Sized Type Analysis for Logic Programs

A. SERRANO[1] P. LOPEZ-GARCIA[1,2] F. BUENO[3] M. V. HERMENEGILDO[1,3] *

[1] *IMDEA Software Institute*
(*e-mail:* alejandro.serrano@imdea.org, pedro.lopez@imdea.org, manuel.hermenegildo@imdea.org)
[2] *Spanish Council for Scientific Research (CSIC)*
[3] *Universidad Politécnica de Madrid (UPM)*
(*e-mail:* bueno@fi.upm.es, herme@fi.upm.es)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

We present a novel analysis for relating the sizes of terms and subterms occurring at different argument positions in logic predicates. We extend and enrich the concept of *sized type* as a representation that incorporates structural (shape) information and allows expressing both lower and upper bounds on the size of a set of terms and their subterms at any position and depth. For example, expressing bounds on the length of lists of numbers, together with bounds on the values of all of their elements. The analysis is developed using abstract interpretation and the novel abstract operations are based on setting up and solving recurrence relations between sized types. It has been integrated, together with novel resource usage and cardinality analyses, in the abstract interpretation framework in the Ciao preprocessor, CiaoPP, in order to assess both the accuracy of the new size analysis and its usefulness in the resource usage estimation application. We show that the proposed sized types are a substantial improvement over the previous size analyses present in CiaoPP, and also benefit the resource analysis considerably, allowing the inference of equal or better bounds than comparable state of the art systems.

1 Introduction

Size analysis is the process of assigning numerical metrics to terms appearing in a program and estimating bounds for these metrics. Such analysis is useful on its own as a source of information for the developer, and it is also often instrumental to other analyses. For example, the consumption of resources, such as memory or time, by a program is usually expressed in terms of the sizes of its arguments. In this paper we focus on the size analysis of Prolog terms. Our starting point is the methodology outlined by (Debray et al. 1990; Debray and Lin 1993) and (Debray et al. 1997), characterized by the setting up of recurrence equations. There, the size analysis is the first of several other analysis steps ultimately arriving at cost bounds. An important limitation of that analysis is that it is only able to cope with size information about subterms in a very limited way. However, dealing fully with subterms is in fact a key issue in the cost analysis of realistic programs. For example, consider a predicate which computes the factorials of a list:

* 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*.

```
listfact([],     []).
listfact([E|R], [F|FR]) :-
  fact(E, F),
  listfact(R, FR).
```

```
fact(0, 1).
fact(N, M) :- N1 is N - 1,
              fact(N1, M1),
              M is N * M1.
```

Intuitively, the best bound for the running time over a list L is $\alpha+\sum_{e\in L}(\beta + time_{fact}(e))$ where α and β are constants related to the unification and calling costs. However, 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 ∞.

Several authors have worked to overcome this limitation. In (Hoffmann et al. 2012) a system is proposed which is able to analyze `listfact`. This is done within the framework of amortized analysis with the potential method, enriched with fixed polynomials relating the cost with sizes of contained elements. However, polynomials are not enough for expressing some kinds of bounds, especially exponential ones.

In (Vasconcelos and Hammond 2003) the authors introduce the idea of *sized types* to directly represent information about the upper bounds on sizes within a Hindley-Milner type system, for functional programs. Our proposal of *sized types* is related to this idea, but differs from it in several significant ways:

- We incorporate structural (shape) information expressing *both lower and upper bounds* on the sizes of a set of terms and their subterms, *at any depth*.
- We focus on *logic programming*, which includes features such as non-determinism and creation of terms without previously having to define the constructors involved.
- Instead of a Hindley-Milner type system, we use *regular types* (Dart and Zobel 1992) as the base for sized types. Regular types are structural instead of nominal, and there is a notion of subtyping based on inclusion, important differences that the analysis must handle. Furthermore, the sized types are *automatically derived*.
- We develop the analysis as an *abstract interpretation* instead of a type and effect system. To our knowledge, this is the first time a recurrence-based analysis is developed entirely as an abstract domain. Using abstract interpretation enables us to integrate the analysis in a standard engine (in our case PLAI within the CiaoPP analysis framework), which brings in features such as *multivariance*, accelerated fixpoint computation, and assertion-based verification and user interaction for free.
- (Vasconcelos and Hammond 2003) allows assigning costs to higher-order functions based on the cost of other functions. Our system does not yet allow this, but we believe the extension is not complex.

2 Overview of the Approach

We show the different ideas in our proposal using the classical `append` predicate:

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

In a first phase we infer types for the predicate arguments by using an existing analysis for regular types (Vaucheret and Bueno 2002). This analysis infers for instance that if we call `append(X, Y, Z)` with `X` and `Y` bound to lists of numbers and `Z` a free variable, then `Z` gets bound to a list of numbers upon success.

Even more importantly, the definition of the *inferred* regular type is the following:

```
listnum -> [] | .(num, listnum)
```

From this inferred definition, or any other expressed as a regular type, we derive the *schema* of the corresponding sized type. Such sized types represent the size of a particular term, i.e., in our case, the sized type *listnum-s*:

$$listnum\text{-}s \rightarrow listnum^{(\alpha,\beta)}(num^{(\gamma,\delta)}_{\langle .,1\rangle})$$

represents that the list has between α and β elements which are numbers between γ and δ. The $\langle .,1\rangle$ below *num* expresses that this inner size description applies to subterms occurring at the first parameter of the `./2` functor.

The next phase involves relating the sized types of the different arguments to the predicate using recurrences. Let[1] $size_X = ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)}_{\langle .,1\rangle})$ be the sized type of a list X of numbers. Assume a call `append(X, Y, Z)`. The inequations for the lower bound on the length of the output argument Z, denoted α_Z, as a function on input data sizes are:

$$\alpha_Z\begin{pmatrix}\alpha_X,\beta_X,\gamma_X,\delta_X,\\ \alpha_Y,\beta_Y,\gamma_Y,\delta_Y\end{pmatrix} \geq \begin{cases}\alpha_Y & \text{if } \alpha_X = 0 \ (first\ clause)\\ 1+\alpha_Z\begin{pmatrix}\alpha_X-1,\beta_X-1,\gamma_X,\delta_X,\\ \alpha_Y,\beta_Y,\gamma_Y,\delta_Y\end{pmatrix} & \text{if } \alpha_X > 0 \ (second\ clause)\end{cases}$$

The whole set of inequations defining all bounds on a sized type is too large. Thus, we aim for a more compact representation. Our proposal is to write parameters directly as sized types and group all inequalities (both upper and lower bounds) on a single type in a expression. We decided to use the symbol $\lesseqgtr$ to mean that both types of inequalities are represented. For example:

$$ln^{(a_1,b_1)}(n^{(c_1,d_1)}) \lesseqgtr ln^{(a_2,b_2)}(n^{(c_2,d_2)}) \iff a_1 \geq a_2, b_1 \leq b_2, c_1 \geq c_2, d_1 \leq d_2$$

Using this syntax, the tightest bounds on the entire recurrence relation are:

$$size_Z\left(ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)}_{\langle .,1\rangle}), ln^{(\alpha_Y,\beta_Y)}(n^{(\gamma_Y,\delta_Y)}_{\langle .,1\rangle})\right) \lesseqgtr ln^{(\alpha_X+\alpha_Y,\beta_X+\beta_Y)}(n^{(\min(\gamma_X,\gamma_Y),\max(\delta_X,\delta_Y))}_{\langle .,1\rangle})$$

3 Sized Types

As shown in the `append` example, the variables we relate in our inequations come from sized types. *Sized types* are representations for summarizing the size of a set of terms, close to those found in (Hughes et al. 1996) for functional languages. In our approach, *sized types* schemas are automatically built from automatically inferred regular types by analyses present in the CiaoPP system (Vaucheret and Bueno 2002). 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* α_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*.

In this paper, we introduce the concept of *sized type* as an abstraction of a set of Herbrand terms that: 1) are a subset of a set abstracted by some regular type τ, and 2) meet

[1] In the examples we will use *ln* and *n* instead of *listnum* and *num* for the sake of conciseness.

$$
\begin{array}{rcll}
\gamma\left(num^{(\alpha,\beta)}\right) & = & \{n \in \mathbb{Z} : \alpha \leq n \leq \beta\} & \\
\gamma\left(\tau^{(\alpha,\beta)}(\bar{x})\right) & = & \bigcup_{\alpha \leq s \leq \beta} \gamma_{exact}\left(\tau^{s}(\bar{x})\right), & \text{if } \tau \text{ is recursive} \\
\gamma\left(\tau(\bar{x})\right) & = & \gamma_{exact}\left(\tau^{1}(\bar{x})\right), & \text{if } \tau \text{ is not recursive} \\
\gamma\left(\tau^{nob}(\bar{x})\right) & = & \emptyset &
\end{array}
$$

$$
\begin{array}{rcl}
\gamma_{exact}\left(\tau^{0}(\bar{x})\right) & = & \emptyset \\
\gamma_{exact}\left(\tau^{s}(\bar{x})\right) & = & \bigcup_{\tau \to \phi \in \Phi} \gamma_{rule}(\phi, \bar{x}, \tau^{s}), \quad s > 0
\end{array}
$$

$$
\begin{array}{rcll}
\gamma_{rule}\left(\sigma, d, \tau^{s}\right) & = & \gamma(d), \quad \text{if } \sigma \text{ is a type symbol} & \\
\gamma_{rule}\left(f(\sigma_1, \ldots, \sigma_n), \bar{x}, \tau^{s}\right) & = & \{f(y_1, \ldots, y_n) : \sum a_i = s - 1\}, & f \text{ functor} \\
\text{where} & & y_i = \begin{cases} \gamma(d), & \sigma_i \neq \tau \text{ and } d_{\langle f,i \rangle} \in \bar{x} \\ \gamma_{exact}\left(\tau^{a_i}(\bar{x})\right), & \sigma_i = \tau \end{cases} &
\end{array}
$$

Fig. 1. Concretization function γ for sized types.

some lower- and upper-bound size constraints on the number of *type rule applications* (or other metrics for base types). A grammar for the these sized types follows:

sized-type	::=	α^{bounds}	α base type
	|	$\tau^{bounds}(\textit{sized-args})$	τ recursive type symbol
	|	$\tau(\textit{sized-args})$	τ non recursive type symbol
bounds	::=	nob | (n, m)	$n, m \in \mathbb{N}, m \geq n$
sized-args	::=	ϵ | *sized-arg*, *sized-args*	
sized-arg	::=	$\textit{sized-type}_{position}$	
position	::=	ϵ | $\langle f, n \rangle$	f functor, $0 \leq n \leq$ arity of f

We say that n and m appearing in the *bounds* element of this grammar are in *bound positions*. The concretization function γ given in Figure 1 takes a *sized type* and returns the set of terms defined by it. Note that for each type appearing in the right hand side of a type rule, we include its sized type along with the position (functor and place) where it appears. In the case of top level types we use ϵ. We use *nob* as a value for *bounds* to prevent the application of a specific type rule.

Other approaches, e.g., the one proposed for CASLOG (Debray et al. 1990; Debray and Lin 1993) and previous CiaoPP analyses (López-García et al. 1996; Navas et al. 2007), use a predefined set of size metrics, such as the actual value of a number, the length of a list, or term depth. In addition, the developer can create new metrics. We only use type rule applications to bound compound terms. This is not a limitation, since most useful metrics can be expressed or bounded as sized types. For example, the size of a list of between a and b elements is $list^{(a+1,b+1)}$ (we have to include the extra `[]`) and the depth of a term is bounded by the sum of all numbers appearing in the bound positions.

Sized Type Schemas In our abstract domain, we need to refer to sets of sized types which satisfy certain conditions on their bounds. For that purpose, we introduce *sized type schemas*: a schema is just a sized type with variables in bound positions, along with

$$
\begin{array}{rcll}
sized(num) & = & num^{(\alpha,\beta)}, & \alpha \text{ and } \beta \text{ fresh} \\
sized(\tau) & = & \tau^{(\alpha,\beta)}\left(sized\text{-}args(\tau)\right), & \tau \text{ recursive}, \alpha \text{ and } \beta \text{ fresh} \\
sized(\tau) & = & \tau\left(sized\text{-}args(\tau)\right), & \tau \text{ not recursive}
\end{array}
$$

$$
sized\text{-}args(\tau) = \bigcup_{\tau \to \phi \in \Phi} sized\text{-}rule(\phi, \tau)
$$

$$
\begin{array}{rcll}
sized\text{-}rule(\sigma, \tau) & = & \emptyset, & \sigma \sqsubseteq \tau \\
sized\text{-}rule(\sigma, \tau) & = & \{d_\epsilon : d = sized(\sigma)\}, & \sigma \not\sqsubseteq \tau \\
sized\text{-}rule(f(\sigma_1, \ldots, \sigma_n), \tau) & = & \bigcup\{d_{\langle f,i\rangle} : d \in sized(\sigma_i), \sigma_i \not\sqsubseteq \tau\} &
\end{array}
$$

Fig. 2. Sized type schema $sized(\tau)$ for a regular type τ.

a set of constraints over those variables. We call such variables *bound variables*. Given a schema s_i, the set of bound variables appearing in it is denoted $vars(s_i)$.

For each regular type, we can compute a sized type schema representing the same set of terms: basically a schema without any constraints on the variables. The algorithm in Figure 2 generates the mentioned schema for a type τ. Basically, it traverses the set of rules while keeping track of the last type seen in order to detect where recursion appears in type rules. If we apply it to the type:

```
nonemptylistnum -> .(num, listnum)
listnum -> [] | .(num, listnum)
```

we get as sized type schema: $nonemptylistnum\left(num^{(\alpha,\beta)}_{\langle .,1\rangle}, listnum^{(\gamma,\delta)}_{\langle .,2\rangle}(num^{(\mu,\nu)}_{\langle .,1\rangle})\right)$

4 The Abstract Domain

To devise the abstract domain we focus specifically on the geneic AND-OR trees procedure of (Bruynooghe 1991), with the optimizations of (Muthukumar and Hermenegildo 1992). This procedure is generic and goal dependent: it takes as input a pair (L, λ_c) representing a predicate along with an abstraction of the call patterns in the chosen *abstract domain* and produces and abstraction λ_o which overapproximates the possible outputs, as well as all different call/success pattern pairs for all called predicates in all paths in the program and the corresponding abstract information at all other program points. This procedure is the basis of the PLAI abstract analyzer found in CiaoPP (Hermenegildo et al. 2012), where we have integrated a working implementation of the proposed analysis.

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

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 bound variables of relevant variables;
3. r (the *relations*) is a set of relations among bound variables.

The analysis will try to infer a functional relation for the size of output variables in terms of the sizes of relevant variables $\textit{output variable} \leqslant f(\textit{relevant variables})$.

The concretization $\tilde{\gamma}$ of the abstract elements comes from that of sized types: we just need to select the subset of the terms for which domain constraints and relations hold.

$$\tilde{\gamma}(\langle\{v_i \to (s_i, c_i)\}, d, r\rangle) = \left\{ \bigcup\{v_i \to t_i\} \;\middle|\; \begin{array}{l} t_i \in \gamma(s_i(\bar{m}_i)), \bar{v}_i = vars(t_i), \\ \bar{v} = (\bar{v}_1, \ldots, \bar{v}_n), \bar{m} = (\bar{m}_1, \ldots, \bar{m}_n), \\ \bar{v} = \bar{m} \models d(\bar{v}) \wedge r(\bar{v}) \end{array} \right\}$$

As mentioned before, the analysis comprises two stages. The first stage involves running a regular (and moded) type analysis over the program, done in our implementation using (Vaucheret and Bueno 2002). In a second stage we feed this information to the proposed size analysis, which takes such types as fixed, and computes an overapproximation of the least fixpoint for the set of domains and relations. We will now look at each of the operations that define this second stage as an abstract domain in CiaoPP's setting. At the same time we will analyze our initial "list of factorials" example.

4.1 $\sqsubseteq$, $\sqcup$ and $\bot$

As mentioned before, these three operations are needed to define the abstract domain correctly as a join-semilattice and for the computation of fixpoints in the analysis (Cousot and Cousot 1992). One important remark here is that we do not have a complete definition for $\sqsubseteq$, because there is no general algorithm for checking the inclusion of sets of integers defined by recurrence relations. Instead, we simply check whether one set of inequations is a subset of another one, up to variable renaming (we will denote this syntactic inclusion as $\subseteq^s$). This check is enough to achieve correctness. Recall that in the size analysis we see the types as being fixed by a previous type analysis. We define $\sqsubseteq$ as follows:

$$\langle t, d, r\rangle \sqsubseteq \langle t, d', r'\rangle \iff t = t' \wedge d \subseteq^s d' \wedge r \subseteq^s r'$$

$\sqcup$ and $\bot$ are defined according to this definition of $\sqsubseteq$. For $\bot$ we need to know the types of the variables being referred to as extra parameters. Union is done syntactically again, taking care of renaming variables in inequations to refer to the same terms.

$$\langle t, d, r\rangle \sqcup \langle t, d', r'\rangle = \langle t, d \cup^s d', r \cup^s r'\rangle \qquad \bot_t = \langle t, \emptyset, \emptyset\rangle$$

4.2 λ_{call} to β_{entry}

This operation abstracts the unification of the subgoal variables (λ_{call}) onto head variables for each clause C_i (β_{entry_i}) defining a predicate. This is done in four steps. The first one is classification of variables, using mode information (also provided by type analysis): if a variable is unbound at the predicate call and bound to some non-variable term upon success, the variable is clasified as *output*. Otherwise, it is classified as *relevant*.[2]

The next step is generating the sized type schemas of subgoal variables by applying the function *sized* in Figure 2 on their corresponding regular types. Then, we set up constraints for the domain of the relevant variables in these sized types. For this purpose, we check if in the clause head the variable is bound to a ground term, in which case

[2] As future work, we plan to extend this classifier for the discovery of irrelevant variables which play no role in the size of outputs.

unify-sizes takes as input a list of unification equations of the form $X = Y$, where X is a subgoal variable and Y a term in the head, and produces a list of assignments. For each variable X, we denote τ_X and s_X its regular and sized type, respectively.

$$\textit{unify-sizes}(R) \;=\; \bigcup_{X=Y\in R}\left(\bigcup_{(V,p)\in paths(Y,[])}\{s_V \lesseqgtr \textit{unify-path}(p,\tau_X,s_X)\}\right)$$

$$\begin{array}{rcll} paths(Y,L) & = & \{(Y,L)\}, & Y \text{ variable} \\ paths(f(Y_1,\ldots,Y_n),L) & = & \bigcup paths(Y_i, L +\!\!+ [\langle f,i\rangle]), & f \text{ functor} \end{array}$$

$$\begin{array}{rcl} \textit{unify-path}([],\tau,s) & = & s \\ \textit{unify-path}\left([\langle f,i\rangle|r],\tau,\tau^{(\alpha,\beta)}(\bar{x})\right) & = & \textit{unify-path}(r,\sigma_i,s') \\ \text{where} & & \tau \to f(\sigma_1,\ldots,\sigma_n)\in\Phi \\ & & s' = \begin{cases} \tau^{(\alpha-1,\beta-1)}(\bar{x}), & \sigma_i = \tau \\ d, & \sigma_i \neq \tau, d_{\langle f,i\rangle}\in\bar{x} \end{cases} \\ \textit{unify-path}\left([\langle f,i\rangle|r],\tau,\tau(\bar{x})\right) & = & \textit{unify-path}(r,\sigma_i,s'), \quad s'_{\langle f,i\rangle}\in\bar{x} \end{array}$$

Fig. 3. Size unification of subgoal and head variables.

we constrain size variables to be bound to concrete numbers, according to the term. Otherwise, we just impose the constraint that size variables must be positive.

Finally, we need to perform the unification between sizes of subgoal input variables and sizes of head variables. This is performed using the algorithm in Figure 3. In the following steps, when a new variable is not found in the first component of the abstract substitution, a new sized type for it is generated, and it is added as *clausal*.

In our `listfact(L, FL)` example, from previous regular type analysis we know that at call time `L` is bound to a list of numbers and `FL` is a free variable, and on success `FL` is also bound to a list of numbers. Thus, we classify `FL` as *output* and `L` as *relevant*. Then, we generate the sized types for them. So far the procedure is the same for both clauses.

From now on, we will focus on the second clause. In β_{entry_2} we have unifications between relevant subgoal variables and head variables: $L = [E|R]$. Following the algorithm for $\textit{unify-sizes}([L = [E|R]])$, given in Figure 3, we need to call $paths([E|R],[])$. We get as output $[E = [\langle .,1\rangle], R = [\langle .,2\rangle]]$. In both calls to *unify-path* we will go to the rule $\textit{listnum} \to .(\textit{num},\textit{listnum})$. Since the type of the variable E is not *listnum*, we just take the sized type referred to by the *position* $\langle .,1\rangle$, in this case $n^{(\gamma_1,\delta_1)}$. For R, s' is similar to the initial sized type for L, but with one rule application less.

$$\beta_{entry\_2} = \left\langle \begin{array}{c} \left\{ \begin{array}{c} L \to (ln^{(\alpha_1,\beta_1)}(n^{(\gamma_1,\delta_1)}), relevant), FL \to (ln^{(\alpha_2,\beta_2)}(n^{(\gamma_2,\delta_2)}), output), \\ E \to (n^{(\gamma_3,\delta_3)}, clausal), R \to (ln^{(\alpha_4,\beta_4)}(n^{(\gamma_4,\delta_4)}), clausal) \end{array} \right\}, \\ \{\alpha_1 > 0, \beta_1 > 0\}, \left\{ \begin{array}{c} n^{(\gamma_3,\delta_3)} \lesseqgtr n^{(\gamma_1,\delta_1)} \\ ln^{(\alpha_4,\beta_4)}(n^{(\gamma_4,\delta_4)}) \lesseqgtr ln^{(\alpha_1-1,\beta_1-1)}(n^{(\gamma_1,\delta_1)}) \end{array} \right\} \end{array} \right\rangle$$

4.3 The Extend Operation

This operation is responsible for extending the current abstract element with the information of a call to a literal. The operation is very simple: include the sized types of any

$$unify\text{-}back(R) = \bigcup_{X=Y \in R} \{s_X \leqslant unify\text{-}back'(Y, \tau_Y)\}$$

$$\begin{array}{rcll} unify\text{-}back'(t,\tau) & = & ground\text{-}size(t,\tau), & t \text{ ground} \\ unify\text{-}back'(X,\tau) & = & s_X, & X \text{ variable} \\ unify\text{-}back'(f(t_1,\ldots,t_n),\tau) & = & none\text{-}but(\tau, f(d_1,\ldots,d_n)) & \\ \text{where} & & d_i = unify\text{-}back'(t_i,\sigma_i) \text{ and} & \\ & & \tau \to f(\sigma_1,\ldots,\sigma_n) \in \Phi & \end{array}$$

Fig. 4. Backwards size unification of subgoal and head variables.

variable which was not yet in the first component of the abstract element and add a call to the equation for the clause referencing the literal.

In our example, we will need to extend $\beta_{entry,2}$ (which will be the first λ in the second clause) with a call to $fact$. To do so, we add the sized type schema for F (we already have information for E) and the call, so the abstract substitution is now:[3]

$$\lambda_{2,2} = \left\langle \ \left\{\ldots, F \to (n^{(\gamma_5,\delta_5)}, clausal)\right\}, \{\ldots\}, \{\ \ldots, n^{(\gamma_5,\delta_5)} \leqslant fact(n^{(\gamma_3,\delta_3)})\ \} \ \right\rangle$$

4.4 β_{exit} to λ'

This operation abstracts the unification of the execution of an entire clause back with the subgoal variables. Thus the algorithm needs to do the opposite of λ_{call} to β_{entry}: deriving the size of a variable from the sizes of its constituent elements. To do so we use the *unify-back* algorithm outlined in Figure 4.[4] After this point we have a complete set of relations defining the output parameters.

For the second clause of `listfact` we have to call $unify\text{-}back'([F|FR], listnum)$. We need to recursively call *unify-back'* with the components F and FR. The final substitution for the second clause will be:

$$\lambda'_2 = \left\langle \ \left\{ \begin{array}{c} \left\{\ldots, FR \to (ln^{(\alpha_6,\beta_6)}(n^{(\gamma_6,\delta_6)}), clausal)\right\}, \{\ldots\}, \\ \cdots \\ ln^{(\alpha_6,\beta_6)}(n^{(\gamma_6,\delta_6)}) \leqslant listfact\left(ln^{(\alpha_4,\beta_4)}(n^{(\gamma_4,\delta_4)})\right) \\ ln^{(\alpha_2,\beta_2)}(n^{(\gamma_2,\delta_2)}) \leqslant ln^{(\alpha_6+1,\beta_6+1)}(n^{(\min(\gamma_5,\gamma_6),\max(\delta_5,\delta_6)})} \end{array} \right\} \ \right\rangle$$

4.5 Closed Forms

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 if presented in this closed form. Second, functions are amenable to comparison as outlined in (López-García et al. 2010), essential for example in verification.

[3] Only additions to the elements will be shown. Three dots (...) will replace previous information.
[4] The function *none-but* returns a sized type restricted to a particular type rule.

The $\uparrow$ operator will try to replace relations with a closed form bound. We can see this operator as overapproximating an abstract element, $x \sqsubseteq \uparrow x$. In our experiments we have integrated Mathematica as recurrence solver, and $\uparrow$ is applied at every $\sqcup$ step.

In our example we obtain the following abstract substitution for the first clause:

$$\lambda_1' = \left\langle \begin{array}{c} \left\{ L \rightarrow (ln^{(\alpha_1,\beta_1)}(n^{(\gamma_1,\delta_1)}), relevant), FL \rightarrow (ln^{(\alpha_2,\beta_2)}(n^{(\gamma_2,\delta_2)}), output) \right\}, \\ \{\alpha_1 = 1, \beta_1 = 1\}, \{ln^{(\alpha_2,\beta_2)}(n^{(\gamma_2,\delta_2)}) \lesseqgtr ln^{(1,1)}(n^{nob})\} \end{array} \right\rangle$$

Then, we can bound the joint inequations to get a closed solution:

$$\uparrow (\lambda_1' \sqcup \lambda_2') = \left\langle \ \{\dots\}, \{\alpha_1, \beta_1 > 0\}, \{ln^{(\alpha_2,\beta_2)}(n^{(\gamma_2,\delta_2)}) \lesseqgtr ln^{(\alpha_1,\beta_1)}(n^{(\gamma_1!,\delta_1!)})\} \ \right\rangle$$

5 Refinements in the Analysis

Multivariance and Widenings In the example analysed above there is an implicit assumption while setting the relations: the recursive call in the body of `listfact` refers to the same predicate call, so we set up a recurrence equation. 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 (such as different modes or even types).

The CiaoPP framework allows multivariance (support for different call patterns of the same predicate) in the analysis. But to do so we cannot just add calls with the bare name of the predicate, because it will conflate all the existing versions. The proposed solution adds a new component to the abstract element: a random name given to the specific instance of the predicate we are analyzing, that is generated in the λ_{call} to β_{entry}. In the computation of the fixpoint, the $\sqcup$ operator is changed to a widening ∇ which conflates all different versions of the same predicate. In this way we obtain the recurrences.

Structural Subtyping Another problem that may arise is that a predicate returns a subtype of the type we were looking for. For example, in:

```
n_to_zero(0, [0]).
n_to_zero(N, [N|R]) :- N1 is N - 1, n_to_zero(N1, R).
```

the regular type inferred is *nonemptylistnum* for the second argument. In this case, in the backwards unification we have variables of type *num* and *nonemptylistnum* but the rule for the `./2` functor reads `.(num, listnum)`. However, since *nonemptylistnum* $\sqsubseteq$ *listnum* we can view the size description as an instance of a description of its supertype.

To do so, we have developed an extended version of the Dart-Zobel algorithm for type inclusion (Dart and Zobel 1992) which can be found in Appendix B.

6 Cardinality and Resource Analysis

In order to assess the usefulness of the new size analysis in the resource usage estimation application (which is our main goal), we have also developed an integrated into the CiaoPP abstract interpretation framework a resource usage analysis and a cardinality analysis. The latter infers lower and upper bounds on the numbers of solutions produced by a predicate. We provide below a sketch of these analyses (the full details are beyond the scope of this paper).

Cardinality has a multiplicative behavior: if we know the number of solutions of every literal in a clause, we can bound the number of solutions contributed by it using $S_{clause}\,(p(\bar{x}) \ :- \ q_1(\bar{x}_1), \ldots, q_n(\bar{x}_n)) \leq \prod_{i=1}^{n} S_{pred}(q_i(\bar{x}_i))$. Here we are implicitly using the previously discussed size analysis. The number of solutions of the whole predicate S_{pred} can be calculated by gathering all the equations and solving the resulting system.

Regarding resources, following (Navas et al. 2007) each *resource* is defined by its *head cost* β, which quantifies the amount of resource used in the unification between a subgoal and a clause head, and its *literal cost* δ, which quantifies the amount of resource needed for preparing a call to a predicate. Apart from that, the user can attach directly some resource usage functions to particular predicates. Using these parameters, we can get a formula for upper bounding the resource usage of a clause $C \equiv p(\bar{x}) : - q_1(\bar{x}_1), \ldots, q_n(\bar{x}_n)$:

$$RU_{clause}\,(C) \leq \beta(p(\bar{x})) + \sum_{i=1}^{n} \left(\prod_{j=1}^{i-1} S_{pred}(q_j(\bar{x}_j)) \right) (\delta(q_i(\bar{x}_i)) + RU_{pred}(q_i(\bar{x}_i)))$$

The resource usage of a predicate RU_{pred} is calculated in a similar way to S_{pred}.

As we have seen, cardinality and resource analyses are tightly related to size analysis. Consequently, our implementation of these analyses is via an extension of the previously defined sized types abstract domain:

- The upper and lower bounds, both for the number of solutions and for each resource, is represented by a pair of bound variables (S_l, S_u) and (RU_l, RU_u) respectively, similarly to those used by the analysis in sized type schemas.
- These variables are initialized in the λ_{call} to β_{entry} step: S_l and S_u to 1 (the cardinality before any literal is called), and RU_l and RU_u to the corresponding resource head cost β.
- In each *extend* step, we need to update the bound variables with new values, given the cardinality (S'_l, S'_u) and resource usage (RU'_l, RU'_u) of the called literal:
  - For upper bounds, the cardinality is updated by the product of the previous cardinality and the one from the called literal, $S_u \times S'_u$. For resource usage, the formula is very similar, $RU_u + S_u \times (\delta + RU'_u)$.
  - The methodology is similar for lower bounds, but we have to take into account the possibility of failure, as explained in (Debray et al. 1997).
- As a result of threading the variables through all *extend* steps, in β_{exit} the values of the bound variables for cardinality and resources will be equal to the ones obtained by the formulas we have previously presented.

7 Experimental Results and Conclusions

We have constructed a prototype implementation in Ciao by defining the abstract operations for sized type analysis that we have described in this paper and plugging them into CiaoPP's PLAI implementation. While full benchmarking is beyond the scope of the paper, we provide preliminary results on two aspects: (a) comparison of the new size analysis to the existing CiaoPP size analyses (Debray and Lin 1993; Debray et al. 1997; Navas et al. 2007), and (b) effect of using the new size analysis in the resource usage analysis application.

Regarding (a), the main advantage of our technique is the richer information about the size of terms that is inferred by the analysis. As an illustrative example, consider the predicate `insert` used in insertion sort of a list of lists. The code we used for analysis is a direct translation to Prolog of the one in (Hoffmann et al. 2012):

```
insert(X,[],[X]).
insert(X,[Y|Ys],[X,Y|Ys]) :- leq(X,Y), !.
insert(X,[Y|Ys],[Y|Zs]) :- insert(X,Ys,Zs).

leq([],_).
leq([X|Xs],[Y|Ys]) :- X =< Y, leq(Xs,Ys).
```

Given input arguments[5] $\langle X \rightarrow ln^{(c,d)}(n^{(e,f)}), L \rightarrow lln^{(g,h)}(ln^{(i,j)}(n^{(k,l)}))\rangle$, we get as sized type relation for the third argument `I`[6]:

$$I \quad \rightarrow \quad nelln \begin{pmatrix} ln_{\langle .,1\rangle}^{(\min(i,c),\max(d,j))}(n^{(\min(k,e),\max(f,l))}) \\ lln_{\langle .,2\rangle}^{(g,h)}(ln^{(\min(i,c),\max(d,j))}(n^{(\min(k,e),\max(f,l))})) \end{pmatrix}$$

We see that the analysis has correctly inferred that the result will be a non-empty list and the bounds for all inner elements. For example, the first element of the list of lists will be either the list `X` or one list in `L`, so the bound at that position will be the largest.

Our results show that the new analysis improves on the previous one in 86% (13/15) of a set of benchmarks and produces the same results in the other 14%.

Regarding (b), we have compared the new CiaoPP lower and upper bound resource analyses using the new size analysis with the previous CiaoPP analyses (Debray and Lin 1993; Debray et al. 1997; Navas et al. 2007), and also (upper bounds) with *RAML*'s analysis (Hoffmann et al. 2012). The new analyses improve on CiaoPP's previous resource analysis and in most cases, and are equal in the rest. RAML only infers polynomial costs, while our new approach can infer exponential costs and many other types of cost functions. For predicates with polynomial cost, we get equal or better results than RAML.

8 Other Related Work

Apart from recurrence equations, there are other approaches to size analysis. One popular one is the use of CLP($\mathbb{R}$) and convex hulls, such as (Benoy and King 1997). In this case, the analysis infers a set of linear inequations between sizes of terms. The main advantage of this proposal is the possibility of relating sizes of several arguments. However, these approaches are usually limited in the mathematical domain used for abstraction (for example, linear inequations), whereas recurrence relations allow much richer expressions.

As mentioned in the introduction, (Hoffmann et al. 2012) shows another approach to size analysis, based on the potential method. Although it allows some costs that we cannot express in our system (for example, sums over all the elements in a list), it is limited to polynomial expressions. In our case, not being tied to polynomial bounds is important, since problems such as the number of solutions usually have exponential behavior.

Inference of norms for termination analysis is also related to size analysis. For example, (Decorte et al. 1994) or (Bruynooghe et al. 2007) use semi-linear norms to prove termination. These norms define the size of a term as the sum of some of its components,

[5] We write *lln* for *listlistnum*, the type of lists of lists of numbers, and *nelln* for its non-empty variant.
[6] We are using a condensed version of the abstract element, where we write the results of inequations directly inside the sized type.

which are later related by linear inequations. This approach summarizes all information in one number, so it is less convenient for the developer and less useful for other analyses.

References

Benoy, F. and King, A. 1997. Inferring Argument Size Relationships with CLP(R). In *Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'97)*. Lecture Notes in Computer Science, vol. 1207. Springer, 204–223.

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

Bruynooghe, M., Codish, M., J. P. Gallagher, Genaim, S., and Vanhoof, W. 2007. Termination analysis of logic programs through combination of type-based norms. *ACM Transactions on Programming Languages and Systems 29,* 2.

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. 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 *1997 International Logic Programming Symposium*. MIT Press, Cambridge, MA, 291–305.

Decorte, S., Schreye, D. D., and Fabris, M. 1994. Exploiting the power of typed norms in automatic inference of interargument relations. Tech. rep., TR 246, Dpt CS, , K.U.Leuven.

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.

Hughes, J., Pareto, L., and Sabry, A. 1996. Proving the correctness of reactive systems using sized types. In *POPL*. 410–423.

López-García, P., Darmawan, L., and Bueno, F. 2010. A Framework for Verification and Debugging of Resource Usage Properties. In *Technical Communications of ICLP*. LIPIcs, vol. 7. Schloss Dagstuhl, 104–113.

López-García, P., Hermenegildo, M., and Debray, S. K. 1996. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *J. of Symbolic Computation, Special Issue on Parallel Symbolic Computation 21*, 715–734.

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.

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.

$$
\begin{array}{rcl}
\mathit{merge}\left(\tau^{(\alpha,\beta)}(\bar{x}), \tau^{(\gamma,\delta)}(\bar{y})\right) & = & \tau^{(\alpha+\gamma,\beta+\delta)}(\mathit{merge\text{-}args}(\bar{x},\bar{y})) \\
\mathit{merge\text{-}args}(\bar{x},\bar{y}) & = & \{\mathit{merge\text{-}arg}(x_i,y_i)_{\langle f,p\rangle} : x_i \text{ and } y_i \text{ have subscript } \langle f,p\rangle\} \\
\mathit{merge\text{-}arg}(\sigma^{nob}, y) & = & y \\
\mathit{merge\text{-}arg}(x, \sigma^{nob}) & = & x \\
\mathit{merge\text{-}arg}\left(\sigma^{(\alpha,\beta)}(\bar{z}), \sigma^{(\gamma,\delta)}(\bar{w})\right) & = & \sigma^{(\max(\alpha,\gamma),\min(\beta,\delta))}(\mathit{merge\text{-}args}(\bar{z},\bar{w}))
\end{array}
$$

$$
\mathit{none}(\tau) \quad = \quad \tau^{nob}\left(\bigcup_{\tau\to\phi\in\Phi} \mathit{none\text{-}rule}(\phi,\tau)\right)
$$

$$
\begin{array}{rcll}
\mathit{none\text{-}rule}(f(\sigma_1,\ldots,\sigma_n),\tau) & = & \{\sigma^{nob}_{i,\langle f,i\rangle} : \sigma_i \not\sqsubseteq \tau\} & \\
\mathit{none\text{-}rule}(\sigma,\tau) & = & \{\sigma^{nob}_{\epsilon}\}, & \sigma \not\sqsubseteq \tau \\
\mathit{none\text{-}rule}(\sigma,\tau) & = & \emptyset, & \sigma \sqsubseteq \tau
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{none\text{-}but}(\tau, f(d_1,\ldots,d_n)) & = & \mathit{merge}(\tau^{(1,1)}(A), \text{fold } \mathit{merge} \text{ over } S) \\
\text{where} & & \langle A, S\rangle = \bigcup_{\tau\to\phi\in\Phi} \mathit{none\text{-}but\text{-}rule}(\tau,\phi,f(d_1,\ldots,d_n))
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{none\text{-}but\text{-}rule}(\tau, f(\sigma_1,\ldots,\sigma_n), f(d_1,\ldots,d_n)) & = & \langle\{d_{i,\langle f,i\rangle} : \sigma_i \not\sqsubseteq \tau\}, \{d_{i,\langle f,i\rangle} : \sigma_i \not\sqsubseteq \tau\}\rangle \\
\mathit{none\text{-}but\text{-}rule}(\tau,\phi,f(d_1,\ldots,d_n)) & = & \langle\emptyset,\emptyset\rangle, \quad \phi \text{ does not start with } f
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{ground\text{-}size}(n, \mathit{num}) & = & \mathit{num}^{(n,n)} \\
\mathit{ground\text{-}size}(f(t_1,\ldots,t_n),\tau) & = & \mathit{none\text{-}but}(\tau, f(d_1,\ldots,d_n)) \\
\text{where} & & d_i = \mathit{ground\text{-}size}(t_i,\sigma_i) \\
& & \tau \to f(\sigma_1,\ldots,\sigma_n) \in \Phi
\end{array}
$$

Fig. A 1. Sized types auxiliary functions.

Appendix A Auxiliary Algorithms over Sized Types

In Figure A 1 we describe the auxiliary algorithms used in the operations in the sized types abstract domain. These algorithms are very similar to the derivation of sized type definitions. For simplicity, we only give the algorithms for recursive types, the non-recursive case just does not compute *bounds* for the number of rule applications.

Appendix B Extended Type Inclusion with Sizes

We do not include the definition of the "plural" functions, which just apply a "singular" function over a list (for example *opens* just collects the results of *open* over every element of a list). The auxiliary functions can be found in Figure B 1 and the main *subset* algorithm is in Figure B 2.

We assume *head* and *tail* functions giving the first and rest elements of a list, respectively, and a $+\!\!+$ list concatenation operator.

$$expand(\psi) = \begin{cases} \{\psi\}, & \tau \text{ not a type symbol} \\ \{[\langle \phi, expand\text{-}size(\phi, s)\rangle] +\!\!+ tail(\psi) : \tau \to \phi \in \Phi\}, & \tau \text{ a type symbol} \end{cases}$$
$$\text{where } \langle \tau, s\rangle = head(\psi)$$

$$\begin{aligned} expand\text{-}size(\sigma, \tau^{(\alpha,\beta)}(\bar{x})) &= \sigma^{s'}(\bar{y}), && \text{if } \sigma^{s'}_{\epsilon}(\bar{y}) \in \bar{x} \\ expand\text{-}size(f(\sigma_1, \ldots, \sigma_n), \tau^{(\alpha,\beta)}(\bar{x})) &= f(d_1, \ldots, d_n) \\ \text{where} \quad d_i &= \begin{cases} \tau^{(\alpha-1,\beta-1)}(\bar{x}), & \sigma_i = \tau \\ s_i, & \sigma_i \neq \tau, s_{i,\langle f,i\rangle} \in \bar{x} \end{cases} \end{aligned}$$

$$selects(\tau, \Psi) = \begin{cases} \{\psi \in \Psi : head(\psi) = \langle \top, s\rangle \vee head(\psi) = \langle \tau, s\rangle\}, & \tau \text{ a type symbol} \\ \{\psi \in \Psi : head(\psi) = \langle \top, s\rangle \vee head(\psi) = \langle f(\omega_1, \ldots, \omega_n), s\rangle\}, & \tau = f(\sigma_1, \ldots, \sigma_n), n > 0 \end{cases}$$

$$open(\langle \tau, s\rangle, \psi) = \begin{cases} tail(\psi), & \tau \text{ is } \top \text{ or a base symbol} \\ [\langle \top, \top\rangle, \ldots, \langle \top, \top\rangle] +\!\!+ tail(\psi), & \tau = f(\omega_1, \ldots, \omega_n), head(\psi) = \langle \top, s'\rangle \\ [\langle \sigma_1, s_1\rangle, \ldots, \langle \sigma_n, s_n\rangle] +\!\!+ tail(\psi), & \tau = f(\omega_1, \ldots, \omega_n), \\ & head(\psi) = \langle f(\sigma_1, \ldots, \sigma_n), f(s_1, \ldots, s_n)\rangle \end{cases}$$

Fig. B 1. Extended type inclusion, auxiliary functions.

$$\begin{aligned} subset(\langle \bot, s\rangle, \langle \tau, s'\rangle) &= \langle \text{true}, \emptyset\rangle \\ subset(\langle \sigma, s\rangle, \langle \bot, s'\rangle) &= \langle \text{false}, \emptyset\rangle \\ subset(\langle \sigma, s\rangle, \langle \tau, s'\rangle) &= \langle b, postprocess(r)\rangle \\ \text{where} \quad & \langle b, r\rangle = subsetv([\langle \sigma, s\rangle], \{[\langle \tau, s'\rangle]\}, \emptyset) \end{aligned}$$

$$\begin{aligned} subsetv(\psi, \emptyset, C) &= \langle \text{false}, \emptyset\rangle \\ subsetv([], \Psi, C) &= \langle \text{true}, \emptyset\rangle \\ subsetv(\psi, \Psi, C) &= subsetv(tail(\psi), tails(\Psi), C) \\ &\text{if } \langle head(\psi), \Theta\rangle \in C \wedge heads(\Psi) \subseteq \Theta \\ subsetv(\psi, \Psi, C) &= subsetvs(expand(\psi), \Psi, C \cup \{\langle head(\psi), heads(\Psi)\rangle\}) \\ &\text{if } head(\psi) = \langle \tau, s\rangle, \tau \text{ a type symbol} \\ subsetv(\psi, \Psi, C) &= \langle b_R, R \cup r_R\rangle \\ &\text{if } head(\psi) = \langle \tau, s\rangle, \tau \text{ is } \top \text{ or } f(\omega_1, \ldots, \omega_n), n > 0 \\ &\quad \Sigma = selects(\tau, expands(\Psi)) \\ &\quad R = \bigcup_{S \in \Sigma} unify(s, head(S)) \\ &\quad \langle b_R, r_R\rangle = subsetv(open(\langle \tau, s\rangle, \psi), opens(\langle \tau, s\rangle, \Sigma), C) \end{aligned}$$

$$\begin{aligned} subsetvs([], \Psi, C) &= \langle \text{true}, \emptyset\rangle \\ subsetvs([\psi|R], \Psi, C) &= \langle b_\psi \wedge b_R, r_\psi \cup r_R\rangle \\ \text{where} \quad & \langle b_\psi, r_\psi\rangle = subsetv(\psi, \Psi, C) \\ & \langle b_R, r_R\rangle = subsetvs(R, \Psi, C) \end{aligned}$$

postprocess(R) gathers all the unifications in R over the same variable X, and generates the maximum or minimum expression of it, depending on whether the variable is in an upper or lower bound position.

Fig. B 2. Extended type inclusion, main *subset* function.