Online appendix for the paper

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

published in Theory and Practice of Logic Programming

A. SERRANO[1]† P. LOPEZ-GARCIA[2,3] 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

Appendix A The Abstract Interpretation Framework

Abstract interpretation (Cousot and Cousot 1992) is a framework for static analysis. Execution of the program on a concrete domain is simulated in an abstract domain, simpler than the former one. Both domains must be lattices, $\langle \mathcal{P}(\Sigma), \subseteq \rangle$ and $\langle \Delta, \sqsubseteq \rangle$. To go from one to another we use a pair of functions, called *abstraction* $\alpha : \mathcal{P}(\Sigma) \to \Delta$ and *concretization* $\gamma : \Delta \to \mathcal{P}(\Sigma)$, which should form a Galois connection:

$$\langle \mathcal{P}(\Sigma), \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \Delta, \sqsubseteq \rangle \text{ if and only if } \alpha(x) \sqsubseteq y \iff x \subseteq \gamma(y)$$

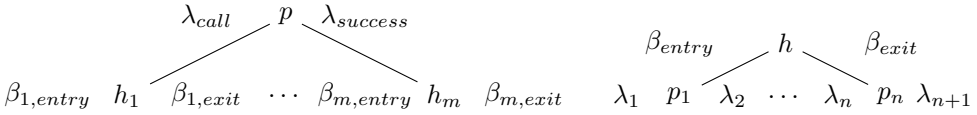
Intuitively $\alpha(\sigma)$ generates the smallest element in Δ that contains all the elements in σ, and $\gamma(\delta)$ computes all the concrete elements represented by δ.

The methodology is very general, so we focus specifically on the PLAI (Muthukumar and Hermenegildo 1989; Muthukumar and Hermenegildo 1992) framework. 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. The procedure is *generic* (*parametric*) in the sense that it factors out the abstraction of program execution flow (the execution and-or trees), which is common to many different analyses, from other (mainly data-related) abstractions, which are more application-specific, and which are encoded as one or more *abstract domains*. It is also 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*)

∗ 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.

and produces an 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, for all procedure versions. This algorithm 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 resource analysis. In PLAI, abstract domains are pluggable units which need to define implementations of $\sqsubseteq$, least upper bound ($\sqcup$), bottom ($\bot$), and a number of other operations related to predicate calls and successes.

For any clause $h :- q_1, \ldots, q_n.$, let λ_i and λ_{i+1} be the abstract substitutions to the left and to the right of literal q_i, and λ_{call_i} and $\lambda_{success\_i}$ their projections onto the variables of q_i respectively. λ_1 and λ_{n+1} are the *entry* and *exit* substitutions of the clause respectively, denoted also as β_{entry} and β_{exit}. We can show this graphically as follows:



To compute $\lambda_{success}$ from λ_{call} of a generic (sub)goal $p(\bar{x})$ with predicate p:

1. Generate a β_{entry_i} from λ_{call} for each of the m clauses C_i defining the predicate p. This transfers the unification of the subgoal and head variables into Δ.
2. For each clause C_i, compute β_{exit_i} from β_{entry_i}, and then project β_{exit_i} back again onto the subgoal variables, obtaining λ'_i.
3. Aggregate all the exit substitutions using the least upper bound, $\lambda_{success} = \bigsqcup_{i=1}^{m} \lambda'_i$.

Computing β_{exit} from β_{entry} is straightforward: set β_{entry} as λ_1. Then, project it onto the variables appearing in the call to the first literal q_1, obtaining λ_{call_1} for q_1, and compute $\lambda_{success\_1}$ from it using the procedure mentioned above. Now λ_1 is integrated with this success substitution, referred to as *extending* λ_1 with $\lambda_{success\_1}$. The result is set as λ_2, for which the same series of steps is performed with respect to the second literal q_2. The process continues until λ_{n+1} is obtained, which is actually β_{exit}.

In the process, more than one call substitution may appear for the same predicate. This is called *multivariance* of predicates. Furthermore, if the predicate is recursive, a fixpoint needs to be computed. To do so, the process above is iterated starting from the bottom element of the lattice, $\bot$. (Muthukumar and Hermenegildo 1992; Puebla and Hermenegildo 1996) describe performant algorithms for this purpose, which are implemented in CiaoPP.

Appendix B The Abstract Elements, Redux

Because of space constraints, in the main part of the paper the concrete and abstract domains have not been described in full. In this section we aim to give a more precise definition of both elements within the framework of abstract interpretation.

In the concrete domain, the *resource usage* of a predicate `p` with respect to a set of resources r_i is given by a set of triples $(\bar{t}, s, r_{p,i})$, where $\bar{t}$ is a tuple of terms. The interpretation of such set is that for a call to `p` with arguments bound to $\bar{t}$, the number

of solutions is exactly s and the resource usage of each r_i is exactly $r_{p,i}$. Note that s and $r_{p,i}$ are actual values, not equations or recurrences. The resource usage is computed by adding the head cost at the point of entering a clause and the literal cost at the point of calling a literal in the body, using the usual SLD resolution semantics. This definition follows closely the one in (López-García et al. 2010), but extended to support several resources and cardinality.

Let $dom(e)$ be the set of tuples of terms $\bar{t}$ for which a concrete element e has information over its resource usage. We define $e \sqsubseteq_c e'$ if and only if $dom(e) \subseteq dom(e')$ and for each $\bar{t} \in dom(e)$, $(p(\bar{t}), s, r_{U,i}) = (p(\bar{t}), s', r'_{U,i})$. That is, the set of terms of the smaller element must be a subset of the larger one, and the cardinality and resource usage must coincide in the common part of their domains.

This concrete domain is abstracted in three different ways, to get a compound domain. Two of them have already been discussed in the literature: the non-failure and determinacy analyses. Those components of the abstract domain correspond to abstracting the set of elements $\bar{t}$ using a regular type abstract domain and then summarizing for those elements whether $s = 0$ or $s > 0$ (for the non-failure domain) and whether $s = 1$ or $s \neq 1$ (for the determinacy one). The $failed?$ component of the abstract elements follows closely the non-failure analysis, keeping different information during the analysis, but with the same result.

For the recurrences part, we perform several abstractions. First of all, we move from strict values for the number of solutions and resource usage to value bounds. Thus, the elements are sets of triples $(\bar{t}, (s_L, s_U), (r_{L,i}, r_{U,i}))$. The ordering is now given by:

$$\begin{array}{rcl} e \sqsubseteq_1 e' & \iff & dom(e) \subseteq dom(e') \\ & \text{and} & \text{for each } \bar{t} \in dom(e), (s_L, s_U) \subseteq (s'_L, s'_U) \text{ and } (r_{L,i}, r_{U,i}) \subseteq (r'_{L,i}, r'_{U,i}) \end{array}$$

The abstraction function in this case is very simple, we just need to send each value to an interval with it as only point:

$$\alpha_1(\{(\bar{t}, s, r_{p,i})\}_{\bar{t}}) = \{(\bar{t}, (s, s), (r_{p,i}, r_{p,i}))\}_{\bar{t}}$$

The second abstraction involves summarizing the domain of each $\alpha_1(e)$ using the sized types abstract domain. As discussed in (Serrano et al. 2013), a set of terms is described via sized types using sized type schemas along with a domain d which tells which are the values of the bound variables which are covered by the abstract element, and a set of recurrences r which defines the relations that bound variables must satisfy between them. When adding resource usage information, apart from the bounds from sized types we can refer to new variables: s_L and s_U refer to the upper and lower bound in the number of solutions, and $v_{resources}$ contains such variables for each resource in the system.

In this case, it is easier to give the concretization function to move from an abstract element e to one in the intermediate abstract domain:

$$\gamma_2(\langle d, (s_L, s_U), v_{res}, r)\rangle) = \bigcup_{\bar{t} \in \gamma_{\text{sized types}}(\langle d, r\rangle)} (\bar{t}, bound_{(s_L, s_U)}(\bar{t}, r), bound_{v_{res}}(\bar{t}, r))$$

where $bound_v(\bar{t}, r)$ returns the upper and lower *numerical* bounds for the variables v as given in the recurrences r for the tuple of values $\bar{t}$. In few words, γ_2 takes all the possible tuples of values given by the sized type we refer to, and computes the cardinality and resource usage of each of them as given by the recurrence equations.

The intermediate domain and this concretization function allows us to define an ordering $\sqsubseteq$ in the abstract elements. But, as stated in the main part of the paper, doing so would entail knowing whether some recurrences define a set that is larger or smaller than another one. This is an undecidable problem, and thus we need to resort to other checks which, while being correct, are not complete. In our case, we chose to use a syntactic check.

From α_1 we can obtain the corresponding concretization function γ_1, and from γ_2 we can do the same to obtain an α_2. By composition we obtain the abstraction $\alpha_r = \alpha_2 \cdot \alpha_1$ and concretization $\gamma_r = \gamma_1 \cdot \gamma_2$ functions that define the Galois connection between concrete resource usage triples and the abstract domain of recurrence equations.

As stated before, our complete abstract elements:

$$\langle (s_L, s_U), v_{resources}, failed?, d, r, nf, det \rangle$$

are the combination of that given by $\langle \alpha_r, \gamma_r \rangle$ with those of non-failure (which give the $failed?$ and nf components) and determinism (which gives the det component), which abstract information about s over all possible values. For an abstract element a to be smaller than b, it must be smaller in all of the three domains at the same time.

References

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

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

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.

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.

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.

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. Cambridge U. Press, 1–14.