Appears in T. Eiter et al. (Eds.): LPNMR 2001, LNAI 2173

Omega-Restricted Logic Programs

Tommi Syrjänen

Helsinki University of Technology, Dept. of Computer Science and Eng.,
Laboratory for Theoretical Computer Science,
P.O.Box 5400, FIN-02015 HUT, Finland
Tommi.Syrjanen@hut.fi

Abstract. We define a new syntactic class of logic programs, omega-restricted programs. We divide the predicate symbols of a logic program into two parts: domain and non-domain predicates, where the domain predicates are defined by the maximal stratifiable subset of the rules of the program. We extend the usual definition of stratification by adding a special omega-stratum that holds all unstratifiable predicates of the program. We demand that all variables that occur in a rule also occur in the rule body in a positive literal that is on a lower stratum than rule head. This restriction is syntactic and can be checked efficiently. The existence of a stable model of an omega-restricted program is decidable even when function symbols are allowed. We prove that the problem is 2-**NEXP**-complete and identify subclasses of omega-restricted programs such that the problem stays in **NEXP** or **NP**. The class of omega-restricted programs is implemented in the SMODELS system.

1 Introduction

The answer set programming (ASP) paradigm has gained popularity in the recent years as a number of ASP systems have become available (for example, DeReS [3], `dlv` [6], and SMODELS [12]). The basic idea of ASP is to encode a problem as a logic program such that the answer sets (stable models) of the program correspond to the solutions of the problem. We then use a logic program engine to find the answer sets of the program. The underlying formal semantics is usually based on some extension of the stable model semantics of normal logic programs [7].

The inference engines of the existing systems work with ground programs, that is, programs without variables. A rule with variables represents the set of ground rules that can be created by replacing the variables in it by constant terms that occur in the program. This instantiation is done in a preprocessing step before the actual inference engine is used. This bottom-up approach to variable use has prevented the use of function symbols since even one function symbol in a program forces its Herbrand instantiation to be infinite. However, in most cases it is enough to examine only a small subset of the Herbrand instantiation since vast majority of the rules will have unsatisfiable bodies so they can be left out without affecting the set of stable models. This holds true even when function symbols are allowed; it is possible that all answer sets of a program are finite and computable even if the Herbrand instantiation is infinite.

The aim of this work is to define a new class of logic programs, ω-restricted programs, that are syntactically guaranteed to be decidable even when function symbols are used. The basic idea is to construct a hierarchy of predicates such that the predicates on the lowest level are defined using only ground facts and all variables that occur in a rule of level $n+1$ have to also occur in a positive literal of level n or lower in the rule body. The definition of the hierarchy extends the usual concept of stratification [1] by adding a special ω-stratum to hold the unstratifiable part of a program.

It turns out that this syntactic restriction is strong enough to guarantee finite answer sets. In fact, we will see that deciding whether an ω-restricted program has an answer set is 2-**NEXP**-complete. Since the stable model semantics of logic programs without functions is **NEXP**-complete [4], we can conclude that by using ω-restricted functions we move up one step in the exponential hierarchy. This result also implies that we cannot solve all computable problems using ω-restricted programs. Recently P. Bonatti [2] has proposed a computationally complete class of logic programs called finitary programs. However, together with Turing equivalence comes semi-decidability of general reasoning problems.

The ω-restricted programs have been implemented in the SMODELS system [12] that has been designed in Helsinki University of Technology. The SMODELS system is available at http://www.tcs.hut.fi/Software/smodels.

In the following sections we will use the following program to illustrate the basic concepts of ω-restriction:

$$\begin{aligned}
number(0) &\leftarrow ; \cdots ; number(n) \leftarrow \\
odd(x+1) &\leftarrow number(x), even(x) \\
even(x+1) &\leftarrow number(x), odd(x) \\
even(0) &\leftarrow \\
two_divides(x) &\leftarrow even(x) \\
interesting(x) &\leftarrow number(x), \text{not } dull(x) \\
dull(x) &\leftarrow number(x), \text{not } interesting(x) \\
interesting_odd(x) &\leftarrow odd(x), interesting(x) \ .
\end{aligned} \tag{1}$$

2 The Stable Model Semantics

The basic component of a logic program is an *atom* of the form:

$$p(t_1, \ldots, t_n) \tag{2}$$

where p is a n-ary predicate symbol ($n \geq 0$) and t_1, ..., t_n are terms. A *term* is either a variable v, a constant c, or an m-ary function symbol $f(t_1, \ldots, t_m)$ where t_1, ..., t_m are terms. We denote the predicate symbol of an atom A by $pred(A)$. A *literal* is either an atom A or its negation not A.

The set $\mathcal{V}(t)$ of variables that *occur in* a term is defined as follows:

$$\mathcal{V}(t) = \begin{cases} \emptyset & \text{, if } t \text{ is a constant;} \\ \{t\} & \text{, if } t \text{ is a variable; and} \\ \bigcup_{i=1}^{m} \mathcal{V}(t_i) & \text{, if } t \text{ is a function } f(t_1, \dots, t_m) \ . \end{cases} \tag{3}$$

A variable occurs in a literal if it occurs in at least one of its arguments:

$$\mathcal{V}(a(t_1, \dots, t_n)) = \bigcup_{i=1}^{n} \mathcal{V}(t_i) \ . \tag{4}$$

An inference *rule* R is of the form:

$$h \leftarrow l_1, \dots, l_n \ , \tag{5}$$

where the *head* h is an atom and $l_1, \dots, l_n$ in the *body* are literals. The sets of positive and negative literals in the body of R are denoted by $\mathit{body}^+(R)$ and $\mathit{body}^-(R)$ respectively. Intuitively, a rule asserts that if all literals in the rule body are true, then the head must be true also. A *logic program* P is a finite set of rules. We denote the set of predicate symbols that occur in P with $\mathit{preds}(P)$.

The set of variables that occur in a rule R is defined in terms of variables that occur in its literals:

$$\mathcal{V}(R) = \mathcal{V}(\mathit{head}(R)) \cup \bigcup_{l \in \mathit{body}(R)} \mathcal{V}(l) \ . \tag{6}$$

A rule is *ground* if $\mathcal{V}(R) = \emptyset$.

Let P be a variable-free logic program and M be a set of atoms that occur in P. Then, the Gelfond-Lifschitz reduct P^M is obtained by:

1. removing each rule with a negative literal not A in its body where $A \in M$.
2. removing all negative literals form the bodies of the remaining rules.

Since P^M is negation-free, it has a unique least model M'. If the model M' coincides with M, then M is a *stable model* of P.

The *Herbrand universe* $\mathbf{HU}(P)$ of a logic program P is the set of constant terms that can be formed using the constants and function symbols of P. A *ground instance* of a literal or a rule can be obtained by replacing variables in it by terms in $\mathbf{HU}(P)$. The *Herbrand instantiation* P_G is the set of all possible instantiations of rules in P. The set of stable models of a logic program P with variables is defined to be the set of stable models of its instantiation P_G. In practice, we usually do not have to construct the full Herbrand instantiation to be able to construct all stable models. Hereafter we will use the term *instantiation* of P to mean any subset of P_G that has the same set of stable models.

Example 1. Let P be the program:

$$\begin{aligned} &a(1) \leftarrow ; \qquad a(2) \leftarrow \\ &b(x) \leftarrow a(x), \text{not } c(x) \\ &c(x) \leftarrow a(x), \text{not } b(x) \ . \end{aligned} \tag{7}$$

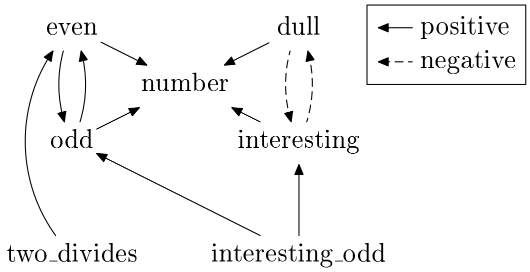

Fig. 1. The dependency graph of Program (1).

Then, the instantiation P_G is:

$$\begin{array}{rr} a(1) \leftarrow & a(2) \leftarrow \\ b(1) \leftarrow a(1), \text{not } c(1) & b(2) \leftarrow a(2), \text{not } c(2) \\ c(1) \leftarrow a(1), \text{not } b(1) & c(2) \leftarrow a(2), \text{not } b(2) \end{array} . \tag{8}$$

Now, P_G has four stable models: $M_1 = \{a(1), a(2), b(1), b(2)\}$, $M_2 = \{a(1), a(2), b(1), c(2)\}$, $M_3 = \{a(1), a(2), c(1), b(2)\}$, and $M_4 = \{a(1), a(2), c(1), c(2)\}$. Consider M_1. The reduct $P_G^{M_1}$ is the program:

$$b(1) \leftarrow; \quad b(2) \leftarrow; \quad a(1) \leftarrow; \quad a(2) \leftarrow \quad . \tag{9}$$

The least model of $P_G^{M_1} = \{a(1), a(2), b(1), b(2)\} = M_1$, so we see that M_1 is really a stable model of P_G and hence of P.

A *splitting set* of a normal logic program P is a set of ground atoms U such that for all rules R in P_G if $head(R)$ is in U, then all atoms occurring in the rule body are also in U. We denote the set of ground rules whose heads are in U with $b_U(P_G)$. Given an an evaluation I of atoms in U, we denote by $e_U(P_G, I)$ the set of ground rules that is obtained by removing from P_G all rules that have a literal l containing an atom of U that is not satisfied by I and removing all literals l containing a member of U from the bodies of the remaining rules. By Splitting Set Theorem [8], M is a stable model of P_G only if $M = I \cup J$ where I is a stable model of $b_U(P_G)$ and J is a stable model of $e_U(P_G \setminus b_U(P), I)$.

3 Omega-Restricted Logic Programs

In this section we give a formal definition for ω-restricted programs. The main idea is to construct a stratification of the predicate symbols such that a predicate p is on a higher level than a predicate q if p is defined in terms of q. We start by formalizing the concept of dependency between predicate symbols.

Definition 1. *Let P be a logic program. Then, the* one-step dependency relation *$D_1(P) \subseteq preds(P) \times preds(P)$ is defined as follows:*

1. $D_1^+(P) = \{\langle pred(a), pred(l)\rangle \mid \exists R \in P : a = head(R) \wedge l \in body^+(R)\}$
2. $D_1^-(P) = \{\langle pred(a), pred(l)\rangle \mid \exists R \in P : a = head(R) \wedge l \in body^-(R)\}$
3. $D_1(P) = D_1^+(P) \cup D_1^-(P)$.

The one-step dependency relation may be drawn as a graph. For example, the dependency graph of Program (1) is shown in Figure 1.

We now generalize the one-step dependency relation to a full dependency relation. The intuition is that a predicate p depends on a predicate q if there is a path from p to q in the dependency graph. If at least one of the edges between p and q is negative, then p depends negatively on q.

Definition 2. *A* dependency path π_P *of a logic program P is a sequence*

$$\pi_P = \langle p_1, p_2, \ldots, p_n\rangle \tag{10}$$

where $p_i \in preds(P)$ for $1 \leq i \leq n$ and $\langle p_j, p_{j+1}\rangle \in D_1(P)$ for $1 \leq j < n$. A path π_P is negative *(denoted by $\overline{\pi}_P$) if and only if $\langle p_j, p_{j+1}\rangle \in D_1^-(P)$ for some $1 \leq j < n$.*

Definition 3. *The* dependency relation $D(P) \subseteq preds(P) \times preds(P)$ *of a logic program P is defined as follows:*

1. *$D^+(P) = \{\langle p, q\rangle \mid \exists \pi_P : \pi_P = \langle p, \ldots, q\rangle\}$;*
2. *$D^-(P) = \{\langle p, q\rangle \mid \exists \overline{\pi}_P : \overline{\pi}_P = \langle p, \ldots, q\rangle\}$; and*
3. *$D(P) = D^+(P) \cup D^-(P)$.*

Next, we define the concept of ω-stratification. The definition extends the traditional definition of stratification [1] by adding a new stratum, the ω-stratum, for the predicates that depend negatively on each other.

Definition 4. *An* ω-stratification *of a program P is a function $\mathcal{S} : preds(P) \to \mathbb{N} \cup \{\omega\}$ such that:*

1. *$\forall p_1 \forall p_2 (\langle p_1, p_2\rangle \in D^+(P) \Rightarrow \mathcal{S}(p_1) \geq \mathcal{S}(p_2))$; and*
2. *$\forall p_1 \forall p_2 (\langle p_1, p_2\rangle \in D^-(P) \Rightarrow \mathcal{S}(p_1) > \mathcal{S}(p_2) \vee \mathcal{S}(p_1) = \omega)$.*

We use the convention that $\omega > n$ for all $n \in \mathbb{N}$. The first condition asserts that a predicate p_1 that depends positively on a predicate p_2 has to be on at least as high stratum as p_2. The second condition states that if p_1 depends negatively on p_2, then p_1 has to be on a higher stratum or they both must be in the ω-stratum. In practice, we are interested in stratifications that are strict in the sense that $\mathcal{S}(p_1) > \mathcal{S}(p_2)$ whenever p_1 depends on p_2 but not vice versa.

Example 2. Consider Program (1). We can construct an ω-stratification $\mathcal{S}$ for it by looking at its dependency graph. As there are no edges leading from *number*, we can set $\mathcal{S}(number) = 0$. Predicates *even* and *odd* depend on *number* and each other, so we set $\mathcal{S}(even) = \mathcal{S}(odd) = 1$. Continuing this, *two_divides* depends on *even* so $\mathcal{S}(two\_divides) = 2$. The negative cycle of *interesting* and *odd* forces that $\mathcal{S}(interesting) = \mathcal{S}(odd) = \mathcal{S}(interesting\_odd) = \omega$. This stratification is shown in Figure 2.

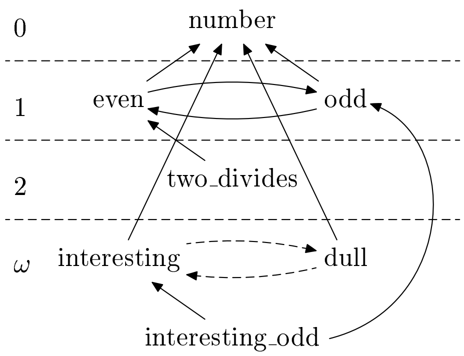

Fig. 2. A stratification of Program (1)

Next, we will extend the ω-stratification to cover also rules and variables by defining the concept of an ω-valuation.

Definition 5. *The* ω-valuation *of a rule R under an ω-stratification $\mathcal{S}$ is the function:*

$$\Omega(R, \mathcal{S}) = \mathcal{S}(pred(head(R)))$$

The ω-valuation *of a variable v in a rule R under an ω-stratification $\mathcal{S}$ is the function:*

$$\Omega(v, R, \mathcal{S}) = \min(\{\mathcal{S}(pred(a)) \mid a \in body^{+}(R) \wedge v \in \mathcal{V}(a)\} \cup \{\omega\})$$

Example 3. Let $\mathcal{S}$ be as defined in Example 2. Consider the rule

$$R: \quad interesting(x) \leftarrow number(x), \text{not } dull(x) \ .$$

Now

$$\begin{aligned} \Omega(R, \mathcal{S}) = &\ \mathcal{S}(interesting) = \omega \\ \Omega(x, R, \mathcal{S}) = &\ \min\{\mathcal{S}(number), \mathcal{S}(dull), \omega\} = \min\{0, \omega\} = 0 \ . \end{aligned}$$

A rule is ω-restricted if all variables that occur in it also occur in a positive body literal that belongs to a strictly lower stratum than the head.

Definition 6. *Let R be a rule in a logic program P. Then R is* ω-restricted *if and only if there exists an ω-stratification $\mathcal{S}$ such that:*

$$\forall v \in \mathcal{V}(R) : \Omega(v, R, \mathcal{S}) < \Omega(R, \mathcal{S}) \ .$$

Definition 7. *Let P be an logic program. Then P is* ω-restricted *if and only if all rules $R \in P$ are ω-restricted.*

Example 4. Consider the rule:

$$s(x+1) \leftarrow s(x)$$

This rule is not ω-restricted since for all ω-stratifications, $\Omega(R, \mathcal{S}) = \Omega(x, R, \mathcal{S})$.

Finally, we divide the predicate symbols into two classes, domain predicates that are on finite strata and non-domain predicates that are on the ω-stratum. The domain predicates are defined by the maximal stratifiable subset of the rules of the program.

Definition 8. *Let P be an ω-restricted program. Then a predicate $p \in preds(P)$ is a* domain predicate *if and only if there exists an ω-stratification $\mathcal{S}$ such that $\mathcal{S}(p) < \omega$. The set of rules defining domain predicates of P is denoted by $\mathcal{D}(P)$.*

4 Domain Predicates and Instantiation

The subprogram $\mathcal{D}(P)$ defining the domain predicates of P is stratified, so it has a unique least model $M_{\mathcal{D}(P)}$. It is easy to verify that $M_{\mathcal{D}(P)}$ is a splitting set of P. Thus, we can compute the stable models of P by first computing $M_{\mathcal{D}(P)}$ and then extending it to cover the atoms on ω-stratum. Moreover, each variable that occurs in a rule R occurs also in a positive domain literal so we can create all relevant ground instances of R by computing the natural join of extensions of the domain literals in $body(R)$.

We can compute $M_{\mathcal{D}(P)}$ and the instantiation P_{NG} of $P_N = P \setminus \mathcal{D}(P)$ using the following algorithm:

1. Find all strongly connected components of the dependency graph of P. Each component becomes a new stratum with the exception that all components that have a path to a negative dependency cycle are put on the ω-stratum. Order the different strata by doing a depth-first search over the strongly connected components.
2. Instantiate the predicates on finite strata starting from the lowest one. After instantiation, compute the deductive closure of the new ground rules and store the resulting atoms as facts in a database. These facts are then used to give domains for variables when we instantiate the rules on higher strata.
3. Finally, instantiate all rules on the ω-stratum and output them along with the domain facts.

5 Computational Complexity

In this section we examine the computational complexity of ω-restricted programs. We are interested in two problems:

- In INSTANTIATION we have an ω-restricted program P and a ground atom $p(t_1, \ldots, t_n)$ and we want to find whether one of the following conditions holds:

Table 1. Computational complexity

		INSTANTIATION	MODEL
No variables		—	**NP**-complete
Fixed variables	No functions	**P**-complete	**NP**-complete
	With functions	**EXP**-complete	**NEXP**-complete
Unlimited variables	No functions	**EXP**-complete	**NEXP**-complete
	With functions	2-**EXP**-complete	2-**NEXP**-complete

1. $p(t_1, \ldots, t_n) \in M_{\mathcal{D}(P)}$; or
2. There is a rule $p(t_1, \ldots, t_n) \leftarrow l_1, \ldots, l_n$ in P_{NG}.

– In MODEL we want to find out whether an ω-restricted program P has an answer set.

The instantiation complexity is included in the model complexity in all cases since we may have to construct the full instantiation of a program before we know whether it has any stable models at all.

In addition to proving complexity results for the whole class of ω-restricted programs, we examine how the computational complexities of INSTANTIATION and MODEL change when we restrict our attention to some subclasses of programs. We use two parameters to divide the ω-restricted programs into four classes:

– The maximum number of variables in a rule is either fixed to some constant d or it is unlimited; and
– Function symbols are either allowed or not.

The main complexity results are presented in Table 1. The MODEL complexities of function-free normal logic programs with the stable model semantics have been presented in earlier literature [10, 4]. The corresponding complexity classes of function-free ω-restricted programs are the same so we see that at least in these categories ω-restricted programs are as expressive as normal logic programs. Since the MODEL problem of the unrestricted case is 2-**NEXP**-complete, we know that ω-restricted programs are decidable:

Theorem 1. *Both* INSTANTIATION *and* MODEL *are decidable for ω-restricted programs.*

5.1 Turing Machine Translation

Most of the complexity results of this work are derived by proving that the computations of a deterministic Turing machine M can be simulated by a logic program P such that the size of P is polynomial with respect to the size of M.

Definition 9. *A deterministic* Turing machine $M = (K, \Sigma, \delta, s)$ *where K is a finite set of states, Σ is a finite alphabet containing the blank symbol $\sqcup$, $s \in K$ is the initial state and δ is a transition function $\delta : K \times \Sigma \to (K \cup \{y, n\}) \times \Sigma \times \{-1, 0, 1\}$.*

A computation of a Turing machine M given an input x starts from the configuration $(s, \underline{\sqcup} x)$ and each computation step yields a new configuration according to δ until one of the halting states y (*accept*) or n (*reject*) is reached.

We encode the states of a Turing machine M using the predicate $\mathit{state}(q)$, the alphabet using $\mathit{symbol}(\sigma)$, and the transitions using $\mathit{transition}(q_1, \sigma_1, q_2, \sigma_2, d)$, where $d \in \{-1, 0, +1\}$. The atom $\mathit{at\text{-}place}(\sigma, p, t)$ is used to denote that the input tape cell p contains the symbol σ at the time step t. The predicate $\mathit{current\text{-}state}(q, p, \sigma, t)$ indicates that the machine is in the state q and the head is over the tape cell p looking at the symbol σ at the time step t.

We encode one computation step using the two rules:

$$\begin{aligned} \mathit{at\text{-}place}(s_2, p, t+1) \leftarrow\ & \mathit{transition}(q_1, s_1, q_2, s_2, d), \\ & \mathit{current\text{-}state}(q_1, p, s_1, t), \\ & \mathit{place}(p), \mathit{time}(t) \end{aligned} \tag{11}$$

$$\begin{aligned} \mathit{current\text{-}state}(q_2, p+d, s_3, t+1) \leftarrow\ & \mathit{transition}(q_1, s_1, q_2, s_2, d), \\ & \mathit{current\text{-}state}(q_1, p, s_1, t), \\ & \mathit{at\text{-}place}(s_3, p+d, t), \mathit{time}(t), \\ & \mathit{place}(p), \mathit{symbol}(s_3)\ . \end{aligned} \tag{12}$$

Here we have used the notation $t+1$ to denote the successor of t. How the successor relation is actually defined depends on the program class that we want to examine. The same thing holds also for predecessor relation that is used in the case of $p-1$.

The rules above handle the cell where the read/write-head is currently positioned. In addition, we have to assert that the state of the other tape cells stays constant:

$$\begin{aligned} \mathit{at\text{-}place}(s_1, p_1, t+1) \leftarrow\ & \mathit{current\text{-}state}(q, p_2, s_2, t), \\ & \mathit{at\text{-}place}(s_1, p_1, t), \mathit{time}(t), \\ & \mathit{symbol}(s_1), \mathit{symbol}(s_2), \mathit{state}(q), \\ & \mathit{place}(p_1), \mathit{place}(p_2), \mathrm{not}\ \mathit{equal}(p_1, p_2)\ . \end{aligned} \tag{13}$$

In the initial configuration all tape cells that are not part of the input are empty:

$$\mathit{at\text{-}place}(\sqcup, p, 1) \leftarrow \mathit{place}(p), \mathrm{not}\ \mathit{part\text{-}of\text{-}input}(p)\ . \tag{14}$$

The first $|x|$ tape cells are initialized from the input and they also belong to the extension of $\mathit{part\text{-}of\text{-}input}/1$. Finally, we want to recognize whether the Turing machine halts in an accepting state or not:

$$\begin{aligned} \mathit{accept} &\leftarrow \mathit{current\text{-}state}(y, p, s, t), \mathit{place}(p), \mathit{symbol}(s), \mathit{time}(t) \\ \mathit{reject} &\leftarrow \mathit{current\text{-}state}(n, p, s, t), \mathit{place}(p), \mathit{symbol}(s), \mathit{time}(t)\ . \end{aligned} \tag{15}$$

Note that we have not yet given definitions for the predicates $\mathit{time}/1$ and $\mathit{place}/1$ that encode the time steps and tape cells. In the following complexity proofs we show how we can define them in a polynomial number of rules using tools that are available for the four different ω-restricted program classes.

Since all rules except (13) and (14) in the translation are negation-free and both negations are over a predicate with a fixed extension that is linear to the size of the input program, the least model of the instantiation can be found in a linear time with respect to its size [5]. All predicates are domain predicates and we can easily find whether *accept* or *reject* is true in $M_{\mathcal{D}(P)}$.

We can generalize the translation to allow non-deterministic Turing machines by forcing the machine to choose between possible transitions at all computation steps. Due to space constraints, we do not include the details here but one possible translation has been presented by V. W. Marek and J. B. Remmel [9]. The existence of such a translation is enough to prove the following lemma:

Lemma 1. *If the* INSTANTIATION *problem of a subclass of the ω-restricted programs is* **C***-complete for some complexity class* **C***, the corresponding* MODEL *problem is* **NC***-complete.*

5.2 Complexity Results

Theorem 2. *The* INSTANTIATION *of an ω-restricted program is* **P***-complete when the number d of variables occurring in it is fixed ($d \geq 3$) and no function symbols are allowed.*

Proof. We construct the proof in two parts:

(a) *Inclusion.* Let P be a program with d distinct variables. Then, each rule has at most n^d ground instances, where n is the number of constants in the program.

(b) *Hardness.* The **P**-complete problem BOOLEAN CIRCUIT VALUE [11] can be expressed as an ω-restricted logic program as follows:

$$\begin{aligned} true(G) &\leftarrow nand\text{-}gate(G, L, R), false(L) \\ true(G) &\leftarrow nand\text{-}gate(G, L, R), false(R) \\ false(G) &\leftarrow nand\text{-}gate(G, L, R), true(L), true(R)\ . \end{aligned} \tag{16}$$

Here we suppose that the Boolean circuit is implemented using only not-and gates and that the truth values of the input gates are given as facts.

Corollary 1. *The* MODEL *problem for a fixed number d of variables and no function symbols is* **NP***-complete, if $d \geq 3$.*

Theorem 3. *The* INSTANTIATION *of an unlimited-variable ω-restricted program is* **EXP***-complete if no function symbols are allowed.*

Proof. For inclusion, see Dantsin *et.al.* [4]. The hardness can be proved by noting that a deterministic **EXP**-time Turing machine M uses at most 2^{n^k} time steps for some k when the length of the input is n. We have to show that we can generate an exponential number of atoms representing time steps and tape cells using a program whose size is polynomial with respect to the size of M. To do this, we need to implement a n^k-bit binary counter that runs from 0 to $2^{n^k} - 1$. This can be done by encoding the numbers as vectors of binary variables:

$$\begin{aligned} \mathit{number}(0, \ldots, 0) &\leftarrow \\ \mathit{number}(y_1, \ldots, y_{n^k}) &\leftarrow \mathit{bit}(y_1), \ldots, \mathit{bit}(y_{n^k}), \\ &\quad\ \mathit{number}(x_1, \ldots, x_{n^k}), \\ &\quad\ \mathit{next}(x_1, \ldots, x_{n^k}, y_1, \ldots, y_{n^k}) \ . \end{aligned} \tag{17}$$

The predicate $\mathit{bit}/1$ is an auxiliary with the extension $\{\mathit{bit}(0), \mathit{bit}(1)\}$ that is used to ensure that the rule is ω-restricted. The successor relation can be encoded with the rule:

$$\begin{aligned} \mathit{next}(x_1, \ldots, x_{n^k}, y_1, \ldots, y_{n^k}) \leftarrow\ & \mathit{add}(x_1, 1, y_1, c_1), \\ & \mathit{add}(x_2, c_1, y_2, c_2), \\ & \quad \vdots \\ & \mathit{add}(x_{n^k}, c_{n^k-1}, y_{n^k}, c_{n^k}) \end{aligned} \tag{18}$$

where $\mathit{add}/4$ is defined using the following four facts:

$$\begin{aligned} &\mathit{add}(0, 1, 1, 0) \leftarrow \qquad \mathit{add}(0, 0, 0, 0) \leftarrow \\ &\mathit{add}(1, 0, 1, 0) \leftarrow \qquad \mathit{add}(1, 1, 0, 1) \leftarrow \ . \end{aligned} \tag{19}$$

We can implement the predecessor function by switching the arguments of the *next* predicate. Now the time steps and tape positions can be defined in terms of numbers:

$$\begin{aligned} \mathit{time}(x_1, \ldots, x_{n^k}) &\leftarrow \mathit{number}(x_1, \ldots, x_{n^k}) \\ \mathit{place}(x_1, \ldots, x_{n^k}) &\leftarrow \mathit{number}(x_1, \ldots, x_{n^k}) \ . \end{aligned} \tag{20}$$

Finally, we replace all references to $\mathit{time}/1$ and $\mathit{place}/1$ by time/n^k and place/n^k and add all necessary domain predicates to the rule bodies.

Theorem 4. *The* INSTANTIATION *of a fixed-variable ω-restricted program that uses function symbols is* **EXP***-complete, if $d \geq 8$.*

Proof.

(a) *Inclusion.* Without a loss of generality we may assume that there are k strata with c rules each in P. Let us use a_n to denote the number of ground instances of rules that belong to the first n strata.

Since the number d of variables is fixed, a rule on the $n+1$-stratum may have at most a_n^d ground instances. Now we can establish an upper bound for the number of ground instances of rules on the stratum $n+1$ or lower:

$$
\begin{aligned}
a_{n+1} &= c \cdot a_n^d + a_n \leq c \cdot a_n^d + c \cdot a_n^d \qquad \text{(when } d \geq 1) \\
&= 2c \cdot a_n^d \\
&= 2^{d^{n-1}+d^{n-2}+\cdots+d+1} \cdot c^{d^n+d^{n-1}+\cdots+d+1} \\
&= 2^{(\log_2 c)d^n+(\log_2 c+1)d^{n-1}\cdots+(\log_2 c+1)}
\end{aligned} \tag{21}
$$

As both c and d are linear with respect to the size of the program, a_n grows $O(2^{n^{2k}})$ so the problem is in **EXP**.

(b) *Hardness.* As in the proof of Theorem 3, we need only to construct a binary counter from 0 up to $2^{n^k}-1$. We do this by encoding an m-bit binary number x as a function $b_1(b_2(\cdots b_m(0)\cdots))$ where b_i is f if the ith bit of x is 0 and t if it is 1. The m-bit binary numbers can be generated recursively from $m-1$-bit numbers by the following two rules:

$$
\begin{aligned}
number_m(t(x)) &\leftarrow number_{m-1}(x) \\
number_m(f(x)) &\leftarrow number_{m-1}(x)
\end{aligned} \tag{22}
$$

Here we need $m+1$ different *number* predicates since otherwise the rules would not be ω-restricted. As the basic basic case of the recursion, we define one 0-bit number as:

$$number_0(0) \leftarrow \ . \tag{23}$$

The successor relation can also be defined recursively:

$$
\begin{aligned}
next_m(t(x),t(y)) &\leftarrow next_{m-1}(x,y) \\
next_m(f(x),f(y)) &\leftarrow next_{m-1}(x,y) \\
next_m(f(x),t(y)) &\leftarrow last_{m-1}(x), first_{m-1}(y)
\end{aligned} \tag{24}
$$

where $last_m/1$ and $first_m/1$ are defined as:

$$
\begin{aligned}
last_m(t^m(0)) &\leftarrow \\
first_m(f^m(0)) &\leftarrow \ .
\end{aligned} \tag{25}
$$

The translation uses $7n^k+3$ rules to create all n^k-bit numbers so we now have a polynomial reduction from **EXP**-time Turing machines to ω-restricted programs using only function symbols and the proof is completed.

Corollary 2. *The* MODEL *problem of a fixed-variable ω-restricted program that uses function symbols is* **NEXP***-complete, if $d \geq 8$.*

Theorem 5. *The* INSTANTIATION *of an ω-restricted program is* 2-**EXP***-complete.*

Proof. We can combine the proofs of Theorems 3 and 4 to see that the problem is in 2-**EXP** and that it is possible to implement all 2^{n^k}-bit integers putting together the two different exponential constructions.

Corollary 3. *The* MODEL *problem of an ω-restricted program is* 2-**NEXP***-complete.*

6 Conclusions

We defined a new class of logic programs, ω-restricted programs, that are decidable even when function symbols are used. We showed that the computational complexity of the program class is 2-**NEXP**-complete. If we make further restrictions either by fixing the maximum number of variables that may occur in a rule or by disallowing the function symbols, the complexity drops to **NEXP**-complete. If both restrictions are in effect, the complexity stays **NP**-complete. We have implemented the ω-restricted programs in the SMODELS system.

Acknowledgements

The author is grateful for the anonymous referees for their comments. The financial support of HeCSE graduate school, Academy of Finland (project no. 43963), and Tekniikan edistämissäätiö are gratefully acknowledged.

References

1. K.R. Apt and R. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19–20:9–71, 1994.
2. Piero A. Bonatti. Resoning with infinite stable models. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, August 2001.
3. Paweł Cholewiński, Victor W. Marek, and Miroslaw Truszczyński. Default reasoning system DeReS. In Luigia Carlucci Aiello, Jon Doyle, and Stuart Shapiro, editors, *KR'96: Principles of Knowledge Representation and Reasoning*, pages 518–528. Morgan Kaufmann, San Francisco, California, 1996.
4. Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. In *Proceedings of the Twelfth Annual IEEE Conference on Computational Complexity*, pages 82–101, Ulm, Germany, June 1997. IEEE Computer Society Press.
5. W.F. Dowling and J.H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 3:267–284, 1984.
6. Eiter, T., Leone, N., Pfeifer G., Mateis C., and Scarcello, F. The kr system `dlv`: Progress report, comparisons and benchmarks. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 406–417. Morgan Kaufmann Publishers, 1998.
7. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. of the 5th ICLP*, pages 1070–1080. The MIT Press, 1988.
8. Vladimir Lifschitz and Hudson Turner. Splitting a logic program. In *Proceedings of the Eleventh International Conference on Logic Programming*, pages 23–37, 1994.
9. V. W. Marek and J. B. Remmel. On the foundations of answer set programming. In *Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, pages 124–131. AAAI Press, March 2001.
10. W. Marek and M. Truszczyński. Autoepistemic logic. *Journal of the Association for Computing Machinery*, 38:588–619, 1991.
11. Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley Publishing Company, Inc, 1994.
12. Tommi Syrjänen and Ilkka Niemelä. The smodels system. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*, Vienna, Austria, September 2001. Springer-Verlag.