A compiler for stratified logic programs: preliminary report

Bernardo Cuteri and Francesco Ricca

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

Abstract. The evaluation of logic programs is traditionally implemented in monolithic systems that are general-purpose in the sense that they are able to process an entire class of programs. In this paper, we follow a different approach; we present a tool that is able to compile a given (non-ground) logic program into a problem-specific executable implementation. An experimental analysis shows the performance benefits that can be obtained by a compilation-based approach.

Keywords: Compilation, Stratified programs, Deductive databases

1 Introduction

Logic programming has proven to be very successful at tackling many computational problems and this fact lies in the inherent properties of logic languages and also in the implementation of efficient systems. 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. In this paper, we follow a different approach; we consider a compilation strategy for the evaluation of a well-known class of logic programs. The target language of this work is Datalog with stratified negation [4, 12]: a simple, yet flexible logic language for deductive databases. This language is sufficiently expressive to model many practical problems, and it recently found new applications in a variety of emerging domains such as data integration, information extraction, networking, program analysis, security, and cloud computing [6]. Moreover, it represents the kernel sub-language of Answer Set Programming (ASP) [2]. Notably, the grounder modules of ASP systems are based on algorithms for evaluating stratified Datalog programs [7], and, basically, any monolithic ASP system is also an efficient engine for this class of programs.

A Datalog program with stratified negation [4] P is a set of rules of the form $Head :- Body$, where $Body$ is a conjunction of literals, and $Head$ is an atom. A literal is either a positive or a negative atom, while an atom is either a propositional variable or a n-ary predicate p with a list of terms $t_1, ..., t_n$. A term is either a constant or a variable. If a variable appears in the head of a rule or in a negative literal, it must then appear also in some positive literal of the same rule (safety), and recursion through negation is not allowed (stratification). Programs are virtually split into two distinct parts: the intensional part (i.e. the set of possibly non-ground rules) that describes the problem, and the extensional part (i.e. the set of ground facts) that represent an instance of the problem. As an example, the following program models the well-known Reachability problem:

```
reaches(X,Y) :- edge(X, Y).
reaches(X,Y) :- edge(X,Z), reaches(Z,Y).
```

Facts of the form `edge(i,j)` for each arc `(i,j)` model the input graph. It is custom in the logic programming community to refer to the instensional part as the *encoding*, and to the extensional part as the *instance*. In applications, it is common to use the same uniform encoding 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.

Actually, the idea of compiling Datalog programs is not new [1, 4, 8]; however, a new complexity-wise optimal compilation-based approach has been recently proposed in [11]. There, Liu and Stoller describe a method for transforming Datalog programs (with rules having at most two literals in the body) into efficient specialized implementations.

We extend the approach in [11] in order to handle *rules with bodies of any size, stratified negation and inequalities*. Moreover, *we implemented a concrete system* entirely developed in C++. In our prototype implementation, both the compiler and its output (the compiled logic programs) are written in C++.

To assess the potential of our tool, we performed an experimental analysis where we compare our implementation against existing general-purpose systems capable of handling stratified logic programs.

2 Compilation of stratified logic programs

In this section, we first overview the evaluation strategy adopted by our system, and then we present the compilation strategy by means of an example.

Evaluation strategy. Stratified logic programs are evaluated in our approach by following the classical bottom-up schema [12]. Basically, rules are applied starting from the known facts to deduce new facts until no new information can be derived. The notion of *dependency graph* of the input program is used both to improve efficiency of the evaluation and to determine a correct order of evaluation of rules in presence of negation. In more detail, given a program P the dependency graph $DG =< V, E >$ of P has a vertex $p \in V$ for each IDB predicate p, and a (direct) edge of the form $(b, h) \in E$ whenever b occurs in the body and h in the head of a rule of P. Edges are labeled as negative if the

Algorithm 1 Reachability compiled program

```
...{Data structures initializations and facts reading}

//Evaluation of rule (1)
while $Wedge \neq \emptyset$ do
    $edge = Wedge.pop()$
    $Redge.insert(edge)$
    $EdgeZMap.insertKeyValue(\{edge[1]\}, \{edge[0]\})$
    $Wreaches.insert(\{edge[0], edge[1]\})$
end while

//Evaluation of rule (2)
while $Wreaches \neq \emptyset$ do
    $reaches = Wreaches.pop()$
    $Rreaches.insert(reaches)$
    for $X : edgeZMap.at(reaches[0])$ do
        $Wreaches.insert(\{X, reaches[1]\})$
    end for
end while

//Evaluation of rule (3)
while $Wvertex \neq \emptyset$ do
    $vertex = Wvertex.pop()$
    $Rvertex.insert(vertex)$
    if not $Rreaches.contains(\{2, vertex[0]\})$ then
        $Rnoreaches.insert(\{vertex[0]\})$
    end if
end while

//Output
$Model = Redge \cup Rreaches \cup Rvertex \cup Rnoreaches$
```

body literal is negated. The dependency graph is, then, partitioned into strongly connected components (SCC)s. A SCC is a maximal subset of the vertices, such that every vertex is reachable from every other vertex. We say that a rule $r \in P$ defines a predicate p if p appears in the head of r. For each strongly connected component (SCC) of DG, the set of rules defining all the predicates in C is called the module of C. The dependency graph yields a topological order $[C_1], ..., [C_n]$ over the SCCs: for each pair (C_i, C_j) with $i < j$, there is no path in the dependency graph from C_i to C_j. Program modules corresponding to components are evaluated by following a topological order. If two predicates do not belong to the same SCC it means that they do not depend on each other, thus their defining rules can be evaluated separately (possibly increasing evaluation performance). Since, by definition, stratified programs admit no negative edge inside any SCC (i.e. no loop can contain a negative edge), an evaluation performed according to a topological order ensures a sound computation of the semantics of programs with negation. Since the rules of a program module (possibly) need to be processed several times, the evaluation of modules is optimized employing the semi-naïve evaluation technique [12]. Basically, at each iteration n, only the significant information derived during iteration $n-1$ is used.

Compilation by example. The strategy described above is quite standard, and it is employed also by general-purpose implementations. In our approach the same strategy is used to produce a specialized implementation. In the following, we exemplify how the compilation of a program module is done in our system. Consider once more the program P^{GR} that models the Reachability problem with an extra rule to derive all vertices that do not reach vertex 2:

```
(1) reaches(X,Y) :- edge(X, Y).
(2) reaches(X,Y) :- edge(X,Z), reaches(Z,Y).
(3) noReach(Y) :- vertex(Y), not reaches(2,Y).
```

The dependency graph analysis basically ensures that the rule (3) is evaluated after rule (1) and (2). Then , the core part of the compilation process applied to P^{GR}, in pseudo-code is presented in Algorithm 1. There, $Wedge$, $Wreaches$ and $Wvertex$ denote working sets while *Redge*, *Rreaches*, *Rvertex* and *Rnoreaches* denote result sets (cf.[11]). Working sets and result sets are implemented in efficient indexed data structures that provide associative access and efficient insertions and deletions of ground atoms. In the example, we first loop on the working set of predicate *edge* which was previously loaded with facts. In every iteration, we retrieve and remove an element from the working set (*pop*) and we insert it in the result set. At line 6 we insert an edge atom into an auxiliary map (note that, attributes in atom variables, such as *edge*, are accessed in the example by position, where the first attribute has index 0). Auxiliary maps are used to index predicate ground atoms that appear in the body of some rule and might be involved in join operations with other predicates of the same body. In our example, rule (2) induces an indexing of predicate *edge* on attribute Z. At line 8, we derive an instance of reaches because of rule (1) and we insert it in its working set. After the edge loop completes, we can loop on $Wreaches$ and once again we retrieve and remove an atom from the working set and we insert it into the result set. At this point (line 15) we loop over joining edge atoms retrieved from the auxiliary map *EdgeZMap*. Inside the loop, we generate the head ground atom and we insert it in Wreaches. In the last loop (line 21), we iterate over vertex ground atoms and we add into *Rnoreaches* all vertices that do not reach vertex 2 because of rule (3). Note that we do not need an extra auxiliary map for *reaches* because we can directly use Rreaches instead. Finally, the model of the program is given by the union of all result sets.

Datalog Compiler. The compiler is written in C++ and the output is first written in C++ and then compiled into an executable program (using standard C++ compilers). Figure 1 shows a high-level architecture of Datalog compilation and evaluation as designed in our system. The resulting executable program can be run on any instance of the compiled logic program.

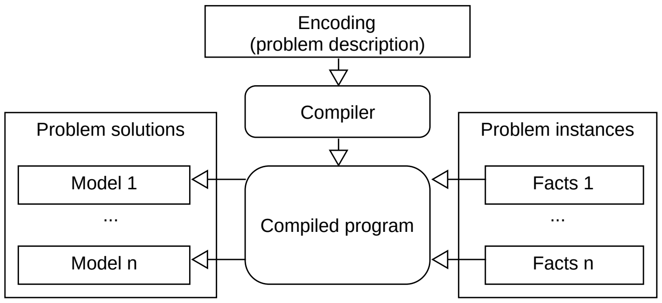

Fig. 1. Datalog compiler architecture.

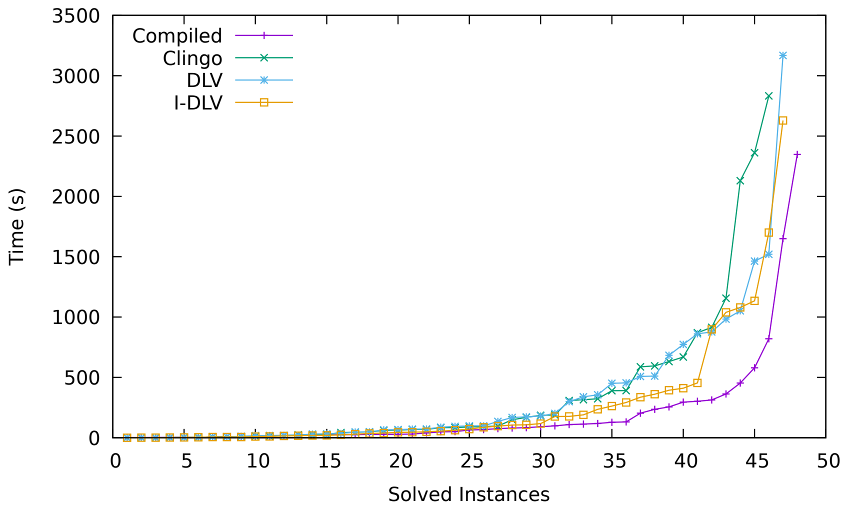

Fig. 2. Overall performance: Cactus plot of the execution times.

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 eval-

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

		compiled		**Clingo**		**DLV**		**I-DLV**	
Benchmark	**Inst.**	**Time**	**Sol.**	**Time**	**Sol.**	**Time**	**Sol.**	**Time**	**Sol.**
LargeJoins	23	272	**22**	418	21	263	21	258	21
Recursion	21	**139**	**21**	331	20	444	21	332	21
Stratified negation	5	109	5	154	5	255	5	**88**	**5**
Totals	49	**197**	**48**	352	46	343	47	273	47

uate stratified logic programs by using bottom-up strategies, namely: *Clingo* [5], *DLV* [9] and *I-DLV* [3]. *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 [10], which is a well-known suite for rule engines. We run all experiments on a Debian Linux server (64bit kernel), equipped with 2.30GHz Intel Xeon E5-4610 v2 Processors and 128GB RAM. Time and memory for each run are limited to 10 minutes cpu-time and 4GB, respectively. The execution times reported for our system include compilation times to provide a fair comparison with general-purpose tools.

Results are summarized in Table 1 and in Figure 2. 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 in Large Joins and Recursion sets. In Stratified negation, our tool is on par with the others in terms of solved instances, but is slower on average than *I-DLV*. This might be explained by the fact that there are still some optimizations, such as join reordering, that general-purpose systems adopt, but we did not consider yet in our prototype implementation. An aggregate view on the results is reported in Figure 2 showing that our tool performs well on the overall. For completeness, we report that compilation required 2.6s on average (over all benchmarks). This performance is acceptable given that compilation is intended as a one-time process in our approach.

4 Conclusions and future works

In this paper, we presented a new compiler for stratified Datalog programs.[1] The prototype takes as input a stratified Datalog program and generates a specialized implementation for a given program. Experimental results are very encouraging. Ongoing work concerns the improvement of data structures generated by our compiler and the inclusion of other known optimization techniques used by ASP grounders [7]. As for future works, we aim at applying compilation-based techniques to the instantiation of ASP programs.

References

1. Arni, F., Ong, K., Tsur, S., Wang, H., Zaniolo, C.: The deductive database system LDL++. TPLP 3(1), 61–94 (2003)
2. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Commun. ACM 54(12), 92–103 (2011)
3. 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)
4. Ceri, S., Gottlob, G., Tanca, L.: Logic Programming and Databases. Springer (1990)

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

5. 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)
6. 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)
7. Kaufmann, B., Leone, N., Perri, S., Schaub, T.: Grounding and solving in answer set programming. AI Magazine 37(3), 25–32 (2016)
8. Kifer, M., Lozinskii, E.L.: On compile-time query optimization in deductive databases by means of static filtering. ACM TDS 15(3), 385–426 (1990)
9. 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)
10. 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)
11. Liu, Y.A., Stoller, S.D.: From datalog rules to efficient programs with time and space guarantees. ACM Trans. Program. Lang. Syst. 31(6), 21:1–21:38 (2009)
12. Ullman, J.D.: Principles of Database and Knowledge-Base Systems. Computer Science Press (1988)