Compilation-based Techniques for Evaluating Normal Logic Programs Under the Well-founded Semantics

Andrea Cuteri, Giuseppe Mazzotta and Francesco Ricca

University of Calabria, Rende 87036, Italy

Abstract
Recent studies have demonstrated that compilation-based techniques can be beneficial for evaluating Datalog and ASP programs. In this paper, we develop a compiler that is able to generate solvers for normal logic programs under the well-founded semantics. The proposed system has been evaluated on different settings and preliminary results highlight significant improvements in the evaluation of non-stratified programs.

Keywords
Logic Programming, Well-founded semantics, Compilation

1. Introduction

Logic programming is a declarative programming paradigm that can be used to model complex problems in terms of logical implications [1]. Logic programs are often evaluated by means of general-purpose systems that implement a given semantics [2, 3]. The need for handling with the same algorithm any possible input, in some cases, makes it impossible to apply specific optimizations that would work only for a subclass of programs. Thus, considerable speedups can be obtained by using ad-hoc evaluation procedures for the program in input. Following this consideration, compilation-based techniques have been recently proposed to speed up the evaluation of Datalog [4] and Answer Set Programming (ASP) [5, 6]. In particular, the idea behind this approach is to compile the input program into a custom system that is optimized by exploiting the syntactic properties of the modeled program and can be used for evaluating different instances of the compiled program. Concerning Datalog, the system soufflé [4] was demonstrated to be very effective, especially for solving tasks connected with software development, but also other prototypical systems were revealed to be very promising [7]. Concerning ASP, compilation techniques have been successfully employed for the evaluation of grounding-intensive ASP programs outperforming state-of-the-art ASP solvers [6, 8] and reducing both time and memory consumption. However, none of the aforementioned systems support the well-known well-founded semantics [9] when no restriction is posed on the usage of negation.

In this paper, we propose a compilation-based technique for evaluating normal logic programs under the well-founded semantics and present a system that is able to generate ad-hoc solvers for programs.

CILC'23: 38th Italian Conference on Computational Logic, June 21–23, 2023, Udine, Italy
cuteri.andrea@gmail.com (A. Cuteri); giuseppe.mazzotta@unical.it (G. Mazzotta); ricca@mat.unical.it (F. Ricca)

An empirical evaluation of the proposed approach has been conducted on hard benchmarks [10, 11]. Obtained results demonstrate that the proposed approach is competitive with existing implementations capable of evaluating both Datalog programs (i.e. positive programs) and program with negation. Notably, our approach outperforms DLV2 [12] on normal logic programs with not-stratified negation.

2. Logic Programs Under the Well-founded Semantics

In this section, some preliminaries are provided on normal logic programs under the well-founded semantics [9].

2.1. Syntax

A *term* is a constant or a variable. *Constants* are strings starting with lowercase letter or integers instead *variables* are terms starting with uppercase letter. An *atom* a is an expression of the form $p(t_1, ..., t_n)$ where p is a predicate of arity n and $t_1, ..., t_n$ are terms. If all the terms are constants then a is a *ground* atom. A *literal* is an atom a, or its negation $not\ a$ where not represents default negation. A literal is *positive* if it is of the form a, *negative* otherwise. Given a literal l, the *complement* of l, denoted by $\bar{l}$, is a if $l = not\ a$, $not\ a$ otherwise. A rule r is an expression of the form:

$$h \leftarrow b_1, ..., b_k,\ not\ b_{k+1},\ ...,\ not\ b_m.$$

where h is an atom referred to as *head*, H_r, $b_1, ...,\ not\ b_m$ is a conjunction of literals referred to as body of the rule, B_r, and $m > 0$. If $m = 0$ then r is a *fact.* A program Π is a set of rules. Given a set of literals B, B^+ (resp. B^-) denotes the set of positive (resp. negative) literals in B. The dependency graph of a program Π, G_Π, is a directed labeled graph where the nodes are predicates appearing in Π and the set of the edges contains a positive (resp. negative) edge (u, v) if exists a rule $v \leftarrow B \in \Pi$ where u appears in B^+ (resp. u appears in B^-). Π is said to be *Datalog* if it does not contain any negative literals; Datalog with *stratified negation* if G_Π does not contain cycles with negative edges.

2.2. Well-founded Semantics

Given a program Π, the *Herbrand Universe* is the set of constants appearing in Π; the *Herbrand Base*, B_Π, is the set of possible ground atoms that can be built using predicate in Π and constants in the Herbrand Universe. Given a rule $r \in \Pi$, $ground(r)$ represents all possible instantiations of r replacing variables with constants in the Herbrand Universe. Given a program Π, $ground(\Pi) = \bigcup_{r\in\Pi} ground(r)$. An interpretation I is a set of literals whose atoms belong to B_Π. I is *consistent* if for each literal $l \in I$, $not\ l \notin I$. I is *total* if for each atom $a \in B_\Pi$ either a or $not\ a$ belongs to I. Given a consistent interpretation I, a literal l is true (resp. false) w.r.t. I if $l \in I$ (resp. $not\ l \in I$). A literal is undefined w.r.t. I if it is neither true nor false. A conjunction of literals is true w.r.t. I if all the literals are true w.r.t. I; it is false if at least one literal is false w.r.t. I; it is undefined otherwise. A set of atoms, U_Π, is an unfounded set w.r.t. I if for every atom $a \in U_\Pi$ and for every rule $r \in ground(\Pi)$ such that $H_r = a$, B_r is false w.r.t. I or B_r contains some positive literals whose atoms belong to U_Π. Intuitively, all rules defining atoms in U_Π have a false body or depend on atoms in U_Π and so they cannot be inferred as true. The greatest unfounded set, $U_\Pi(I)$, is the union of possible unfounded sets U_Π. Let $T_\Pi(I)$ be a transformation defined as the set of atoms a such that there exists a rule $r \in ground(\Pi)$ where $H_r = a$ and B_r is true w.r.t. I, we define $W_\Pi(I) = T_\Pi(I) \cup \neg U_\Pi(I)$ where $\neg U_\Pi(I) = \{not\ l \mid l \in U_\Pi(I)\}$. Let $I_0 = \emptyset$, $I_{\alpha+1} = \mathcal{W}_\pi(I_\alpha)$, with $\alpha \geq 0$, the (partial) well-founded model is defined as $\mathcal{W}_\pi = I_\beta$ where $\beta \geq 0$ is the smallest ordinal such that $I_\beta = I_{\beta+1}$. Basically, $\mathcal{W}_\pi$ is the least fixed point $\mathcal{W}_P$ [9]

Algorithm 1 compileProgram

```
Input  : A normal program $P$
Output : Prints evaluation procedure for $P$
begin
    $\ll$def $computeWellFounded(facts)$$\gg$
    $\ll$    $I = facts$$\gg$
    $\ll$    $\mathcal{B} = \emptyset$$\gg$
    $DG = computeDG(P)$
    $SCC = computeSCC(DG)$
    for all $C \in SCC$ do
        $\ll$$d\_stack = F = \emptyset$$\gg$
        for all $r \in computeRulesForComponent(P, C)$ do
            $compileRuleForHeadDerivation(B_r, H_r, C)$
        $\ll$while $d\_stack \neq \emptyset$ do$\gg$
        $\ll$    $starter = d\_stack.pop()$$\gg$
        $\ll$    switch $pred(starter)$$\gg$
        for all $r \in computeRecursiveRules(P, C)$ do
            for all $i \in [1, \ldots, |B_r|]$ do
                if $pred(B_r[i]) \in C \wedge B_r[i]$ is positive literal then
                    $\ll$case $[\![pred(B_r[i])]\!]_\lightning$$\gg$
                    $\ll$    $\sigma = \epsilon$$\gg$
                    for all $j \in 1, ..., |trm(B_r[i])|$ do
                        if $trm(B_r[i])[j]$ is variable then
                            $\ll$    $\sigma = \sigma \cup \{ [\![trm(B_r[i])[j]]\!]_\lightning \mapsto trm(starter)[ [\![j]\!]_\lightning ]\}$$\gg$
                    $compileRuleForHeadDerivation(B_r \setminus \{B_r[i]\}, H_r, C)$
        $\ll$done$\gg$
        $\ll$do$\gg$
        $\ll$    $loop = \bot$$\gg$
        $\ll$    for $starter \in \mathcal{B}$ do$\gg$
        $\ll$        $true = undef = \bot$$\gg$
        $\ll$        $starter = s\_stack.pop()$$\gg$
        $\ll$        switch $pred(starter)$$\gg$
        $compileRulesForSupportDerivation(computeRecursiveRules(P, C), C)$
        $\ll$    done$\gg$
        $\ll$while $loop == \top$ $\gg$
    $\ll$end def$\gg$
```

Algorithm 2 compileRuleForHeadDerivation

```
Input  : A list of literals $B$, an atom $a$, a set of predicates $C$
Output : Prints a procedure that instantiates $B$ and derives new atoms matching $a$
1 begin
2     $compileRuleBody(B, C)$
3     $printHeadDerivation(B, a)$
4     for all $i \in [|B|, \dots, 1]$ do
5         if $B[i]$ is positive literal then
6             $\ll \quad \sigma = \sigma\ [\![j]\!]_{\lightning} \gg$
7             $\ll$done$\gg$
8         else
9             $\ll$fi$\gg$
```

3. Compilation of Well-founded Semantics

In this section, we describe the compilation procedure for generating an ad-hoc solver for an input program Π. In particular, proposed algorithms (see Algorithms 1-5) describe the compilation of a normal program with not-stratified negation. However, our approach is also able to generate simplified code for the case of Datalog programs with stratified negation that is indeed a particular case. Reported algorithms follow the syntactic convention used in [8]. To recall, the code enclosed between $\ll\gg$ is printed by the compiler as it is. Instead, the code enclosed in $[\![\,]\!]_{\lightning}$, is first substituted with its run-time value and then is printed. For example, let $B_r[i] = a(X)$, Algorithm 1 at line 17 prints "**case** "a" :".

Description of the Compilation Algorithms. As the first step, the compiler builds the dependency graph of Π (Alg. 1 lines 5-6) and computes its strongly connected components SCCs, $C_1, ..., C_n$, that give us a topological order of G_Π such that no paths exist from C_j to C_i if $i < j$. By following that order, the compiler produces for each component C the code that evaluates the rules defining atoms in C (i.e., whose predicate belongs to C), referred to as P_C. Then, each rule r is compiled into a sub-procedure that iterates over possible instantiations of B_r that are either true or undefined w.r.t. $I \cup \mathcal{B}$ and, successively, derives H_r (Alg. 1 lines 9-10). Such sub-procedures are generated by means of Algorithm 2. Algorithm 2 starts by calling Algorithm 3 that prints different nested join loops or if statements for each literal in B_r. These nested blocks implement iterations over possible rule instantiations (Alg. 3 lines 3-13). Inside the deepest block, the code that collects negative dependencies within the component C into the set ns is printed (Alg. 3 lines 15-19). Then Algorithm 2 prints the code that derives new atoms matching H_r by calling Algorithm 4. In particular, the code generated by Algorithm 4 checks if the current body (i.e. $t_1, \cdots, t_n$) is true w.r.t. $I \cup \mathcal{B}$. In particular, if all positive literals belong to I, no negative literals are undefined (i.e. $\neg b^- \cap \mathcal{B}$), and no negative literals in the same component occur in the current body (i.e. $ns = \emptyset$), then all literals $t_1, \cdots, t_n$ are true w.r.t. I. Thus, if it is the case then the head of the rule is derived as true, otherwise it is derived as undefined (code generated by lines 5 and 10 of Alg. 4). In both cases, derived atoms are collected into the derivation stack in order to be used in the second derivation phase. As the last step, for each literal $l \in B_r$, if l is a positive literal then a nested for-loop is closed by restoring the variable substitution σ (Alg. 2 lines 5-7). Otherwise, an if-statement scope is closed (Alg. 2 lines 8-9). The second derivation scenario is generated by looking at recursive rules defining atoms whose predicate belongs to C. In this case, the generated procedure will consume literals collected into d_stack and, for each of them, different sub-procedures are executed according to the predicate name of the consumed literal (Alg. 1 lines 11-13). In particular, the compiler, for each rule r, generates different switch-cases for each literal $l \in B_r^+$ whose predicate belongs to C (Alg. 1 lines 15-22). Inside each case, a sub-procedure that evaluates r starting from a literal, $starter$, that matches l is generated. Each sub-procedure starts with the initialization of a variable substitution σ from variables in l to constant in $starter$ (Alg. 1 lines 18-21) and then contains the code that evaluates the remaining part of B_r, generated by Algorithm 2. In this way, the generated procedure is able to simulate a semi-naive evaluation of recursive rules. Out of the while-loop scope, the compiler generates the code that derives, if it is possible, undefined atoms either as true or false (Alg. 1 lines 24-32). At this point, atoms in B_Π with predicates in C that do not belong to $I \cup B$ are considered false since no rule instantiations that can derive them exists. Thus, the generated procedure, in this case, will iterate until some undefined atoms (i.e. atoms belonging to $\mathcal{B}$) are derived either as true or false. For each iteration, the generated procedure evaluates possible rules defining remaining atoms in $\mathcal{B}$ and so different switch-cases, one for each recursive rule r, are generated. Each case contains the procedure that evaluates a rule r starting from an undefined atom $starter$ that can be substituted to H_r. These sub-procedures are generated by Algorithm 5 that, for each switch-case, prints the code that initializes a variable substitution σ from variables in H_r to constants in $starter$ (Alg. 5

Algorithm 3 compileRuleBody

```
Input  : A list of literals $B$, a set of predicate $C$
Output : Prints the instantiation procedure for the rule body $B$
1  begin
2      $\ll \sigma = \epsilon \gg$
3      for all $j \in 1, ..., |B|$ do
4          $\ll \sigma_{\llbracket j \rrbracket_{\lightning}} = \sigma \gg$
5          if $B[j]$ is a positive literal then
6              $\ll T_{\llbracket j \rrbracket_{\lightning}} = \{p \in (I \cup \mathcal{B}) \mid match(\sigma(\llbracket B[j] \rrbracket_{\lightning}), p)\} \gg$
7              $\ll$for all $t_{\llbracket j \rrbracket_{\lightning}} \in T_{\llbracket j \rrbracket_{\lightning}}$ do$\gg$
8              for all $k \in 1, ..., |trm(B[j])|$ do
9                  if $trm(B[j])[k]$ is variable then
10                     $\ll \sigma = \sigma \cup \{ \llbracket trm(B[j])[k] \rrbracket_{\lightning} \mapsto trm(t_{\llbracket j \rrbracket_{\lightning}})[\llbracket k \rrbracket_{\lightning}]\} \gg$
11         else
12             $\ll t_{\llbracket j \rrbracket_{\lightning}} = \sigma(\llbracket B[j] \rrbracket_{\lightning}) \gg$
13             $\ll$if $\overline{t_{\llbracket j \rrbracket_{\lightning}}} \notin I$ then$\gg$
14     $\ll b = \emptyset \gg$
15     $\ll ns = \emptyset \gg$
16     for all $j \in 1, ..., |B|$ do
17         $\ll b = b \cup \{t_{\llbracket j \rrbracket_{\lightning}}\} \gg$
18         if $B[j]$ is a negative literal $\wedge\ pred(B[j]) \in C$ then
19             $\ll ns = ns \cup \{t_{\llbracket j \rrbracket_{\lightning}}\} \gg$
```

Algorithm 4 printHeadDerivation

```
Input  : An atom $a$
Output : Prints the derivation procedure for new atoms matching $a$
begin
    $\ll h = \sigma(\ [\![a]\!]_{\not{t}}\ )\gg$
    $\ll$if $ns == \emptyset \wedge b^{+} \subseteq I \wedge (\neg b^{-} \cap \mathcal{B}) == \emptyset$ then$\gg$
    $\ll$    $d\_stack = d\_stack \cup \{h\}\gg$
    $\ll$    $I = I \cup \{h\}\gg$
    $\ll$    $\mathcal{B} = \mathcal{B} \setminus \{h\}\gg$
    $\ll$else$\gg$
    $\ll$    if $h \notin (I \cup \mathcal{B})\gg$
    $\ll$        $d\_stack = d\_stack \cup \{h\}\gg$
    $\ll$        $\mathcal{B} = \mathcal{B} \cup \{h\}\gg$
    $\ll$    fi$\gg$
    $\ll$fi$\gg$
```

lines 4-7) and then prints the nested block needed for evaluating B_r by means of Algorithm 3. Then, inside the last nested level, the code for head derivation is generated (Alg. 5 lines 9-16). If the conjunction $t_1, \cdots, t_n$, with $n = |B_r|$, is true w.r.t. $I \cup \mathcal{B}$ (i.e. all positive literals are in I and all atoms appearing in some negative literals are not in $\mathcal{B}$, Alg. 5 line 10) then $starter$ is derived as true, otherwise a flag variable stating that a ground rule with $starter$ as head and an undefined body w.r.t. $I \cup \mathcal{B}$ exists is enabled (Alg. 5 line 15). Then, nested blocks' scopes are closed (Alg. 5 lines 17-22). After evaluating all switch-cases, if neither $starter$ has not been derived as true nor the $undef$ flag is true then $starter$ is derived as false (Alg. 5 lines 23-27).

Example of Compilation. In order to help understanding the outcome of our compiler, an example is described in the following. Consider the following program Π:

$$\begin{array}{rl} r1: & a(Y) \leftarrow b(X,Y), c(Y,Z), not\ d(Z) \\ r2: & a(X) \leftarrow f(X),\ not\ g(X) \\ r3: & g(X) \leftarrow e(X),\ not\ a(X) \end{array}$$

The SCCs of G_Π are $C_0 = \{b\}$, $C_1 = \{c\}$, $C_2 = \{d\}$, $C_3 = \{f\}$, $C_4 = \{e\}$, $C_5 = \{a, g\}$. Since for components C_i, with i from 0 to 4, there are no rules the code produced for them is empty and so let us focus on C_5. In this case, P_{C_5} is the entire program so Algorithm 1 prints a sub-procedure for each rule of Π. Algorithm 6 reports the code produced by Algorithm 2 for rule $r1$.

For evaluating B_{r1}, Algorithm 3 generates an external for-loop that iterates over ground literals t_1 that are not false w.r.t. $I \cup \mathcal{B}$ and match $b(X,Y)$ (Alg. 6 lines 4-35). For every t_1, the variable substitution σ is updated mapping X and Y to the first and the second term of t_1 respectively (Alg. 6 lines 6-7). Nested into this for-loop, another for-loop is printed to iterate over ground literals t_2 matching $\sigma(c(Y,Z))$ (Alg. 6 lines 9-33). Note that the application of σ to a literal will replace mapped variables with the value they are mapped to (i.e. $Y \mapsto 1$ then $\sigma(c(Y,Z)) = c(1,Z)$). Inside this for-loop σ is updated by mapping Z to the second term of t_2 (Alg. 6 lines 11-12). Then, the last literal to evaluate is $not\ d(Z)$ and so, since the value of Z is already fixed by t_2 then, the last nested block is an if-statement that checks whether

Algorithm 5 compileRulesForSupportDerivation

```
Input  : A set of rules $recursiveRules$, a set of predicates $C$
Output : Prints a procedure that search for rule body that can support a given atom
1  begin
2      for all $r \in recursiveRules$ do
3          $\ll$ case $[\![pred(H_r)]\!]_{\lightning}\gg$
4          $\ll$     $\sigma = \epsilon\gg$
5          for all $j \in 1, ..., |trm(H_r)|$ do
6              if $trm(H_r)[j]$ is variable then
7                  $\ll$ $\sigma = \sigma \cup \{ [\![trm(H_r)[j]]\!]_{\lightning} \mapsto trm(starter)[ [\![j]\!]_{\lightning} ]\}\gg$
8          $compileRuleBody(B_r, C)$
9          $\ll$         $h = \sigma( [\![a]\!]_{\lightning} )\gg$
10         $\ll$         if $b^+ \subseteq I \wedge (\neg b^- \cap \mathcal{B}) = \emptyset \wedge h \notin I$ then$\gg$
11         $\ll$             $I = I \cup \{h\}\gg$
12         $\ll$             $\mathcal{B} = \mathcal{B} \setminus \{h\}\gg$
13         $\ll$             $true = loop = \top\gg$
14         $\ll$         else if $((b^+ \cup \neg b^-) \cap \mathcal{B}) \neq \emptyset \wedge h \notin I$ then$\gg$
15         $\ll$             $undef = \top\gg$
16         $\ll$         fi$\gg$
17         for all $i \in [|B_r|, \dots, 1]$ do
18             if $B_r[i]$ is positive literal then
19                 $\ll$ $\sigma = \sigma_{[\![j]\!]_{\lightning}}\gg$
20                 $\ll$done$\gg$
21             else
22                 $\ll$fi$\gg$
23     $\ll$ if $undef = \bot \wedge true = \bot$ then$\gg$
24     $\ll$     $\mathcal{B} = \mathcal{B} \setminus \{starter\}\gg$
25     $\ll$     $F = F \cup \{\overline{starter}\}\gg$
26     $\ll$     $loop = \top\gg$
27     $\ll$ fi$\gg$
```

$\overline{t_3} = \sigma(d(Z))$ is not true w.r.t. I (Alg. 6 lines 14-31). At this point the conjunction $t_1,\ t_2,\ t_3$ is an instantiation of B_r and the generated code should derive the $a(Y)$ (Alg. 6 lines 20-30). Thus, if the body is true w.r.t. $I \cup \mathcal{B}$ then $h = \sigma(a(Y))$ is added to I, otherwise it is added to $\mathcal{B}$. Then, Algorithm 1 produces other two analogues procedures also for $r2$ and $r3$. Since in this case, recursive rules are $r2$ and $r3$, but no positive literals have predicates in C_5 then no switch-cases are generated at all and so the while-loop over d_stack can be omitted. Thus, Algorithm 1 continues by printing the last derivation scenario as previously described. In particular, Algorithm 7 reports the switch-cases generated by Algorithm 5 for rules $r2$ and $r3$ w.r.t. the component C_5. The first switch-case refers to rule $r2$ (Alg. 7 lines 2-24) and so it contains the code for evaluating $r2$ starting from a literal matching the $a(X)$. Thus, σ is initialized by mapping X to the first term of $starter$ literal (Alg. 7 lines 3-4), and then the nested blocks are generated according to B_{r2} (Alg. 7 lines 5-13). Inside the last block, the head derivation code has been printed (Alg. 7 lines 14-21). If the body instantiation $t_1,\ t_2$ is true w.r.t. $I \cup \mathcal{B}$ then $h = starter$ is derived as true, otherwise the $undef$ flag is enabled. The second switch-case is analogous to the first one but it refers to $r3$ and so the evaluation starts from a literal, $starter$, matching the atom $b(X)$. Out of the scope of the switch-statement if no rule instantiations have been founded for the atom $starter$ then it is derived as false (Alg. 7 lines 48-51).

Algorithm 6 Output example compileRuleForHeadDerivation

```
Input  : $\{b(X,Y), c(Y,Z),\ not\ d(Z)\}, a(Y), \{'a','g'\}$
1  begin
2      $\sigma = \epsilon$
3      $\sigma_1 = \sigma$
4      $T_1 = \{p \in (I \cup \mathcal{B}) \mid match(\sigma(b(X,Y)), p)\}$
5      for all $t_1 \in T_1$ do
6          $\sigma = \sigma \cup \{X \mapsto trm(t_1)[1]\}$
7          $\sigma = \sigma \cup \{Y \mapsto trm(t_1)[2]\}$
8          $\sigma_2 = \sigma$
9          $T_2 = \{p \in (I \cup \mathcal{B}) \mid match(\sigma(c(Y,Z)), p)\}$
10         for all $t_2 \in T_2$ do
11             $\sigma = \sigma \cup \{Y \mapsto trm(t_2)[1]\}$
12             $\sigma = \sigma \cup \{Z \mapsto trm(t_2)[2]\}$
13             $\sigma_3 = \sigma$
14             $t_3 = \sigma(not\ d(X))$
15             if $\overline{t_3} \notin I$ then
16                 $b = ns = \emptyset$
17                 $b = b \cup \{t_1\}$
18                 $b = b \cup \{t_2\}$
19                 $b = b \cup \{t_3\}$
20                 $h = \sigma(a(Y))$
21                 if $ns == \emptyset \wedge b^+ \subseteq I \wedge (\neg b^- \cap \mathcal{B}) = \emptyset$ then
22                     $d\_stack = d\_stack \cup \{h\}$
23                     $I = I \cup \{h\}$
24                     $\mathcal{B} = \mathcal{B} \setminus \{h\}$
25                 else
26                     if $h \notin (I \cup \mathcal{B})$
27                         $d\_stack = d\_stack \cup \{h\}$
28                         $\mathcal{B} = \mathcal{B} \cup \{h\}$
29                     fi
30                 fi
31             fi
32             $\sigma = \sigma_2$
33         done
34         $\sigma = \sigma_1$
35     done
```

Algorithm 7 Output example compileRulesForSupportDerivation

```
Input  : $\{g(X) \leftarrow e(X),\ not\ a(X);\ a(X) \leftarrow f(X),\ not\ g(X)\}, \{'a','g'\}$
begin
    case 'a'
        $\sigma = \epsilon$
        $\sigma = \sigma \cup \{X \mapsto trm(starter)[1]\}$
        $\sigma_1 = \sigma$
        $T_1 = \{p \in (I \cup \mathcal{B}) \mid match(\sigma(f(X)), p)\}$
        for all $t_1 \in T_1$ do
            $\sigma = \sigma \cup \{X \mapsto trm(t_1)[1]\}$
            $t_2 = \sigma(not\ g(X))$
            if $\overline{t_2} \notin I$ then
                $b = ns = \emptyset$
                $b = b \cup \{t_1\} \quad b = b \cup \{t_2\}$
                $ns = ns \cup \{t_2\}$
                $h = starter$
                if $b^+ \subseteq I \wedge (\neg b^- \cap \mathcal{B}) = \emptyset \wedge h \notin I$ then
                    $I = I \cup \{h\}$
                    $\mathcal{B} = \mathcal{B} \setminus \{h\}$
                    $true = loop = \top$
                else if $((b^+ \cup \neg b^-) \cap \mathcal{B}) \neq \emptyset \wedge h \notin I$ then
                    $undef = \top$
                fi
            fi
            $\sigma = \sigma_1$
        done
    case 'g'
        $\sigma = \epsilon$
        $\sigma = \sigma \cup \{X \mapsto trm(starter)[1]\}$
        $\sigma_1 = \sigma$
        $T_1 = \{p \in (I \cup \mathcal{B}) \mid match(\sigma(e(X)), p)\}$
        for all $t_1 \in T_1$ do
            $\sigma = \sigma \cup \{X \mapsto trm(t_1)[1]\}$
            $t_2 = \sigma(not\ a(X))$
            if $\overline{t_2} \notin I$ then
                $b = ns = \emptyset$
                $b = b \cup \{t_1\} \quad b = b \cup \{t_2\}$
                $ns = ns \cup \{t_2\}$
                $h = starter$
                if $b^+ \subseteq I \wedge (\neg b^- \cap \mathcal{B}) = \emptyset \wedge h \notin I$ then
                    $I = I \cup \{h\}$
                    $\mathcal{B} = \mathcal{B} \setminus \{h\}$
                    $true = loop = \top$
                else if $((b^+ \cup \neg b^-) \cap \mathcal{B}) \neq \emptyset \wedge h \notin I$ then
                    $undef = \top$
                fi
            fi
            $\sigma = \sigma_1$
        done
    if $undef = \bot \wedge true = \bot$ then
        $\mathcal{B} = \mathcal{B} \setminus \{starter\}$
        $loop = \top$
    fi
```

4. Implementation and Experiments

Implementation Details. The compilation strategy described in the previous section has been entirely implemented in C++ and so both compiler and generated procedures are written in C++. Generated code is built on top of optimized data structures that allow to speed up the whole computation process. In particular, it uses a numerical representation of constant terms that allows a compact and uniform representation. Moreover, different indexing structures are used for each predicate. Indexes are defined on the subset of terms of predicates, for fast retrieval of the list of literals that match a possible tuple.

Benchmarks, Systems and Experiments Setup. In order to assess the performances of the proposed approach we conducted an empirical evaluation both on positive programs (i.e., Datalog) and programs with negation. Among positive programs, we considered:

- Large-join problem [11] defined as follows:

$$\begin{array}{c} a(X,Y) \leftarrow b1(X,Z),\ b2(Z,Y). \\ b1(X,Y) \leftarrow c1(X,Z),\ c2(Z,Y). \\ b2(X,Y) \leftarrow c3(X,Z),\ c4(Z,Y). \\ c1(X,Y) \leftarrow d1(X,Z),\ d2(Z,Y). \end{array}$$

  where $d1/2$, $d2/2$, $c2/2$, $c3/2$, and $c4/2$ are defined as facts that represent an instance of the problem. For this benchmark, we have generated instances of different sizes in a random fashion. More precisely, for each instance, we set the size (number of facts) roughly from 10000 to 10000000 and we randomly divided the number of facts among the previous predicates (around 10-25% for each predicate). Then for each predicate p, we randomly estimated the max value for each term, n and m, in such a way that $n * m = s$, where s is the size of the predicate set of p.
- Reachability problem defined as follows:

$$\begin{array}{c} reach(X,Y) \leftarrow edge(X,Y). \\ reach(X,Y) \leftarrow reach(X,Z),\ edge(Z,Y). \end{array}$$

  Instances of this problem are directed graphs that we have generated varying the number of nodes and the density of the edges. In particular, we considered graphs with a number of nodes from 100 to 2000 and density 20, 40, 60, 80, and 100%.

Among programs with negation, instead, we considered three hard benchmarks from asp competitions that are *Knight Tour with Holes*, *Stable Marriage*, and *Graph Colouring* [10].

Our approach, labeled WF-COMP, was compared with the following tools:

- General-purpose systems that can evaluate Datalog programs: IDLV [13] and GRINGO [14]
- The soufflé framework [4] for which we have rewritten the encoding to produce a suitable encoding for this framework. In particular, two versions have been considered, the interpreted, SOUFFLE, and compiled, SOUFFLE-COMP, ones.
- Compilation-based approach for stratified normal programs WASP-LAZY [7].

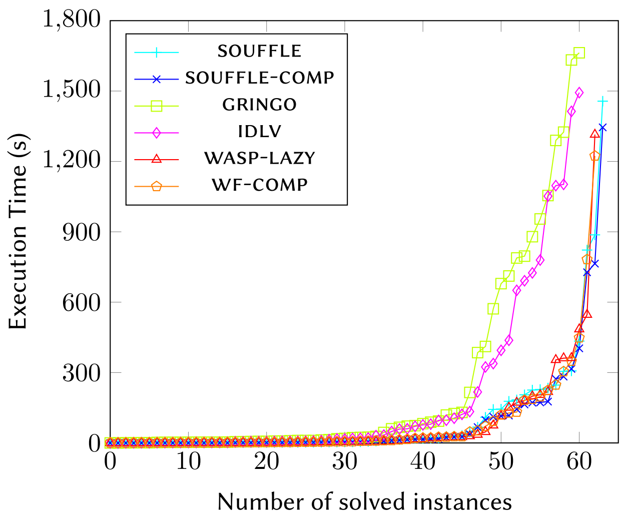

Figure 1: Systems comparison on large-join domain

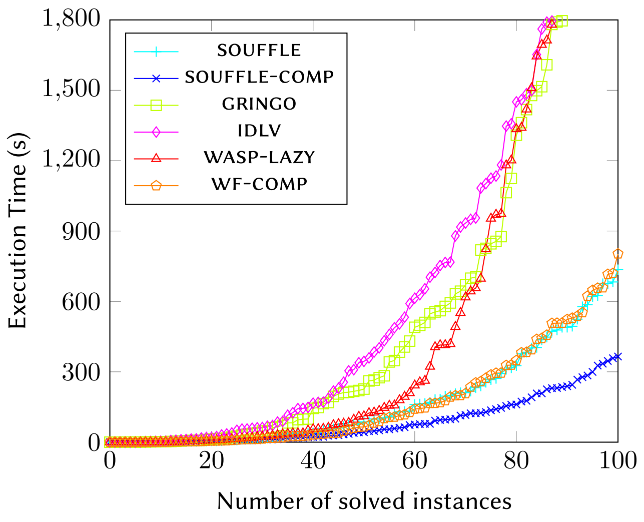

Figure 2: Systems comparison on reachability domain

- ASP system DLV2 [12] that can evaluate normal programs under well-founded semantics.

All the experiments were executed on a machine equipped with Xeon(R) Gold 5118 CPUs, running Ubuntu Linux (kernel 5.4.0-77-generic). Time and memory were limited to 1800 seconds and 8GB, respectively. Source code and benchmark suite are available at https://osf.io/g9n2z/?view_only=aeb0777c469e46499247284b56dfb598

Evaluation on Positive Programs. In this comparison, we run the systems on instances of large-join and reachability. Obtained results are summarized by the cactus plots in Figures 1-2.

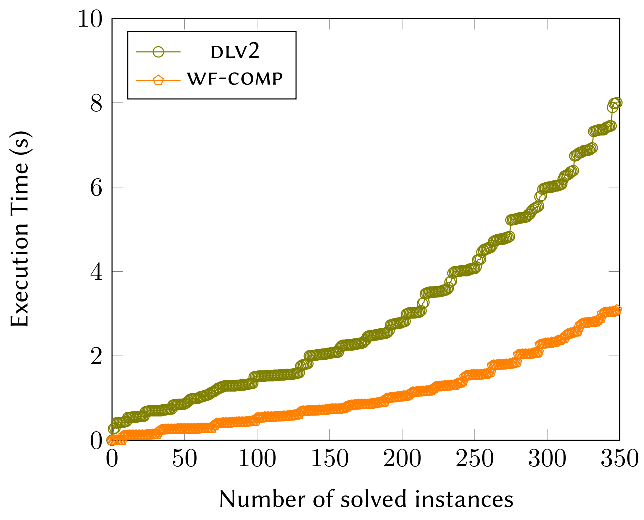

Figure 3: Systems comparison on Knight Tour With Holes benchmark

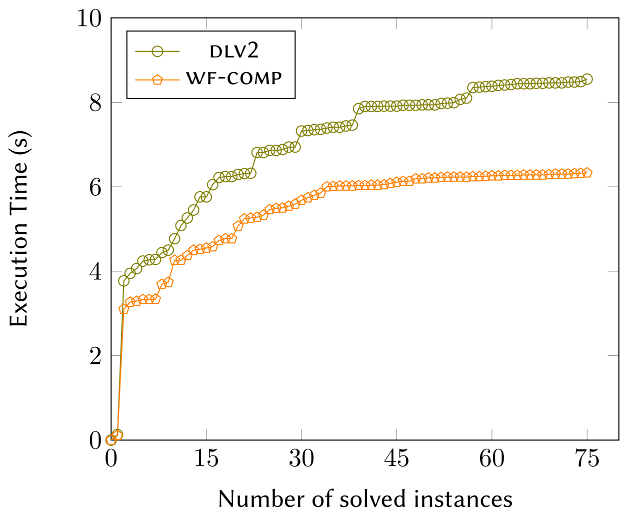

Figure 4: Systems comparison on Stable Marriage benchmark

Recall that a cactus plot reports a line for each system and each line contains a point (X, Y) if a given system is able to solve X within a time limit of Y seconds.

In both cases, WF-COMP outperforms state-of-the-art ASP systems GRINGO and IDLV solving more instances (roughly 15 for reachability and 2 for large-join) and in less time overall. Our approach also outperforms WASP-LAZY on reachability problem solving 16 more instances than the latter, while WF-COMP is comparable on the large-join domain with WASP-LAZY. The best method in this comparison is SOUFFLE system. The difference with our tool (SOUFFLE is preferable in complete graphs) is due to different data structures and also to the different input formats. Indeed, SOUFFLE takes as input a numeric format where input facts are organized in files for

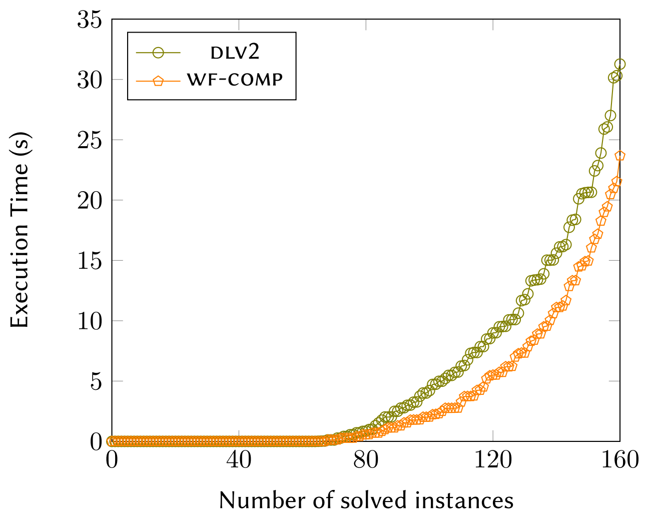

Figure 5: Systems comparison on Graph Colouring benchmark

each input predicate while WF-COMP reads plain text files.

Evaluation on Programs with Negation. In the case of programs with negation, we compare WF-COMP and DLV2 (the other methods do not support unrestricted negation). For each considered benchmark we report a cactus plot see Figures 3, 4, and 5. Obtained results highlight the strength of the proposed approach that outperforms DLV2 on considered benchmarks. In particular, both systems are able to solve all problem instances within time and memory limits, but WF-COMP significantly reduced the total execution time for each benchmark (33.24% on Graph Colouring, 63.26% on Knight Tour With Holes, and 22.37% on Stable Marriage).

5. Conclusion

Logic programming is a widely employed programming paradigm for modeling complex problems in a declarative fashion. Efficient implementations are needed in order to exploit the strength of such formalism in real-world applications. Compilation-based techniques were revealed to be effective in tackling different issues raised in the evaluation of logic programs [4, 7, 6, 8]. In this paper, we proposed a compilation-based approach for logic programs under well-founded semantics. Obtained results demonstrate the effectiveness of the proposed approach both for positive programs and programs with negation. The improvements are significant on programs with negation while in the evaluation of positive programs our implementation is competitive with existing systems. As future works, we planned to extend our technique also to the class of disjunctive programs and to programs with aggregates (again under the well-founded semantics). Also, there is space for improvements in the data structures used for evaluating Datalog programs, where better performance could possibly be achieved (especially on dense graphs) by employing more efficient indexing structures.

References

[1] J. W. Lloyd, Foundations of Logic Programming, 2nd Edition, Springer, 1987.

[2] C. Baral, M. Gelfond, Logic programming and knowledge representation, The Journal of Logic Programming 19 (1994) 73–148.

[3] E. Dantsin, T. Eiter, G. Gottlob, A. Voronkov, Complexity and expressive power of logic programming, ACM Comput. Surv. 33 (2001) 374–425.

[4] H. Jordan, B. Scholz, P. Subotic, Soufflé: On synthesis of program analyzers, in: S. Chaudhuri, A. Farzan (Eds.), Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II, volume 9780 of *Lecture Notes in Computer Science*, Springer, 2016, pp. 422–430. URL: https://doi.org/10.1007/978-3-319-41540-6_23.

[5] B. Cuteri, C. Dodaro, F. Ricca, P. Schüller, Partial compilation of ASP programs, Theory Pract. Log. Program. 19 (2019) 857–873. URL: https://doi.org/10.1017/S1471068419000231.

[6] B. Cuteri, C. Dodaro, F. Ricca, P. Schüller, Overcoming the grounding bottleneck due to constraints in ASP solving: Constraints become propagators, in: C. Bessiere (Ed.), Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020, ijcai.org, 2020, pp. 1688–1694. URL: https://doi.org/10.24963/ijcai.2020/234.

[7] B. Cuteri, F. Ricca, A compiler for stratified datalog programs: preliminary results, in: S. Flesca, S. Greco, E. Masciari, D. Saccà (Eds.), Proceedings of the 25th Italian Symposium on Advanced Database Systems, Squillace Lido (Catanzaro), Italy, June 25-29, 2017, volume 2037 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2017, p. 158. URL: http://ceur-ws.org/Vol-2037/paper_23.pdf.

[8] G. Mazzotta, F. Ricca, C. Dodaro, Compilation of aggregates in ASP systems, in: AAAI, AAAI Press, 2022, pp. 5834–5841.

[9] A. Van Gelder, K. A. Ross, J. S. Schlipf, The well-founded semantics for general logic programs, J. ACM 38 (1991) 620–650. URL: https://doi.org/10.1145/116825.116838.

[10] M. Gebser, M. Maratea, F. Ricca, The sixth answer set programming competition, Journal of Artificial Intelligence Research 60 (2017) 41–95.

[11] S. Liang, P. Fodor, H. Wan, M. Kifer, Openrulebench: an analysis of the performance of rule engines, in: J. Quemada, G. León, Y. S. Maarek, W. Nejdl (Eds.), Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009, ACM, 2009, pp. 601–610. URL: https://doi.org/10.1145/1526709.1526790.

[12] M. Alviano, F. Calimeri, C. Dodaro, D. Fuscà, N. Leone, S. Perri, F. Ricca, P. Veltri, J. Zangari, The ASP system DLV2, in: Logic Programming and Nonmonotonic Reasoning - 14th International Conference, LPNMR 2017, Espoo, Finland, July 3-6, 2017, Proceedings, volume 10377 of *Lecture Notes in Computer Science*, 2017, pp. 215–221. URL: https://doi.org/10.1007/978-3-319-61660-5_19.

[13] F. Calimeri, D. Fuscà, S. Perri, J. Zangari, I-DLV: the new intelligent grounder of DLV, Intelligenza Artificiale 11 (2017) 5–20. URL: https://doi.org/10.3233/IA-170104.

[14] M. Gebser, R. Kaminski, A. König, T. Schaub, Advances in *gringo* series 3, in: J. P. Delgrande, W. Faber (Eds.), Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings, volume 6645 of *Lecture Notes in Computer Science*, 2011, pp. 345–351.