that we deal with predicates having a behavior that is close to functional or imperative programs, i.e., that are deterministic and do not fail. Then, we can bound the resource consumption of a clause $C \equiv p(\bar{x})\ \ {:-}\ q_1(\bar{x}_1), \ldots, q_n(\bar{x}_n)$, denoted $r_{U,clause}$:

$$r_{U,clause}(C) \leqslant \varphi(p(\bar{x})) + \textstyle\sum_{i=1}^{n} \left(\omega(q_i(\bar{x}_i)) + r_{U,pred}(q_i(\bar{x}_i))\right)$$

As in sized type analysis, the sizes of some input arguments may be explicitly computed, or, otherwise, we express them by using a generic expression, giving rise (in the case of recursive clauses) to a recurrence equation that we need to solve in order to find closed form resource usage functions.

The resource usage of a predicate, $r_{U,pred}$, depending on its input data sizes, is obtained from the resource usage of the clauses defining it, by taking the maximum of the equation expressions that meet the constraints on the input data sizes (i.e., have the same domain).

In addition, we need to deal with two extra features of logic programming:

- We may execute a literal more than once on backtracking. To bound the number of times a literal is executed, we need to know the *number of solutions* each literal (to its left) can generate. Using the information provided by cardinality analysis, the number of times a literal is executed is at most the product of the upper bound on the number of solutions, s_U, of all the previous literals in the clause. We get:

$$\begin{aligned}&r_{U,clause}\,(p(\bar{x})\ \ {:-}\ q_1(\bar{x}_1), \ldots, q_n(\bar{x}_n))\\ &\quad\leqslant \varphi(p(\bar{x})) + \textstyle\sum_{i=1}^{n} \left(\prod_{j=1}^{i-1} s_{pred}(q_j(\bar{x}_j))\right) \left(\omega(q_i(\bar{x}_i)) + r_{U,pred}(q_i(\bar{x}_i))\right)\end{aligned}$$

- Also, in logic programming more than one clause may unify with a given subgoal. In that case it is incorrect to take the maximum of the resource usages of each clause when setting up the recurrence equations (whereas this was valid in size analysis). A correct solution is to take the sum of every set of equations with a common domain, but the bound becomes then very rough. Finer-grained possibilities can be considered by using different *aggregation* procedures per resource.

Lower bounds analysis is similar, but needs to take into account the possibility of failure, which stops clause execution and forces backtracking. Basically, no resource usage should be added beyond the point where failure may happen. For this reason, in our implementation we use the non-failure analysis already present in CiaoPP. Also, the aggregation of clauses with a common domain must be different to that used in the upper bounds case. The simplest solution is to just take the minimum of the clauses. However, this again leads to very rough bounds. We will discuss lower bound aggregation later.

Cardinality Analysis. We have already discussed why cardinality analysis (which estimates bounds on the number of solutions) is instrumental in resource analysis of logic programs. We can consider the number of solutions as another resource, but, due to its importance, we treat it separately.

An upper bound on the number of solutions of a single clause could be gathered by multiplying the number of solutions of its body literals:

$$s_{U,clause}\,(p(\bar{x})\ :-\ q_1(\bar{x}_1),\ldots,q_n(\bar{x}_n)) \leqslant \textstyle\prod_{i=1}^{n} s_{U,pred}(q_i(\bar{x}_i))$$

For aggregation we need to add the equations with a common domain, to get a recurrence equation system. These equations will be solved later to get a closed form function giving an upper bound on the number of solutions.

It is important to remark that many improvements can be added to this simple cardinality analysis to make it more precise. Some of them are discussed in (Debray and Lin 1993), like maintaining separate bounds for the relation defined by the predicate and the number of solutions for a particular input, or dealing with mutually exclusive clauses by performing the max operation, instead of the addition operation when aggregating. However, our focus here is the definition of an abstract domain, and see whether a simple definition produces comparable results for the resource usage analysis.

One of the improvements we decided to include is the use of the determinacy analysis present in CiaoPP (López-García *et al.* 2010). If such analysis infers that a predicate is deterministic, we can safely set the upper bound for the number of solutions to 1.

In the case of lower bounds, we need to know for each clause whether it may fail or not. For that reason we use the non-failure analysis already present in CiaoPP (Bueno *et al.* 2004). In case of a possible failure, the lower bound on cardinality is set to 0.

The Abstract Elements. Within the PLAI abstract interpretation framework (Muthukumar and Hermenegildo 1992; Puebla and Hermenegildo 1996) an analysis is defined by the abstract elements involved in it and a set of operations. We refer the reader to the on-line Appendix A for an overview of the overall framework.

In our case, the abstract elements are derived from sized type analysis by adding some extra components. In particular:

1. The *current variable for solutions*, and *current variable for each resource.*
2. A boolean element for telling whether we have already found a failing literal.
3. An abstract element from the non-failure domain.
4. An abstract element encoding information about determinacy.

We will denote the abstract elements by $\langle (s_L, s_U), v_{resources}, failed?, d, r, nf, det\rangle$ where (s_L, s_U) are the lower and upper bound variables for the number of solutions, $v_{resources}$ is a set of pairs (r_L, r_U) giving the lower and upper bound variables for each resource, $failed?$ is a boolean element (`true` or `false`), d and r are defined as in the sized type abstract domain, and nf and det can take values `not_fails`/`fails` and `non_det`/`is_det` respectively, as explained in (López-García *et al.* 2010; Bueno *et al.* 2004). The on-line Appendix B gives some more details of the domain.

We assume that we are given the definition of a set of resources, which are fixed throughout the whole analysis process. We assume that for each resource r we have: its head cost, φ_r, which takes a clause head as parameter; its literal cost, ω_r, which takes a literal as parameter; its aggregation procedure, Γ_r, which takes the equations for each of the clauses and creates a new set of recurrence equations from them; and the default upper $\perp_{r,U}$ and lower $\perp_{r,L}$ bound on resource usage.

To better understand how the domain works, we will continue with the analysis of `listfact` that we started in the previous section. We assume that the only resource to be analyzed is the "number of resolution steps," which uses the following parameters:

$$\varphi = 1, \quad \omega = 0, \quad \Gamma_r = +, \quad (\perp_L, \perp_U) = (0, 0)$$

The $\sqsubseteq$, $\sqcup$ Operations and the $\perp$ Element. We do not have a decidable definition for $\sqsubseteq$ or $\sqcup$, because there is no general algorithm for checking the inclusion or union of sets of integers defined by recurrence relations. Instead, for the inequation components we just check whether one is a subset of another one, up to variable renaming, or perform a syntactic union of the inequations. The ordering is finished by taking the product order with the non-failure and determinacy parts. This is enough for having a correct analysis. For the bottom element, $\perp$, we first generate new variables for each of the resources and the solution. Then, we add relations between them and the default cost for each resource. For an unknown predicate, the number of solutions should be $[0, \infty)$ and it may fail. For example, the bottom element for the "number of resolution steps" resource will be:

$$\langle (s_L, s_U), \{(n_L, n_U)\}, \texttt{true}, \emptyset, \{(s_L, s_U) \lesseqgtr (0, \infty), (n_L, n_U) \lesseqgtr (0, 0)\}, \texttt{fails}, \texttt{non\_det}\rangle$$

where `fails` and `non_det` are the bottom elements of their respective domains.

The λ_{call} to β_{entry} Operation. In this operation we need to create the initial structures for handling the bounds on the number of solutions and resources. This implies the generation of fresh variables for each of them, and setting them to their initial values. In the case of the number of solutions, the initial value is 1 (which is the number of solutions generated by a fact). For a resource r, the initial value is exactly φ_r.

We will name new fresh variables by adding an integer subscript. For example, $s_{L,1,1}$ will be the first fresh variable related to the *l*ower bound on *s*olutions on *first* clause.

The addition of constraints over sized types when the head arguments are partially instantiated is inherited from the sized types domain. Finally, for the $failed?$ component, we should start with value `false`, as no literal has been executed yet, so it cannot fail.

In the `listfact` example, the entry substitutions are:

$$\beta_{entry,1} = \left\langle \begin{array}{c} (s_{L,1,1}, s_{U,1,1}), \{(n_{L,1,1}, n_{U,1,1})\}, \texttt{false}, \{\alpha_1 = 0, \beta_1 = 0\}, \\ \{(s_{L,1,1}, s_{U,1,1}) \lesseqgtr (1,1), (n_{L,1,1}, n_{U,1,1}) \lesseqgtr (1,1)\}, \texttt{not\_fails}, \texttt{is\_det} \end{array} \right\rangle$$

$$\beta_{entry,2} = \left\langle \begin{array}{c} (s_{L,2,1}, s_{U,2,1}), \{(n_{L,2,1}, n_{U,2,1})\}, \texttt{false}, \{\alpha_1 > 0, \beta_1 > 0\}, \\ \{(s_{L,2,1}, s_{U,2,1}) \lesseqgtr (1,1), (n_{L,2,1}, n_{U,2,1}) \lesseqgtr (1,1)\}, \texttt{not\_fails}, \texttt{is\_det} \end{array} \right\rangle$$

The Extend Operation. In the *extend* operation we get both the current abstract substitution and the substitution from the literal call. We need to update several components of the abstract element. First of all, we need to include a call to the function giving the number of solutions and the resource usage from the called literal.

Afterwards, we need to generate new variables for the number of solutions and resources, which will hold the bounds for the clause up to that point. New relations must be added to the abstract element to give a value to those new variables:

- For the number of solutions, let $s_{U,c}$ be the new upper bound variable, $s_{U,p}$ the previous variable defining an upper bound on the number of solutions, and $s_{U,\lambda}$ an upper bound on the number of solutions for the subgoal. Then we need to include a constraint: $s_{U,c} \leqslant s_{U,p} \times s_{U,\lambda}$.
  In the case of lower bound analysis, there are two phases. First of all, we check whether the called literal can fail, looking at the output of the non-failure analysis. If it is possible for it to fail, we update the $failed?$ component of the abstract element to `true`. If after this checking the $failed?$ component is still `false` (meaning that neither this literal nor any of the previous ones may fail) we include a relation similar to the one for the upper bound case: $s_{L,c} \geqslant s_{L,p} \times s_{L,\lambda}$. Otherwise, we include the relation $s_{L,c} \geqslant 0$, because failing predicates produce no solutions.
- The approach for resources is similar. Let $r_{U,c}$ be the new upper bound variable, $r_{U,p}$ the previous variable defining an upper bound on that resource and $r_{U,\lambda}$ an upper bound on resources from the analysis of the literal. The relation added in this case is $r_{U,c} \leqslant r_{U,p} + s_{U,p} \times \left(\omega + r_{U,\lambda}\right)$.
  For lower bounds, we have already updated the $failed?$ component, so we only have to work in consequence. If the component is still `false`, we add a new relation similar to the one for upper bounds. If it is `true`, it means that failure may happen at some point, so we do not have to add that resource any more. Thus the relation to be included is $r_{L,c} \geqslant r_{L,p}$.

In our example, consider the extension of `listfact` after performing the analysis of the `fact` literal, whose resource components of the abstract element will be:

$$\left\langle \begin{array}{c} (s_L, s_U), \{(n_L, n_U)\}, \texttt{false}, \{\alpha, \beta \geqslant 0\} \\ \{(s_L, s_U) \lesseqgtr (1,1), (n_L, n_U) \lesseqgtr (\alpha, \beta)\}, \texttt{not\_fails}, \texttt{is\_det} \end{array} \right\rangle$$

This literal is known not to fail, so we do not change the value of *failed*? in our abstract element for the second clause. That means that it is still `false`, so we add complete calls:

$$\beta_{entry,2} = \left\langle \begin{array}{c} (s_{L,2,2}, s_{U,2,2}), \{(n_{L,2,2}, n_{U,2,2})\}, \texttt{false}, \{\dots\} \\ \left\{ \begin{array}{c} \dots, \\ (s_{L,2,2}, s_{U,2,2}) \lesseqgtr (1 \times s_{L,2,1}, 1 \times s_{U,2,1}), \\ (n_{L,2,2}, n_{U,2,2}) \lesseqgtr (\gamma_1 + n_{L,2,1}, \delta_1 + n_{U,2,1}) \end{array} \right\}, \\ \texttt{not\_fails}, \texttt{is\_det} \end{array} \right\rangle$$

The β_{exit} to λ' Operation. After all the extend operations, the variables appearing in the number of solutions and resources positions will hold the correct value for their properties. As we did with sized types, we follow now a normalization step, based on (Debray and Lin 1993): replace each variable appearing in an expression with its definition in terms of other variables, in reverse topological order. Following this process, we should reach the variables in the sized types of the input parameters in the head.

Going back to `listfact`, the final substitutions are as follows. s'_L, s'_U, n'_L and n'_U refer to number of solutions and resolution steps from the recursive call to `listfact`.

$$\lambda'_1 = \left\langle \begin{array}{c} (s_{L,1,1}, s_{U,1,1}), \{(n_{L,1,1}, n_{U,1,1})\}, \texttt{false}, \{\alpha_1 = 0, \beta_1 = 0\}, \\ \{(s_{L,1,1}, s_{U,1,1}) \lesseqgtr (1,1), (n_{L,1,1}, n_{U,1,1}) \lesseqgtr (1,1)\}, \texttt{not\_fails}, \texttt{is\_det} \end{array} \right\rangle$$

$$\lambda'_{entry,2} = \left\langle \begin{array}{c} (s_{L,2,3}, s_{U,2,3}), \{(n_{L,2,3}, n_{U,2,3})\}, \texttt{false}, \{\alpha_1 > 0, \beta_1 > 0\}, \\ \left\{ \begin{array}{c} s_{L,2,3} \geqslant 1 \times s'_L(ln^{(\alpha_1-1,\beta_1-1)}(n^{(\gamma_1,\delta_1)})), \\ s_{U,2,3} \leqslant 1 \times s'_U(ln^{(\alpha_1-1,\beta_1-1)}(n^{(\gamma_1,\delta_1)})), \\ n_{L,2,3} \geqslant \gamma_1 + n'_L(ln^{(\alpha_1-1,\beta_1-1)}(n^{(\gamma_1,\delta_1)})), \\ n_{U,2,3} \leqslant \delta_1 + n'_U(ln^{(\alpha_1-1,\beta_1-1)}(n^{(\gamma_1,\delta_1)})) \end{array} \right\}, \\ \texttt{not\_fails}, \texttt{is\_det} \end{array} \right\rangle$$

The Widening Operator ∇ and Closed Forms. As mentioned before, in contrast to previous cost analyses, at this point we bring in the possibility of different aggregation operators. Thus, when we have the equations, we need to pass them to each of the corresponding Γ_r per each resource r to get the final equations.

This process can be further refined in the case of solution analysis, using the information from the non-failure and determinacy analyses. If the final output of the non-failure analysis is `fails`, we know that the only correct lower bound is 0. So we can just assign the relation $s_L \geqslant 0$ without further relations. Conversely, if the final output of the determinacy analysis is `is_det`, we can safely set the relation $s_U \leqslant 1$, because at most one solution will be produced in each case. Furthermore, we can refine the lower bound on the number of solutions with the minimum between the current bound and 1.

In the example analyzed above there was an implicit assumption while setting up the relations: that the recursive call in the body of `listfact` refers to the same predicate call, so we can set up a recurrence. This fact is implicitly assumed in Hindley-Milner type systems. But in logic programming it is usual for a predicate to be called with different patterns (for example, modes). Fortunately, the CiaoPP framework allows multivariance (support for different call patterns of the same predicate). For the analysis to handle it, we cannot just add calls with the bare name of the predicate, because it will conflate all the versions. The solution is to add a new component to the abstract element: a random name given to the specific instance of the predicate, and generated in the λ_{call} to β_{entry}. In the widening step, all different versions of the same predicate are conflated.

Even though the analysis works with relations, these are not as useful as functions defined without recursion or calls to other functions. First of all, developers will get a better idea of the sizes presented in such a closed form. Second, functions are amenable to comparison as outlined in (López-García *et al.* 2010), which is essential in verification. There are several packages able to get bounds for recurrence equations: computer algebra systems, such as Mathematica (which has been used in our experiments) or Maxima; and specialized solvers such as PURRS (Bagnara *et al.* 2005) or PUBS (Albert *et al.* 2011). In our implementation we apply this overapproximation operator after each widening. For our example, the final abstract substitution is:

$$\lambda_1' \nabla \lambda_2' = \left\langle \begin{array}{c} (s_L, s_U), \{(n_L, n_U)\}, \mathtt{false}, \{\alpha_1, \beta_1 \geqslant 0\}, \\ \{(s_L, s_U) \lesseqgtr (1,1), (n_L, n_U) \lesseqgtr (\alpha_1\gamma_1, \beta_1\delta_1)\}, \mathtt{not\_fails}, \mathtt{is\_det} \end{array} \right\rangle$$

5 Experimental Results

We have constructed a prototype implementation in Ciao by defining the abstract operations for sized type and resource analysis that we have described and plugging them into CiaoPP's PLAI. Our objective is to assess the gains in precision in resource analysis.

Table 1 shows the results of the comparison between the new lower (***LB***) and upper bound (***UB***) resource analyses implemented in CiaoPP, which also use the new size analysis (columns *New*), and the previous resource analyses in CiaoPP (Debray and Lin 1993; Debray *et al.* 1997; Navas *et al.* 2007) (columns *Prev.*). We also compare (for upper bounds) with *RAML* (Hoffmann *et al.* 2012). Although the new resource analysis and the previous one infer concrete resource usage bound functions, for the sake of conciseness and to make the comparison with RAML meaningful, Table 1 only shows the complexity orders of such functions, e.g., if the analysis infers the resource usage bound function Φ, and $\Phi \in \Theta(\Psi)$, Table 1 shows Ψ. The parameters of such functions are (lower or upper) bounds on input data sizes. The symbols used to name such parameters have been chosen assuming that lists of numbers L_i have size $ln^{(\alpha_i,\beta_i)}(n^{(\gamma_i,\delta_i)})$, lists of lists of lists of numbers have size

Table 1. *Experimental results.*

Program	***Resource A. (LB)***			***Resource A. (UB)***					***A. Times (s)***	
	New	*Prev.*		*New*	*Prev.*		*RAML*		*New*	*Prev.*
`append`	α	α	$=$	β	β	$=$	β	$=$	0.999	0.530
`appendAll2`	$a_1a_2a_3$	a_1	$+$	$b_1b_2b_3$	∞	$+$	$b_1b_2b_3$	$=$	2.408	0.668
`coupled`	μ	0	$+$	ν	∞	$+$	ν	$=$	1.365	0.644
`dyade`	$\alpha_1\alpha_2$	$\alpha_1\alpha_2$	$=$	$\beta_1\beta_2$	$\beta_1\beta_2$	$=$	$\beta_1\beta_2$	$=$	1.658	0.620
`erathos`	α	α	$=$	β^2	β^2	$=$	β^2	$=$	2.251	0.772
`fib`	ϕ^μ	ϕ^μ	$=$	ϕ^ν	ϕ^ν	$=$	infeasible	$+$	1.064	0.671
`hanoi`	1	0	$+$	2^ν	∞	$+$	infeasible	$+$	0.819	0.603
`isort`	α^2	α^2	$=$	β^2	β^2	$=$	β^2	$=$	1.675	0.617
`isortlist`	a_1^2	a_1^2	$=$	$b_1^2b_2$	∞	$+$	$b_1^2b_2$	$=$	2.546	0.669
`listfact`	$\alpha\gamma$	α	$+$	$\beta\delta$	∞	$+$	unknown	?	1.387	0.644
`listnum`	μ	μ	$=$	ν	ν	$=$	unknown	?	1.189	0.581
`minsort`	α^2	α	$+$	β^2	β^2	$=$	β^2	$=$	1.938	0.671
`nub`	a_1	a_1	$=$	$b_1^2b_2$	∞	$+$	$b_1^2b_2$	$=$	3.614	0.910
`partition`	α	α	$=$	β	β	$=$	β	$=$	1.698	0.647
`zip3`	$\min(\alpha_i)$	0	$+$	$\min(\beta_i)$	∞	$+$	β_3	$+$	2.484	0.570

$llln^{(a_1,b_1)}(lln^{(a_2,b_2)}(ln^{(a_3,b_3)}(n^{(a_4,b_4)})))$, and numbers have size $n^{(\mu,\nu)}$. The calling modes are the usual ones with the last argument as output.

Table 1 includes columns with symbols summarizing whether the new CiaoPP resource analysis improves on the previous one and *RAML*'s: $+$ (resp. $-$) indicates more (resp. less) precise bounds, and $=$ the same. The new resource analysis improves on CiaoPP's previous analysis. Moreover, RAML can only infer polynomial costs, while our approach is able to infer other types of functions, as shown for the divide-and-conquer benchmarks `hanoi` and `fib`, which represent a common class of programs. For predicates with polynomial cost, we get equal or better results than RAML.

The last two columns show the times (in seconds) required by both lower and upper bound analysis together for the new resource analysis, and for the previous resource analysis in CiaoPP (Ciao/CiaoPP version 1.15-2124-ga588643, on an Intel Core i7 2.4 GHz, 8 GB 1333 MHz DDR3 memory, running MAC OS X Lion 10.7.5). These times include also the auxiliary non-determinism and failure analyses. The resulting times are encouraging, despite the currently relatively inefficient implementation of the interface with the Mathematica system which is used for solving recurrence equations.

6 Related Work

Several other analyses for resources have been proposed in the literature. Some of them just focus on one particular resource (usually execution or heap consumption), but it seems clear that they could be generalized. We already mentioned RAML (Hoffmann *et al.* 2012) in Section 5. Their approach differs from ours in the theoretical framework being used: RAML uses a type and effect system, whereas we use abstract interpretation. Another difference is the use of polynomials in RAML, which allows a complete method of resolution but limits the type of closed forms that can be analyzed. In contrast, we use recurrence equations, which have no complete decision procedure, but encompass a much larger class of functions. Type systems are also used to guide inference in (Grobauer 2001) and (Igarashi and Kobayashi 2002). In (Nielson *et al.* 2002), the authors use sparsity information to infer asymptotic complexities, instead of recurrences. (Giesl *et al.* 2012) uses symbolic evaluation graphs to derive termination and complexity properties. The recurrence equation approach was proposed originally by Wegbreit (Wegbreit 1975). Similarly to CiaoPP's previous analysis, the approach of (Albert *et al.* 2011) applies the recurrence equation method directly (i.e., not within an abstract interpretation framework). (Rosendahl 1989) shows a complexity analysis based on abstract interpretation over a step-counting version of functional programs, but which does not generate closed forms. Types with embedded size information have also been proposed by (Vasconcelos and Hammond 2003) for functional programs. Our sized type analysis is based on regular types and abstract interpretation, and deals with the logic programming features such as unification, non-determinism, and backtracking.

7 Conclusions

We have presented a new formulation of resource analysis as a domain within abstract interpretation and which uses as input information the sized types that we developed in (Serrano *et al.* 2013). Our approach overcomes important limitations of existing resource analyses and enhances their precision. It also benefits from an easier implementation and integration within an abstract interpretation framework such as PLAI/CiaoPP, which brings in useful features such as *multivariance* for free. Finally, the results of our experimental assessment regarding accuracy and efficiency are quite encouraging.

References

Albert, E., Genaim, S., and Masud, A. N. 2011. More Precise yet Widely Applicable Cost Analysis. In *12th Verification, Model Checking, and Abstract Interpretation (VMCAI'11)*, R. Jhala and D. Schmidt, Eds. Lecture Notes in Computer Science, vol. 6538. Springer Verlag, 38–53.

Bagnara, R., Pescetti, A., Zaccagnini, A., and Zaffanella, E. 2005. PURRS: Towards Computer Algebra Support for Fully Automatic Worst-Case Complexity Analysis. Tech. rep. `arXiv:cs/0512056` available from `http://arxiv.org/`.

Bruynooghe, M. 1991. A practical framework for the abstract interpretation of logic programs. *J. Log. Program. 10,* 2, 91–124.

Bueno, F., López-García, P., and Hermenegildo, M. 2004. Multivariant Non-Failure Analysis via Standard Abstract Interpretation. In *7th International Symposium on Functional and Logic Programming (FLOPS 2004)*. Number 2998 in LNCS. Springer-Verlag, Heidelberg, Germany, 100–116.

Cousot, P. and Cousot, R. 1992. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming 13,* 2-3, 103–179.

DART, P. AND ZOBEL, J. 1992. A Regular Type Language for Logic Programs. In *Types in Logic Programming*. MIT Press, 157–187.

DEBRAY, S. K. AND LIN, N. W. 1993. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems 15,* 5 (November), 826–875.

DEBRAY, S. K., LIN, N.-W., AND HERMENEGILDO, M. 1990. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation.* ACM Press, 174–188.

DEBRAY, S. K., LÓPEZ-GARCÍA, P., HERMENEGILDO, M., AND LIN, N.-W. 1997. Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium.* MIT Press, Cambridge, MA, 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 *PPDP*. ACM, 1–12.

GROBAUER, B. 2001. Cost recurrences for DML programs. In *International Conference on Functional Programming*. 253–264.

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 (January), 219–252. http://arxiv.org/abs/1102.5497.

HOFFMANN, J., AEHLIG, K., AND HOFMANN, M. 2012. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst. 34,* 3, 14.

IGARASHI, A. AND KOBAYASHI, N. 2002. Resource usage analysis. In *Symposium on Principles of Programming Languages*. 331–342.

LÓPEZ-GARCÍA, P., BUENO, F., AND HERMENEGILDO, M. 2010. Automatic Inference of Determinacy and Mutual Exclusion for Logic Programs Using Mode and Type Information. *New Generation Computing 28,* 2, 117–206.

LÓPEZ-GARCÍA, P., DARMAWAN, L., AND BUENO, F. 2010. A Framework for Verification and Debugging of Resource Usage Properties. In *Technical Communications of the 26th Int'l. Conference on Logic Programming (ICLP'10)*, M. Hermenegildo and T. Schaub, Eds. Leibniz International Proceedings in Informatics (LIPIcs), vol. 7. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 104–113.

MUTHUKUMAR, K. AND HERMENEGILDO, M. 1989. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*. MIT Press, 166–189.

MUTHUKUMAR, K. AND HERMENEGILDO, M. 1992. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming 13,* 2/3 (July), 315–347.

NAVAS, J., MERA, E., LÓPEZ-GARCÍA, P., AND HERMENEGILDO, M. 2007. User-Definable Resource Bounds Analysis for Logic Programs. In *23rd International Conference on Logic Programming (ICLP'07)*. Lecture Notes in Computer Science, vol. 4670. Springer.

NIELSON, F., NIELSON, H. R., AND SEIDL, H. 2002. Automatic complexity analysis. In *European Symposium on Programming*. 243–261.

PUEBLA, G. AND HERMENEGILDO, M. 1996. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *International Static Analysis Symposium (SAS 1996)*. Number 1145 in LNCS. Springer-Verlag, 270–284.

ROSENDAHL, M. 1989. Automatic Complexity Analysis. In *4th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*. ACM Press.

SERRANO, A., LOPEZ-GARCIA, P., BUENO, F., AND HERMENEGILDO, M. 2013. Sized Type Analysis for Logic Programs (technical communication). In *Theory and Practice of Logic Programming, 29th Int'l. Conference on Logic Programming (ICLP'13) Special Issue, On-line Supplement*, T. Swift and E. Lamma, Eds. Vol. 13 (4-5). Cambridge U. Press, 1–14.

Vasconcelos, P. B. and Hammond, K. 2003. Inferring cost equations for recursive, polymorphic and higher-order functional programs. In *IFL*, P. W. Trinder, G. Michaelson, and R. Pena, Eds. Lecture Notes in Computer Science, vol. 3145. Springer, 86–101.

Vaucheret, C. and Bueno, F. 2002. More Precise yet Efficient Type Inference for Logic Programs. In *International Static Analysis Symposium.* Lecture Notes in Computer Science, vol. 2477. Springer-Verlag, 102–116.

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