Towards a Rule-based Approach for Deriving Abstract Domains (Extended Abstract) * †

[1,2]Daniel Jurjo, [1,2]Jose F. Morales, [3,2]Pedro López-García, and [1,2]Manuel V. Hermenegildo

[1]Universidad Politécnica de Madrid (UPM) Madrid, Spain

[2]IMDEA Software Institute, Madrid, Spain

[3]Spanish Council for Scientific Research

{daniel.jurjo,josef.morales,pedro.lopez,manuel.hermenegildo}@imdea.org

Abstract interpretation [3] allows constructing sound program analysis tools which can extract properties of a program by safely approximating its semantics. Static analysis tools are a crucial component of the development environments for many programming languages. Abstract interpretation proved practical and effective in the context of (Constraint) Logic Programming ((C)LP) [15, 12, 14, 13, 1, 10, 9] which was one of its first application areas (see [6]), and the techniques developed in this context have also been applied to the analysis and verification of other programming paradigms by using semantic translation to Horn Clauses (see the recent survey [4]). Unfortunately, the implementation of (sound, precise, efficient) abstract domains usually requires coding from scratch a large number of domain-related operations. Moreover, due to undecidability, a loss of precision is inevitable, which makes the design (and implementation) of more domains, as well as their combinations, eventually necessary to successfully prove arbitrary program properties. In this paper we focus on the latter problem by proposing a rule-based methodology for the design and rapid prototyping of new domains for logic programs, as well as composing and combining existing ones. Our techniques are inspired by those used in logic-based languages for implementing constraint domains at different abstraction levels.

Proposal. The construction of analyses based on abstract interpretation requires the defintion of some basic domain operations ($\sqsubseteq, \sqcap, \sqcup$ and, optionally, the widening ∇ operator); the abstract semantics of the primitive constraints (representing the *built-ins*, or basic operations of the source language) via *transfer functions* (f^{α}); and possibly some other additional instrumental operations over abstract substitutions. In addition, the classical top-down analysis approach requires a number of additional definitions of derived operations used by the analysis framework to implement procedure call, return, recursion, etc. Detailed descriptions of all these operations can be found in [12, 11, 2, 7, 5]. We propose a rule language inspired in rewriting to derive, script, and combine abstract domains. The objective is to reduce the time and effort required to write new abstract domains, both from scratch and as combinations of other domains

The proposed rule-based language. Given $s+1$ sets of constraints, $\mathscr{L}, \mathscr{C}_1, \ldots, \mathscr{C}_s$, we define $AND(\mathscr{L}, \mathscr{C}_1, \ldots, \mathscr{C}_s)$ as the set of rules of the form $l_1, \ldots, l_n \mid g_1, \ldots, g_l \Rightarrow r_1, \ldots, r_m \,\#\, label$, where s, n, m, and l are arbitrary positive integers, and the rule meets the following condition:

$$\forall i, j, k \text{ s.t. } i \in \{1, \ldots, n\}, j \in \{1, \ldots, m\} \text{ and } k \in \{1, \ldots, l\} : (l_i, r_j \in \mathscr{L} \text{ and } \exists u \in \{1, \ldots, s\} \text{ s.t. } g_k \in \mathscr{C}_u)$$

The elements $l_1, \ldots, l_n$ constitute the *left side* of the rule; $r_1, \ldots, r_m$ the *right side*; and $g_1, \ldots, g_l$ the *guards*. Given $t+s$ sets of constraints $\mathscr{L}_1, \ldots, \mathscr{L}_t, \mathscr{C}_1, \ldots, \mathscr{C}_s$ such that $\forall v \in \{1, \ldots, t\} : \mathscr{L}_v \subseteq \mathscr{L}$, we

*An extended version of this work can be found in [8].

†This work has been partially supported by MICINN projects PID2019-108528RB-C21 *ProCode*, TED2021-132464B-I00 *PRODIGY*, and FJC2021-047102-I, the European Union NextGenerationEU/PRTR, and the Tezos foundation.

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 381, 2023, pp. 55–57, doi:10.4204/EPTCS.381.6

```
inf(X, top)  | X = []           ==> inf(X, 0.Inf).  # empty
inf(L, X) | L = [H|T]           ==> inf(T, X).      # list_const1
inf(T, X) | L = [H|T], H =< X   ==> inf(L, H).      # list_const2
inf(L, X) | L = S               ==> inf(S, X).      # unif_prop
inf(L, X) | Y =< X              ==> inf(L, Y).      # reduction
inf(X, A) ; inf(X, B) | A =< B ==> inf(X, A).      # lub_1
inf(X, A) ; inf(X, B) | A >= B ==> inf(X, B).      # lub_2
```

Figure 2: A subset of the inf-domain rules.

define $OR(\mathscr{L},\mathscr{C}_1,\ldots,\mathscr{C}_n)$ as the set of rules of the form $l_1;\ldots;l_n \mid g_1,\ldots,g_l \Rightarrow r_1,\ldots,r_m \,\#\, label$, where s, t, n, m, and l are arbitrary positive integers, and the rule meets the following condition:

$$\forall i,j,k \quad \text{s.t. } i \in \{1,\ldots,n\}, j \in \{1,\ldots,m\} \text{ and } k \in \{1,\ldots,l\} :$$
$$\exists v \in \{1,\ldots,t\}\, \exists u \in \{1,\ldots,s\} \text{ s.t. } (l_i \in \mathscr{L}_v, r_j \in \mathscr{L} \text{ and } g_k \in \mathscr{C}_u)$$

Notice that while in *AND*−rules all the elements l_i belong to the same set of constraints $\mathscr{L}$, in the *OR*-rules they belong to (possibly) different subsets of a set of constraints $\mathscr{L}$. The operational meaning of *AND*−rules is similar to that of rewriting rules. If the left side holds in the set where the rule is being applied to, and the guards also hold, then the left-side elements are replaced by the right-side elements. The operational meaning of *OR*−rules is similar, but instead of rewriting over the "same" set the right-side elements are written in a "new" set. When no more rules can be applied, different strategies can be followed. In general we can rewrite the pending elements to a given element or simply delete them.
In the context of abstract interpretation the sets of constraints that we have mentioned have to be seen as abstract domains being the rules applied then over abstract substitutions/constraints. *AND-rules* are intended to capture the behaviour of operations over one abstract substitution with the knowledge that can be inferred from other substitutions that meet the guards. This is useful for example when defining the *greatest lower bound*. Moreover, these rules are also useful for refining abstract substitutions, performing abstractions, combining different abstract domains, etc. On the other hand, *OR-rules* are intended to capture the behaviour of operations applied over multiple abstract substitutions of an abstract domain, such as the *least upper bound* or the *widening*.

```
1 partition([], _, [], []).
2 partition([E|R], C, Left,
      [E|Right1]) :-
3     E >= C,
4     partition(R, C, Left, Right1).
5 partition([E|R], C, [E|Left1],
      Right) :-
6     E < C,
7     partition(R, C, Left1, Right).
```

Figure 1: A Prolog program.

An example. Fig. 1 shows a classic Prolog predicate for partitioning a list. A call `partition(L, X, L1, L2)` is expected to satisfy some properties; for example, that $\forall v \in L2, X \leq v$, which we can express as `inf(L2, X)`. With the help of two auxiliary domains to deal with structures containing variables and with constraints (resp. *depth*$-k$ and *polyhedra*) we can derive an abstract domain for the `inf/2` property. A subset of the rules can be seen in Fig. 2. These rules allow, when connected with the abstract domain operations, to exploit the information gathered from the previous domains and use it to infer `inf(L2, X)`. Similarly, we can also capture the equivalent `sup(L1, X)`, or *multiset* properties capturing that $L \subseteq L1 \cup L2$ and $L1 \cup L2 \subseteq L$. Moreover, we can infer the *sortedness* property for the classical *quicksort* implementation.

Conclusions & future work. We have presented a framework for simplifying the development of abstract domains for logic programs in the context of abstract interpretation frameworks, and concretely that of `CiaoPP`. While some domains are easier to specify with a rule-based language, keeping a constraint-based representation for abstract substitutions may not be efficient compared with specialized representations and operations. In this respect, we plan to explore the use of rules both as an input language for

abstract domain compilation and as a specification language for debugging or verifying properties of hand-written domains. In our experience so far, the proposed approach seems promising for prototyping and experimenting with new domains, enhancing the precision for particular programs, and adding domain combination rules, without the need for understanding the analysis framework internals.

References

[1] M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens & W. Simoens (1996): *Global Analysis of Constraint Logic Programs*. ACM Transactions on Programming Languages and Systems 18(5), pp. 564–615.

[2] M. Bruynooghe (1991): *A Practical Framework for the Abstract Interpretation of Logic Programs*. Journal of Logic Programming 10, pp. 91–124.

[3] P. Cousot & R. Cousot (1977): *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. In: ACM Symposium on Principles of Programming Languages (POPL'77), ACM Press, pp. 238–252, doi:10.1145/512950.512973.

[4] Emanuele De Angelis, Fabio Fioravanti, John P. Gallagher, Manuel V. Hermenegildo, Alberto Pettorossi & Maurizio Proietti (2021): *Analysis and Transformation of Constrained Horn Clauses for Program Verification*. Theory and Practice of Logic Programming FirstView, pp. 1–69, doi:10.1017/S1471068421000211. Available at `http://arxiv.org/abs/2108.00739`.

[5] I. Garcia-Contreras, J. F. Morales & M. V. Hermenegildo (2021): *Incremental and Modular Context-sensitive Analysis*. Theory and Practice of Logic Programming 21(2), pp. 196–243, doi:10.1017/S1471068420000496. Available at `https://arxiv.org/abs/1804.01839`.

[6] R. Giacobazzi & F. Ranzato (2022): *History of Abstract Interpretation*. IEEE Ann. Hist. Comput. 44(2), pp. 33–43. Available at `https://doi.org/10.1109/MAHC.2021.3133136`.

[7] M. V. Hermenegildo, G. Puebla, K. Marriott & P. Stuckey (2000): *Incremental Analysis of Constraint Logic Programs*. ACM Transactions on Programming Languages and Systems 22(2), pp. 187–223.

[8] D. Jurjo, J. F. Morales, P. Lopez-Garcia & M.V. Hermenegildo (2022): *A Rule-based Approach for Designing and Composing Abstract Domains*. Technical Report, CLIP Lab, IMDEA Software Institute.

[9] A. Kelly, K. Marriott, H. Søndergaard & P.J. Stuckey (1998): *A Practical Object-Oriented Analysis Engine for CLP*. Software: Practice and Experience 28(2), pp. 188–224.

[10] B. Le Charlier & P. Van Hentenryck (1994): *Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog*. ACM Transactions on Programming Languages and Systems 16(1), pp. 35–101.

[11] K. Muthukumar & M. Hermenegildo (1989): *Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation*. In: 1989 North American Conference on Logic Programming, MIT Press, pp. 166–189.

[12] K. Muthukumar & M. Hermenegildo (1990): *Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs*. Technical Report ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759. Available at `http://cliplab.org/papers/mcctr-fixpt.pdf`.

[13] K. Muthukumar & M. Hermenegildo (1992): *Compile-time Derivation of Variable Dependency Using Abstract Interpretation*. Journal of Logic Programming 13(2/3), pp. 315–347.

[14] P. Van Roy & A. M. Despain (1990): *The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler*. In: North American Conference on Logic Programming, MIT Press, pp. 501–515.

[15] R. Warren, M. Hermenegildo & S. K. Debray (1988): *On the Practicality of Global Flow Analysis of Logic Programs*. In: Fifth International Conference and Symposium on Logic Programming, MIT Press, pp. 684–699.