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† P. LOPEZ-GARCIA2,3 M. V. HERMENEGILDO3,4

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 \( \langle \Delta, \subseteq \rangle \). To go from one to another we use a pair of functions, called abstraction \( \alpha : \mathcal{P}(\Sigma) \rightarrow \Delta \) and concretization \( \gamma : \Delta \rightarrow \mathcal{P}(\Sigma) \), which should form a Galois connection:

\[
\langle \mathcal{P}(\Sigma), \subseteq \rangle \xrightarrow{\alpha} \langle \Delta, \subseteq \rangle \text{ if and only if } \alpha(x) \subseteq y \iff x \subseteq \gamma(y)
\]

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

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 \((L, \lambda_c)\) 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 $\lambda_o$ 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 $\lambda_1$ and $\lambda_{i+1}$ be the abstract substitutions to the left and to the right of literal $q_i$, and $\lambda_{\text{call},i}$ and $\lambda_{\text{success},i}$ their projections onto the variables of $q_i$ respectively. $\lambda_1$ and $\lambda_{n+1}$ are the entry and exit substitutions of the clause respectively, denoted also as $\beta_{\text{entry}}$ and $\beta_{\text{exit}}$. We can show this graphically as follows:

$$
\begin{array}{c}
\lambda_{\text{call}} & p & \lambda_{\text{success}} \\
\beta_{\text{entry}} & h_1 & \beta_{\text{exit}} \\
\beta_{\text{entry}} & \cdots & \beta_{m,\text{entry}} & h_m & \beta_{m,\text{exit}}
\end{array}
$$

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

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

Computing $\beta_{\text{exit}}$ from $\beta_{\text{entry}}$ is straightforward: set $\beta_{\text{entry}}$ as $\lambda_1$. Then, project it onto the variables appearing in the call to the first literal $q_1$, obtaining $\lambda_{\text{call},1}$ for $q_1$, and compute $\lambda_{\text{success},1}$ from it using the procedure mentioned above. Now $\lambda_1$ is integrated with this success substitution, referred to as extending $\lambda_1$ with $\lambda_{\text{success},1}$. The result is set as $\lambda_2$, for which the same series of steps is performed with respect to the second literal $q_2$. The process continues until $\lambda_{n+1}$ is obtained, which is actually $\beta_{\text{exit}}$.

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 $(t, s, r_{p,i})$, where $t$ 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 \( \text{dom}(e) \) be the set of tuples of terms \( \vec{t} \) for which a concrete element \( e \) has information over its resource usage. We define \( e \sqsubseteq_c e' \) if and only if \( \text{dom}(e) \subseteq \text{dom}(e') \) and for each \( \vec{t} \in \text{dom}(e), (p(\vec{t}), s, r_{U,i}) = (p(\vec{t}), s', r'_{U,i}) \). 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 \( \vec{t} \) 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 \( \text{failed?} \) 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 \( (\vec{t}, (s_L, s_U), (r_{L,i}, r_{U,i})) \). The ordering is now given by:

\[
e \sqsubseteq_1 e' \iff \text{dom}(e) \subseteq \text{dom}(e') \\
\text{and for each } \vec{t} \in \text{dom}(e), (s_L, s_U) \subseteq (s'_L, s'_U) \text{ and } (r_{L,i}, r_{U,i}) \subseteq (r'_{L,i}, r'_{U,i})
\]

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

\[
\alpha_1((\vec{t}, s, r_{p,i}))_{\vec{t}} = ((\vec{t}, (s, s), (r_{p,i}, r_{p,i})))_{\vec{t}}
\]

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 \( v_{\text{resources}} \) 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:

\[
\gamma_2((d, (s_L, s_U), v_{\text{res}})) = \bigcup_{\vec{t} \in \gamma_1(\text{dom}(d, (s_L, s_U))))} ((\vec{t}, \text{bound}_{(s_L, s_U)}(\vec{t}, r)), \text{bound}_{v_{\text{res}}}(\vec{t}, r))
\]

where \( \text{bound}_{n}(\vec{t}, r) \) returns the upper and lower \textit{numerical} bounds for the variables \( v \) as given in the recurrences \( r \) for the tuple of values \( \vec{t} \). In few words, \( \gamma_2 \) 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 $\alpha_1$ we can obtain the corresponding concretization function $\gamma_1$, and from $\gamma_2$ we can do the same to obtain an $\alpha_2$. 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: 

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

are the combination of that given by $\langle \alpha_r, \gamma_r \rangle$ with those of non-failure (which give the $\text{failed}?$ and $\text{nf}$ components) and determinism (which gives the $\text{det}$ 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


