Towards Compilation-Based Grounding of Answer Set Programming

Bernardo Cuteri

DeMaCS, Università della Calabria, Rende (CS), Italy
cuteri@mat.unical.it

Abstract. In this paper, I present a major goal of my Ph.D. project and a recent work which summarizes what I have obtained so far towards such goal during my first two years of Ph.D.. In particular, this paper is about a novel technique that intends to enhance grounding in Answer Set Programming by a compilation-based approach. The grounding of ASP programs, as well as the evaluation of logic programs in general, is traditionally implemented in monolithic systems that are general-purpose in the sense that they are able to process an entire class of programs. We follow a different approach by exploiting a compilation procedure and so far we implemented this procedure for Datalog, which is a core fragment of ASP. By using our compilation technique we are able to generate a problem-specific executable implementation of a given (non-ground) Datalog program. Experimental results show the performance benefits that can be obtained by a compilation-based approach.
Most part of this paper is a summary of [1] which is an accepted paper to be presented at AI*IA 2017 [1]
I work under the supervision of prof. Francesco Ricca from the Department of Mathematics and Computer Science at the University of Calabria (Italy).

Keywords: Compilation, Logic programming, Stratified programs, Deductive databases

1 Introduction

Logic programming has proven to be very successful at tackling many computational problems in AI, and this fact lies in the inherent properties of logic languages and also in the implementation of efficient systems [6]. Yet, many modern applications might involve very large inputs requiring very efficient techniques to be processed.

The evaluation of logic programs is traditionally performed by systems that are *general-purpose* in the sense that they are able to process an entire class of programs. We follow a different approach; we consider a compilation strategy for the evaluation of a well-known class of logic programs.

[1] Work in collaboration with prof. Francesco Ricca and Dott. Francesco Alessandro De Rosis (DeMaCS, University of Calabria, Italy).

Being the grounding of Answer Set Programs a long-term goal of my Ph.D. carrier, Datalog evaluation comes as a natural step towards it. Datalog is indeed a kernel sub-language of Answer Set Programming (ASP) [2]. Notably, core modules of ASP systems are based on algorithms for evaluating stratified Datalog programs [9], and, basically, any monolithic ASP system is also an efficient engine for this class of programs.

ASP is a logic formalism for Knowledge Representation and Reasoning with roots in non-monotonic logics and allows to express and solve problems in a declarative fashion. State-of-the-art solvers for Answer Set Programming adopt the so-called *ground+solve* approach in which a grounder module transforms the input program (containing variables) in an equivalent variable-free one, whose models are subsequently computed by the solver module. Pure Datalog admits no negation. We instead focus on Datalog with stratified negation [5, 12] which allows to model more interesting problems and is closer to ASP, where negation is not constrained to be stratified. Datalog by itself is sufficiently expressive to model many practical problems, and it recently found new applications in a variety of classical and emerging domains such as ontologies [3], data integration, information extraction, networking, program analysis, security, and cloud computing (cf. [8]).

A common way of solving a problem in logic programming includes the modeling of the problem in terms of two parts, the first contains rules and the second only facts. Whereas rules allow the declarative specification of the problem, facts can be used to model problem instances. It is custom in the logic programming community to refer to the rules of a program as the *intensional part* (or *encoding*), while the set of facts is referred to as the *extensional part* (or *instance*). Basically, the same uniform encoding is used several times with different instances. A general-purpose system (often needlessly) processes the same encoding every time a new instance is processed. By compiling the encoding in a specialized procedure, one can wire it inside the evaluation procedure so that one does not have to process it every time. Moreover, specialized evaluation strategies on a per-rule basis can be determined at compilation time, possibly increasing the evaluation performance.

We developed *lp2cpp* which is a compiler for stratified Datalog programs. In our prototype implementation, both the compiler and its output (the compiled logic programs) are written in C++. Figure 1 shows a high-level architecture of Datalog compilation and evaluation as designed in our system. The compiler takes as input a stratified program and generates a C++ procedure which is then compiled into an executable implementation that follows a bottom-up evaluation strategy. The compiled program can then be run on any instance of the input problem to retrieve solutions.

To assess the performance of *lp2cpp*, we have compared our implementation against existing general-purpose systems capable of handling stratified logic programs obtaining encouraging results. The experimental analysis has been carried out on benchmarks from OpenRuleBench [11], which is a well-known suite for logic programming engines.

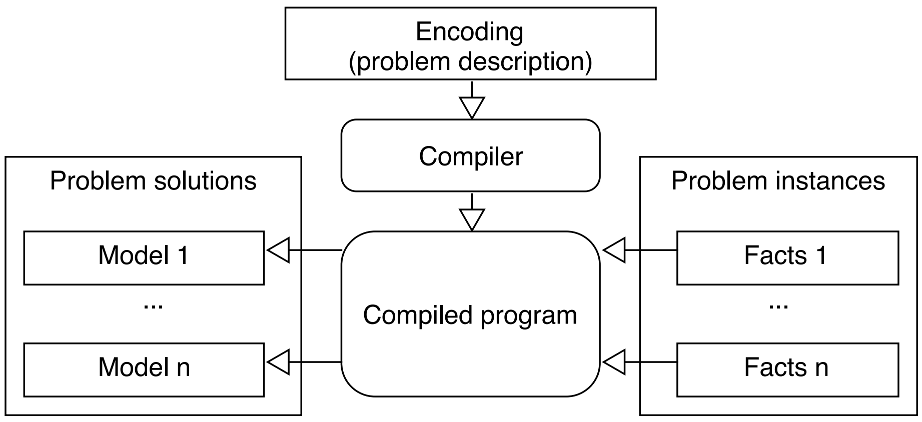

Fig. 1. The *lp2cpp* architecture.

2 Compilation of Datalog programs

Given a stratified Datalog program Π, the compilation procedure takes as input Π and generates imperative code implementing the evaluation of Π on an input set of facts. The first step of the procedure consists in the computation of a topologically ordered list of the strongly connected component of the *dependency graph* (DG) of the input program Π; then our compilation procedure generates the declaration of data structures.

Two types of data structures are used: *predicate sets* and *auxiliary maps*. Predicate sets are used to store ground atoms that will be part of the output, grouped by predicate. The operations associated with predicate sets are insertion, iteration and containment check. Since predicate sets correspond to mathematical sets of ground atoms, insertion avoids duplicates. Auxiliary maps are key-value maps, where keys are tuples and values are sets of tuples. They are used to access tuples of a given predicate that matches a given key. We introduced auxiliary maps since rules are evaluated as nested-joins [12] of the body literals appearing in them, and we would like to perform join operations efficiently.

The auxiliary maps are always maintained up-to-date with respect to its predicate set, i.e. every time we insert a new ground atom of a given predicate p in its predicate set, we then insert the same tuple into every auxiliary map of p.

After computing SCCs and after declaring data structures in terms of predicate sets and auxiliary maps, the compiler prints a procedure which is able to read input facts and load predicate sets and auxiliary maps with them.

Finally, it prints the evaluation of rules in Π.

The computation follows a topological order of the SCCS of the dependency graph of Π. For each component C in $SCCs$ we loop over the rules in the module of C (i.e. the rules whose head belongs to C). Then we print the statements performing the evaluation of each rule of a module, and this is done for each starter predicate of a rule. Intuitively, the join operation starts from a starter predicate whose definition is given in terms of the one of exit and recursive rule. A rule r in a component C is an exit rule if and only if all predicates that appear in the body of r belong to a component that precedes C. This means that the body predicates are already computed when the rule is processed. Otherwise, r is said to be recursive (i.e. there is some body predicate in the body of r that belongs to C). The first predicate is the starter for an exit rule, whereas all recursive predicates are starters for recursive rules. Thus, the evaluation procedure of a rule iterates over all atoms in the predicate set of the starter and implements a nested join starting from it.

The evaluation of recursive rules adds an external loop that continues until no new atoms of the predicates in C are generated.

Finally, the code needed to print the result is added to the output.

3 Experimental analysis

To assess the potential of our tool, we performed an experimental analysis where we compared our system against existing general-purpose systems that can evaluate stratified logic programs by using bottom-up strategies, namely: *Clingo* [7], *DLV* [10] and *I-DLV* [4]. *Clingo* and *DLV* are two well-known ASP solvers, while *I-DLV* is a recently-introduced grounder. Even though the target language of such systems is *ASP* they can also be considered as rather efficient implementations for stratified logic programs.

The experimental analysis has been carried out on benchmarks from OpenRuleBench [11], which is a well-known suite for rule engines.

Results are summarized in Table 1 and in Figure 2. In particular, Table 1 reports both the number of instances solved and the sum of the execution times for each compared tool. The totals are reported in the bottom of the table, whereas the performance obtained in each class of Datalog programs from OpenRuleBench with different features is reported in a separate row. By looking at the table, we observe that our tool solves more instances than any alternative on the overall, and is the fastest on average on all benchmark sets. For completeness, we report that compilation required 2.6s on average, considering all benchmarks, but one. The exceptional benchmark is the wine problem, which

Table 1. Average execution times in seconds and number of solved instances grouped by solvers and domains, best performance outlined in bold face.

		lp2cpp		**Clingo**		**DLV**		**I-DLV**	
Benchmark	**Inst.**	**Time**	**Sol.**	**Time**	**Sol.**	**Time**	**Sol.**	**Time**	**Sol.**
LargeJoins	20	**319**	**18**	677	15	578	16	637	15
Recursion	20	**337**	**19**	510	18	458	19	356	19
Stratified negation	5	**72**	**5**	195	5	237	5	76	5
Totals	45	**300**	**42**	545	38	484	40	444	39

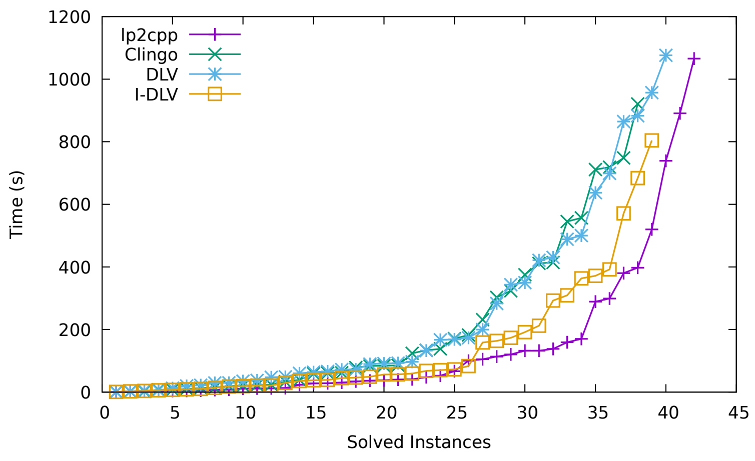

Fig. 2. Overall performance: Cactus plot of the execution times.

has an encoding of almost 1000 rules and takes 12 minutes to compile. This performance is acceptable given that compilation is intended as a one-time process in our approach. An aggregate view of the results is reported in Figure 2 by means of a cactus plot. Cactus plots are customarily used to report the overall performance of tools in competitions. A point (x, y) belongs to a line in the plot whenever the corresponding method solved x instances in less than y seconds. Figure 2 confirms that *lp2cpp* performs well on the overall both in terms of speed and number of instances.

4 Conclusions and future works

In this paper, I presented a major research goal of my Ph.D. project and what I have obtained so far during my first two years of Ph.D.. In particular, I aim at applying compilation-based techniques to the grounding of ASP programs. So far I started working on the application of compilation-based techniques to Datalog with stratified negation, which is a kernel sub-language of ASP. The result is materialized in *lp2cpp*[2]: a compiler for stratified logic programs[3]. *lp2cpp* takes as input a stratified Datalog program and generates a specialized implementation for that program. Experimental results are very encouraging.

[2] The tool can be downloaded from http://goo.gl/XhZXWh.

[3] Work in collaboration with prof. Francesco Ricca and Dott. Francesco Alessandro De Rosis (DeMaCS, University of Calabria, Italy).

References

1. Bernardo Cuteri, A.F.D.R., Ricca, F.: lp2cpp: A tool for compiling stratified logic programs. In: 16th International Conference of the Italian Association for Artificial Intelligence, AI*IA 2017, Bari, Italy, November 14-17, 2017. (To appear.) (2017)
2. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Commun. ACM 54(12), 92–103 (2011)
3. Calì, A., Gottlob, G., Lukasiewicz, T., Pieris, A.: Datalog+/-: A family of languages for ontology querying. In: Datalog. Lecture Notes in Computer Science, vol. 6702, pp. 351–368. Springer (2010)
4. Calimeri, F., Fuscà, D., Perri, S., Zangari, J.: I -dlv: The new intelligent grounder of dlv. In: AI*IA. LNCS, vol. 10037, pp. 192–207. Springer (2016)
5. Ceri, S., Gottlob, G., Tanca, L.: Logic Programming and Databases. Springer (1990)
6. Erdem, E., Gelfond, M., Leone, N.: Applications of answer set programming. AI Magazine 37(3), 53–68 (2016)
7. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Wanko, P.: Theory solving made easy with clingo 5. In: ICLP 2016 TCs. pp. 2:1–2:15 (2016)
8. Huang, S.S., Green, T.J., Loo, B.T.: Datalog and emerging applications: an interactive tutorial. In: Proceedings of SIGMOD 2011. pp. 1213–1216. ACM (2011)
9. Kaufmann, B., Leone, N., Perri, S., Schaub, T.: Grounding and solving in answer set programming. AI Magazine 37(3), 25–32 (2016)
10. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM TOCL 7(3), 499–562 (2006)
11. Liang, S., Fodor, P., Wan, H., Kifer, M.: Openrulebench: an analysis of the performance of rule engines. In: Proceedings of WWW 2009. pp. 601–610. ACM (2009)
12. Ullman, J.D.: Principles of Database and Knowledge-Base Systems. Computer Science Press (1988)