Modular Logic Programming and Generalized Quantifiers

Thomas Eiter, Georg Gottlob, Helmut Veith

Christian Doppler Labor für Expertensyteme
Technische Universität Wien
Paniglgasse 16, 1040 Wien

Janurary 11, 1997

CD-TR 97/108

Modular Logic Programming and Generalized Quantifiers

(Preliminary Report)

Thomas Eiter[+] **Georg Gottlob**[*] **Helmut Veith**[*]

[+] AG Informatik, Mathematical Institute, University of Giessen
Arndtstraße 2, D-35392 Gießen, Germany.
Email: `Thomas.Eiter@informatik.uni-giessen.de`

[*] Information Systems Department, TU Vienna
Paniglgasse 16, A-1040 Wien, Austria.
Email: `(gottlob|veith)@dbai.tuwien.ac.at`

January 13, 1997

Abstract

The research on logic programming with modules has followed two mainstreams: Programming-in-the-large, where compositional operators are provided for combining separate and independent modules, and programming-in-the-small, which aims at enhancing logic programming with new logical connectives, in order to provide abstraction and scoping mechanism available in other programming paradigms. Both approaches suffer from certain problems, and a number of important issues in modular logic programming (e.g., negation) remain to be satisfactorily answered.

In this paper, we present a general model theoretic approach which combines programming in-the-large and in-the-small in a satisfactory way. Rather than inventing completely new constructs, however, we resort to a well-known concept in formal logic: generalized quantifiers.

Generalized quantifiers have been conceived as devices for expressing higher order properties which are not first-order definable. For example, it is well-known that transitive closure of a graph is not first-order expressible, which suggests to enrich first-order logic by a generalized quantifier for transitive closure.

We show how generalized quantifiers can be incorporated into logic programs, both for Horn logic programs as well as in the presence of negation. Our basic observation is then that a logic program can be seen as a generalized quantifier, and we obtain a semantics for modular logic programs this way.

The presence of generalized quantifiers in logic programs gives rise to interesting classes of logic programs. We present a taxonomy of natural such classes, and investigate their properties. In particular, their expressive power over finite structures is thoroughly analyzed.

Contents

1 Introduction **1**

2 Preliminaries **3**

2.1 Logic Programs 3

2.2 Generalized Quantifiers 4

3 Logic Programs with Generalized Quantifiers **5**

3.1 Syntax 5

3.2 Semantics 6

3.3 Monotone Generalized Quantifiers 12

4 Modular Logic Programs **14**

4.1 Syntax of modular logic programs 15

4.2 Semantics of modular logic programs 16

4.3 Nested LP modules 17

5 A Taxonomy of Program Classes **18**

5.1 Dependency graph 18

6 Expressiveness of Modular Logic Programs **20**

6.1 Preliminaries on expressive power 21

6.2 Results 22

6.3 Expressive power of MLPs with **LP** modules 22

6.4 Expressive power with monotone modules 25

6.5 **2Pos(*)** programs 27

6.6 Nested modular logic programs 28

7 Further Issues and Conclusion **28**

7.1 Summary 28

7.2 Related Work 29

7.3 Further work 30

1 Introduction

A necessary extension to logic programming for successful use in practice are constructs which allow for structured programming. In particular, the possibility of defining subprograms must be supported, which may be referred to at different places in a program. This requires appropriate scoping and abstraction mechanisms for program modules.

This need has initiated a large number of investigations, which evolved into the area of modular logic programming, e.g. [32, 35, 29, 17, 22, 5]; see [5] for a comprehensive survey.

The research on logic programming with modules has followed two mainstreams [5]. The one is programming-in-the-large, where compositional operators are provided for combining separate and independent modules, e.g. [35, 29, 17]. Here, the approach is to provide operators which allow to "algebraically" combine programs, which does not need for an extension of the theory of logic programs. The other direction is programming-in-the-small, which aims at enhancing logic programming with abstraction and scoping mechanism available in other programming paradigms, e.g. [32, 22]. This approach fosters the introduction of new logical connectives in an extended logical language. Thus, the two mainstreams are quite divergent; nontheless, a uniform reconstruction of them has been attempted in [5].

Both approaches suffer from certain problems, and a number of important issues in modular logic programming remain to be satisfactorily answered (cf. [5]). For example, as pointed out in [5, page 497], the treatment of negation in different languages for modular logic programming is an interesting open issue.

Our goal is to find a general model theoretic approach that combines programming in-the-large and in-the-small in a satisfactory way. Rather than inventing completely new constructs, however, we resort to a well-known concept in formal logic, which appears to be a valuable tool for modular logic programming: generalized quantifiers.

Generalized quantifiers [34, 27] have been conceived as devices for expressing higher order properties which are not first-order definable. Enriching a first-order language by such quantifiers allows to increase the expressive capability of the logic. Informally, the semantics of a generalized quantifier Q is given by a set of structures on which Q evaluates to true. The application of Q to a formula φ is evaluated by testing membership of the structure defined by φ in this set; if it belongs to it, $Q\varphi$ is true, otherwise false (see Section 2 for details). Mostowski [34] gave a formalization of generalized quantifiers, which served as a basis for the seminal work of Lindström [27].

For example, it is well-known that transitive closure of a graph is not first-order expressible, which suggests to enrich first-order logic by a generalized quantifier for transitive closure. Another well-known example of generalized quantifiers are Henkin (or branching) quantifiers [24], which have been studied extensively in logic. A further well-known generalized quantifier is the majority (or Rescher) quantifier, which tells whether more individuals satisfy a formula φ_1 than a formula φ_2. To date, general quantifiers have applications in various areas including linguistics, finite model theory, and databases, cf. [4, 6, 2, 25].

After fixing notation and recalling basic concepts in Section 2, we show in Section 3 how generalized quantifiers can be incorporated into logic programs. We thus establish a clean interface between LP and any other semantically defined predicates.

The key idea of this paper is then presented in Section 4: A logic program can be seen as a generalized quantifier. Therefore, together with the results of the previous sections, we automatically obtain a semantics for modularly nested logic programs: A logic programs P_1 can refer to a module P_2 by using P_2 as a generalized quantifier. By such an approach we obtain a clean and simple semantics for modular logic programs, which enjoys the following advantages:

- It is a semantics from first principles. Except for the concept o generalized quantifier, only well-known concepts from logic programming are used.
- It is conservative, i.e., it properly extends the classical semantics of logic programming.
- It is amenable to various extensions of basic Horn-clause semantics, such as stratified negation or stable negation.
- It facilitates the use of modules employing different semantics. For example, a logic program that is interpreted according to the skeptical variant of stable model semantics may use a module which is interpreted according to the brave variant of stable model semantics.
- It allows for multiple use of the same module in a program, where each call may involve different parameters. Thus, the most important feature of subprograms, which is heavily needed in practice, is fully supported.
- The expressive power of modular LP according to our approach is well understood.
- It supports information hiding. This feature is naturally provided by the scope of a generalized quantifier.

These properties document that our approach meets a number of desiderata which appear to be natural and important for an approach to modular logic programming meeting the needs in practice.

After having introduced the main concept of modular logic programs in Section 4, we present a taxonomy of program classes in Section 5. The classes are motivated by natural syntactical and semantical concepts which arise in the context of modular logic programming. In particular, stratification naturally extends from ordinary logic programs to logic programs with general quantifiers and hence also to modular logic programs. Stratifiedness can be violated by improper negation, but also by improper use of generalized quantifiers. This leads us to the concept of call recursion-freeness. Intuitively, a program is call recursion-free if there is no recursion through generalized quantifiers.

For each of the program classes, the expressive power over finite structures ist studied in Section 6. By means of descriptive complexity theory, we characterize the expressiveness of all discussed program classes through well-known complexity classes. As a byproduct of our proofs, we obtain interesting normal forms for some classes of modular logic programs. For example, we show that every stratified modular logic program whose modules consists of (possible unstratified) LPs under the stable model semantics can be replaced by a single clause of the form

$$A(\mathbf{y}) \leftarrow M_1[\mathbf{Q}](\mathbf{x}, \mathbf{y}), \neg M_2[\mathbf{Q}](\mathbf{x}, \mathbf{y})$$

where the predicate $M_i[\mathbf{Q}]$, $i = 1, 2$, describes the result of the call of module M_i on input of predicates $\mathbf{Q} = Q_1, \ldots, Q_m$.

In Section 7 we conclude the paper with a brief discussion of related approaches and an outlook to further work.

2 Preliminaries

2.1 Logic Programs

We assume that the reader is familiar with the concepts and terminology of logic programming; see [28] for background.

Letters $P, Q, \ldots$ denote predicates, lower case letters x, y, z variables, $a, b, c, \ldots$ constants and $f, g \ldots$ functions. The bold face version $\mathbf{P}$ of a predicate symbol P denotes a list $P_1, \ldots, P_m$ of predicate symbols, and similar for variable, function, and constant symbols.

As usual, a logic program is a finite collection of clauses

$$A \leftarrow B_1, \ldots, B_m \tag{1}$$

where A is an atom and the B_i's are literals in a first-order language. We use $\neg$ for the negation (as failure) operator in logic programming. A program is positive, if no negation occurs in it.

The meaning of a positive logic program P has been defined in terms of its least model; for a general logic program, a number of different semantics have been proposed, since the definition of negation is nontrivial and allows for alternative reasonable interpretations.

From an abstract point of view, we can see a logic programming semantics as a mapping which associates, given a program P, with every suitable structure $\mathcal{A}$ another structure $\mathcal{A}'$, i.e., as a map

$$\mathcal{M}_P : \mathcal{S} \rightarrow \mathcal{S}$$

from the set $\mathcal{S}$ of structures into itself. The structure $\mathcal{A} = (A, \mathbf{f}, \mathbf{c}, \mathbf{R})$ provides the domain of discourse A, an interpretation of functions $\mathbf{f}$ and individual constants $\mathbf{c}$ in P, and an interpretation of some predicates $\mathbf{R}$, which are regarded as input facts to the program. The structure $\mathcal{M}_P(\mathcal{A}) = (A, \mathbf{S})$ has relations of $\mathbf{S}$ over the universe A of $\mathcal{A}$; intuitively, they contain the positive ground atoms from the predicates which are inferred by the semantics $\mathcal{M}$ from P on $\mathcal{A}$.

Notice that for the purpose of this paper, we disregard inference of negative facts (ground literals); this is no shortcoming however, since for positive programs, negative facts can be inferred by the CWA, and for programs with negation, inference of negative facts can be emulated by positive facts and stratified negation in any reasonable semantics $\mathcal{M}$ (cf. below).

Usually, the interest in logic programming is with a fixed domain A and fixed interpretations of constants and functions, namely the Herbrand universe of P and the Herbrand interpretation of functions and constants. Moreover, usually no input relations $\mathbf{R}$ are explicitly provided; we forsee this possibility, however, since we shall consider logic programming modules, which compute output on a given input provided by relations. Technically, the input can be incorporated by adding for each ground atom $R(\mathbf{t})$ such that $\mathcal{A} \models R(\mathbf{t})$ to P the rule $R(\mathbf{t}) \leftarrow$, under the proviso that input predicates only may occur in rule bodies. (An alternative is the modification of the program P applying the Gelfond-Lifschitz reduction.)

While the various semantics $\mathcal{M}$ from the literature differ in general, they coincide on stratified logic programs (see Section 5 for a definition); for this reason, we adopt here the following postulate:

- A semantics $\mathcal{M}$ must extend the semantics of stratified logic programs, i.e., for every stratified logic program P, $\mathcal{M}_P = \mathcal{M}_P^{str}$, where $\mathcal{M}_P^{str}$ is the stratified model semantics [1, 37].

A number of other properties of semantics are desirable, e.g. relevance and modularity; see [10, 8, 9].

2.2 Generalized Quantifiers

The extension of first-oder logic by generalized quantifiers (GQs) was studied in depth in the seminal work by Lindström [27]; particular GQs, e.g. Henkin quantifiers [24], have been considered earlier. Formally, GQs can be defined as follows.

Let C be a class of logical structures over a relational signature

$$\sigma = (R_1, \ldots, R_n)$$

with arities $a_1, \ldots, a_n$, and suppose that C is closed under isomorphism, i.e., if $\mathcal{A} \cong \mathcal{B}$ and $\mathcal{A}$ belongs to C, then also $\mathcal{B}$ belongs to C. Each such class C has an associated GQ Q_C whose intended semantics is to check if a relation defined by the underlying logic belongs to C.[1]

The extension $L(Q_C)$ of a logic L by a GQ Q_C is the closure of L under the following rule:

If $\varphi_1(\mathbf{x}_1), \ldots, \varphi_n(\mathbf{x}_n)$ are formulas of $L(Q_C)$ where every φ_i has at least a_i free variables $\mathbf{x}_i$, then

$$Q_C \mathbf{x}_1 \cdots \mathbf{x}_n [\varphi_1, \ldots, \varphi_n] \tag{2}$$

is a formula of $L(Q_C)$, in which the occurrences of $\mathbf{x}_1, \ldots, \mathbf{x}_n$ are bound. For clarity, we shall often write the list of free variables $\mathbf{y}$ after a formula (2).

The semantics of Q_C is defined as follows. Consider the formula

$$\Theta = Q_C \mathbf{x}_1 \cdots \mathbf{x}_n [\varphi_1, \ldots, \varphi_n](\mathbf{y}),$$

and let $\mathcal{A}$ be a structure and $\mathbf{b}$ a tuple in $\mathcal{A}$ matching the arity of $\mathbf{y}$. Then $(\mathcal{A}, \mathbf{b}) \models \Theta$ if and only if the structure $(A, \varphi_1^{\mathcal{A},\mathbf{b}}, \ldots, \varphi_n^{\mathcal{A},\mathbf{b}})$ belongs to C, where $\varphi_i^{\mathcal{A},\mathbf{b}} = \{\mathbf{a} \mid \mathcal{A} \models \varphi_i[\mathbf{a}, \mathbf{b}]\}$. The following example provides some prominent GQs.

Example 1 *Consider the following GQs:*

1. *The* universal quantifier: $C_\forall = \{ (A, A) \}$.

[1] Closedness under isomorphism assures that terms are treated as completely uninterpreted by the GQ. If only particular interpretations are considered, e.g., Herbrand interpretations, weaker closure conditions are in order; cf. the discussion about genericity in logical database query languages.

2. *The* existential quantifier: $C_\exists = \{ (A,U) \mid \emptyset \neq U \subseteq A\}$.
3. *The* complement quantifier: $C_\sim = \{ (A,U,\{v\}) \mid v \in A - U\}$
4. *The* majority quantifier: $C_M = \{ (A,U,V) \mid U,V \subseteq A, |U| > |V|\}$.
5. *The* modularity quantifier: $C_k = \{ (A,U) \mid |U| \equiv 0 \pmod{k}\}$.
6. *The* isomorphism quantifier: $C_\cong = \{ (A,E,F) \mid (A,E),$ (A,F) *are isomorphic graphs*$\}$.
7. *The* transitive closure quantifier: $C_{TC} = \{ (A,E,\{(u,v)\}) \mid$ *there is a path from* u *to* v *in the graph* $E \subseteq A \times A\}$.

All these quantifiers are decidable over finite domains; however, undecidable quantifiers in this context might be employed as well. E.g., for a Gödelization $G(T)$ *of Turing machines* T,

8. $C = \{(A,U) \mid$ *the Turing machine* T *with* $G(T) = |U|$ *halts.*$\}$

Or, suppose $P_1, P_2, \ldots$ *is a recursive enumeration of logic programs; define*

9. $C = \{(A,U,V) \mid P_{|U|}$ *and* $P_{|V|}$ *are equivalent over all models.*$\}$.

3 Logic Programs with Generalized Quantifiers

In this section, we consider the extension of logic programming with GQs. We first provide a syntax of logic programs with GQs, and then we attempt to define the semantics of such programs, starting from a semantics $\mathcal{M}$. The latter is not straightforward, and may be difficult for arbitrary programs. For restricted classes of programs, in particular for stratified programs and for positive programs using monotone GQs, however, an intuitive extension seems possible. Starting from scratch, we propose in the spirit of [21] the concept of GQ stable model, which serves to select the preferred models among the models of a logic program with GQs.

3.1 Syntax

The syntax of a *logic program with GQs* (*GQLP*) is like the syntax of an ordinary logic program, with the only difference that the literals B_i in the rule (1) may also be GQ-literals, which are defined next.

Suppose that Q is a GQ defined over the signature $\sigma = (R_1, \ldots, R_n, R_{n+1})$ with associated arities $n_1, \ldots, a_n, a_{n+1} = l$, and that $S_1, \ldots, S_n$ are predicates from the signature of an ordinary logic program P. Then, the formula

$$Q\mathbf{x}_1 \cdots \mathbf{x}_{n+1}[S_1(\mathbf{x}_1), \ldots, S_n(\mathbf{x}_n), \mathbf{x}_{n+1} = \mathbf{v}](\mathbf{v}) \qquad (3)$$

is a GQ-atom (w.r.t. P). Notice that the free variables of this GQ-atom are $\mathbf{v} = v_1 \ldots, v_l$ (cf. the previous section) The formula $\mathbf{x}_{n+1} = \mathbf{v}$ defines the singleton relation $\{\mathbf{v}\}$; intuitively, it transfers domain elements into the quantifier. A *GQ-literal w.r.t.* P is a possibly negated GQ-atom w.r.t. P. For brevity, we denote a GQ atom (3) by

$$Q[\mathbf{S}](\mathbf{v}) \qquad \text{(resp. } Q[\mathbf{S}]\text{, if } \mathbf{v} \text{ is void)} \qquad (4)$$

where $\mathbf{S} = S_1, \ldots, S_n$, and similar for negative GQ-literals.

Therefore, a logic program with generalized quantifiers can syntactically be seen as an ordinary logic program over an extended signature. Let τ be a signature, and let $\mathcal{Q}$ be a collection of generalized quantifiers. Denote by $\tau_{\mathcal{Q}}^*$ the extension of τ by all predicate letters $Q[\mathbf{S}]$, where $Q \in \mathcal{Q}$ and $\mathbf{S} = S_1, \ldots, S_n$ is a list of predicate letters S_i from τ which matches the signature of Q; every such $Q[\mathbf{S}]$ is a *GQ-predicate*. Notice that $\tau_{\mathcal{Q}}^*$ is finite if τ and $\mathcal{Q}$ are finite.

A logic program P over τ with GQs from $\mathcal{Q}$ is then syntactically an ordinary logic program over $\tau_{\mathcal{Q}}^*$.

Example 2 *Consider the following program P, which uses the isomorphism GQ $C_{\cong}$ and the transitive closure GQ C_{TC}:*

$$\begin{aligned} S(x,y) &\leftarrow C_{TC}[E](x,y), C_{TC}[E](y,x) \\ G_a(x,y) &\leftarrow E(x,y), S(a,x), S(a,y) \\ G_b(x,y) &\leftarrow E(x,y), S(b,x), S(b,y) \\ Iso &\leftarrow C_{\cong}[G_a, G_b] \end{aligned}$$

Intuitively, this program should compute Iso true on a graph with edges E if the strongly connected components in which a and b lie are isomorphic.

Some remarks about the use of GQs as proposed above are in order. In first-order logic, the scope of a GQ $Q\mathbf{x}_1 \cdots \mathbf{x}_n[\varphi_1, \ldots, \varphi_n](\mathbf{v})$ may contain arbitrary first-order formulas φ_i as well as new generalized quantifiers. In the context of logic programming, we do not need this possibility because formulas φ_i can be associated with an equivalent auxiliary relation A_{φ_i} which is computed within the logic program. The restriction of free variables to the formula $\mathbf{y} = \mathbf{v}$ is not severe; roughly, free variables $\mathbf{z}$ in a formula $\varphi(\mathbf{x}_i, \mathbf{z})$ can be treated as bound and newly introduced free variables $\mathbf{z}'$ (put in $\mathbf{v}$) can be used to single out the relation $\varphi^{\mathcal{A}}$ for the semantical definition of the GQ. Therefore, the syntactical restriction of GQ-atoms is not essential.

3.2 Semantics

In the context of first-order logic, the evaluation of formulas with generalized quantifiers can be easily defined by structural induction. However, in logic programming, we have to deal with recursion, which makes the definition of a semantics nontrivial, in particular if negation is involved.

It is immediately clear that similar difficulties arise if we incorporate GQs into logic programs, since negation can be provided by the complement GQ $C_{\sim}$. Therefore, any semantics of positive logic programs with GQs implicitly provides a semantics for logic programs with negation. Since there is no general consensus about the semantics of a logic program with negation, a generally accepted semantics for GQLPs may not be expected.

However, as discussed above, all reasonable semantics $\mathcal{M}$ for logic programs coincide on stratified logic programs, in which negation is evaluated along the strata of the program. A generalization to programs in which GQs are evaluated in a stratified way appears to be natural.

For this reason, we extend the concept of stratification GQLPs. A GQLP P is *stratified*, if the ordinary logic program P^* obtained from P by adding all rules $Q[\mathbf{S}](\mathbf{v}) \leftarrow$

$\neg S_1(\mathbf{x}_1), \ldots, \neg S_n(\mathbf{x}_n)$, where $\mathbf{S} = S_1, \ldots, S_n$ and the GQ predicate $Q[\mathbf{S}]$ occurs in P, is stratified. (Recall that a logic program is stratified, if its dependency graph does not contain a directed cycle containing a negative edge. We defer detailed definitions and a finer grained syntactic classification of GQLPs to Section 5.)

A stratified GQLP can be naturally evaluated stratum by stratum, where evaluation of each stratum intuitively reduces to a positive logic program. Guided by this intuition, we define a meaning $\mathcal{M}^*$ for ordinary logic programs from a meaning $\mathcal{M}$ of ordinary logic programs as follows.

Let P be a GQLP over signature τ. For any structure $\mathcal{A}$ over τ, denote by $GQ(\mathcal{A}, P)$ the relations for the GQ-predicates $Q[\mathbf{S}]$ of P such that $Q[\mathbf{S}]^{GQ(\mathcal{A},P)} = \{\mathbf{t} \mid \mathcal{A} \models Q[\mathbf{S}](\mathbf{t})\}$, i.e., the relation for $Q[\mathbf{S}]$ are the tuples $\mathbf{t}$ for which the ground formula $Q[\mathbf{S}](\mathbf{t})$ is true on $\mathcal{A}$. We call the extension of τ by the GQ-predicates of P the *GQ-signature of* P, which is denoted by τ_P^*.

Definition 3.1 *Let $\mathcal{M}$ be a semantics for ordinary LPs. The semantics $\mathcal{M}^*$ for a GQLP P over a structure $\mathcal{A}$ for its signature τ is defined as the intersection of all structures $\mathcal{B}$ for the signature τ_P^* which satisfy the equation*

$$\mathcal{B} = \mathcal{M}_P(\mathcal{A}, GQ(\mathcal{B}, P)), \tag{5}$$

i.e., P seen as an ordinary logic program and evaluated on $\mathcal{A}$ extended by relations for the GQ-literals of P w.r.t. B*, yields the structure $\mathcal{B}$. (The intersection of a collection of structures $\mathcal{S}$ is componentwise.)*

If we fix a domain and interpretations of logical constants (usually, Herbrand universe and interpretation, respectively), then we simply speak of interpretations I of the predicates rather than structures; equation (5) then amounts to

$$I = \mathcal{M}_P(\mathbf{R}, GQ(I, P)) \tag{6}$$

where $\mathbf{R}$ are input relations.

Example 3 *Reconsider the program P in Example 2. Suppose the input relation is $E = \{(a,b),(a,c),(c,a),(b,b)\}$ and consider Herbrand interpretations over a,b,c. Let $I = \{E(a,b), E(a,c), E(c,a), E(b,b)\}$. Then, in $GQ(I,P)$ the relations for the GQ-predicates are: $C_{TC}[E] = \{(a,b), (a,c), (a,a), (c,a), (b,b)\}$ and $C_{\cong}[G_a, G_b] = \{()\}$, i.e., $C_{\cong}[G_a, G_b]$ is true. Interpretation I does not satisfy equation (6) for any reasonable $\mathcal{M}$: E.g., $S(a,b) \notin I$, while $S(a,b)$ is true in P over $E \cup GQ(I,P)$.*

On the other hand, the interpretation J in which E is like in I, $C_{TC}[E]$ is the transitive closure of E, S tells whether two vertices are in the same strongly connected component of E, G_a resp. G_b are the edges from E between vertices from the strongly connected component of a resp. b, and both $C_{\cong}[G_a, G_b]$, Iso are false satisfies equation (6). In fact, J is the only interpretation for E with this property.

Notice that if P has no GQ-literals, then equation (5) resp. (6) amounts to $\mathcal{B} = \mathcal{M}_P(\mathcal{A})$; therefore, it has a unique solution and $\mathcal{M}_P^*(\mathcal{A}) = \mathcal{M}(\mathcal{A})$. Thus,

Proposition 3.1 *$\mathcal{M}^*$ is a conservative extension of $\mathcal{M}$ from ordinary LPs to GQLPs.*

For stratified programs with GQ-literals, $\mathcal{M}^*$ has the following property.

Proposition 3.2 *Let P be a GQLP. If P is stratified, then the equation (5) (resp. 6) has a unique fixpoint.*

Proof. This can be shown by applying the following well-known property of stratified ordinary logic programs.

Lemma 3.3 *Let $P = P_1 \cup \cdots \cup P_k$ be a decomposition of a stratified ordinary LP into strata P_i, $i = 1, \ldots, k$. Denote $P_{<k} = P_1 \cup \cdots \cup P_{k-1}$. Then,*

$$\mathcal{M}_P(\mathcal{A}) = \mathcal{M}_{P_{<k}}(\mathcal{A}) \cup \mathcal{M}_{P_k}(\mathcal{A} \cup \mathcal{M}_{P_{<k}}(\mathcal{A})).$$

Let $\mathcal{B}_1, \mathcal{B}_2$ be fixpoints, i.e., solutions, of the equation (5). We show $\mathcal{B}_1 = \mathcal{B}_2$ by induction on the number k of strata in the stratified ordinary logic program $P^* = P_1^* \cup \cdots \cup P_k^*$ which results from P by adding the rules for the GQ-literals (cf. above; we have to use P^* instead of P since intuitively, P^* locates the GQ-literals of P, which are in stratum 0 there, into the appropriate strata.)

(Basis) $k = 1$. Then, P is a positive program, and all GQ-literals in P are of the form $Q[](\mathbf{t})$, i.e., the GQ predicates of P have no input predicates. Clearly, $GQ(\mathcal{B}_1, P) = GQ(\mathcal{B}_2, P)$ in this case. Hence, $\mathcal{B}_1 = \mathcal{M}_P(\mathcal{A} \cup GQ(\mathcal{B}_1, P)) = \mathcal{M}_P(\mathcal{A} \cup GQ(\mathcal{B}_2, P)) = \mathcal{B}_2$.

(Induction) k. Assume the hypothesis holds for $k - 1$, and consider k. We have $P^* = P_1^* \cup \cdots \cup P_k^*$, $k > 1$. Let $P = P_1 \cup \cdots \cup P_k$ be the partition of P into strata P_i induced by the P_i^* (note that P_i's might be empty). Suppose $\mathcal{B}_1, \mathcal{B}_2$ are solutions of (5). Let for $i = 1, 2$ denote $\mathcal{B}_i^*$ the restriction of $\mathcal{B}_i$ to the signature of $P_{<k}$, and denote $\mathcal{C}_i = \mathcal{A} \cup GQ(\mathcal{B}_i^*, P_{<k})$. Then, by Lemma 3.3,

$$\mathcal{B}_i = \mathcal{M}_P(\mathcal{A} \cup GQ(\mathcal{B}_i, P)) = \mathcal{M}_{P_{<k}}(\mathcal{C}_i) \cup \mathcal{M}_{P_k}(\mathcal{A} \cup \mathcal{M}_{P_{<k}}(\mathcal{C}_i)).$$

Consequently,

$$\mathcal{B}_i^* = \mathcal{M}_{P_{<k}}(\mathcal{A} \cup GQ(\mathcal{B}_i^*, P_{<k}))$$

holds, i.e., $\mathcal{B}_i^*$ is a fixpoint of equation (5) for $P_{<k}$ on $\mathcal{A}$. Clearly, $P_{<k}^*$ has the stratification $P_1^* \cup \cdots \cup P_{k-1}^*$, which induces the stratification $P_{<k} = P_1 \cup \cdots \cup P_{k-1}$. Therefore, by the induction hypothesis, $\mathcal{B}_1^* = \mathcal{B}_2^*$ follows, which means $\mathcal{C}_1 = \mathcal{C}_2$. Consequently,

$$\begin{aligned} \mathcal{B}_1 &= \mathcal{M}_{P_{<k}}(\mathcal{C}_1) \cup \mathcal{M}_{P_k}(\mathcal{A} \cup \mathcal{M}_{P_{<k}}(\mathcal{C}_1)) \\ &= \mathcal{M}_{P_{<k}}(\mathcal{C}_2) \cup \mathcal{M}_{P_k}(\mathcal{A} \cup \mathcal{M}_{P_{<k}}(\mathcal{C}_2)) \\ &= \mathcal{B}_2. \end{aligned}$$

This concludes the induction and proves the result. ■

Notice that the proof shows how the meaning of $\mathcal{M}_P$ can be computed along the strata $P_1^*, \ldots, P_k^*$ of the program P^* associated with P. (Evaluate a GQ predicate $Q[\mathbf{S}]$ using the already computed predicates $\mathbf{S}$; don't use the "syntactic rule" $Q[\mathbf{S}] \leftarrow \neg S_1(\mathbf{x}_1), \ldots, \neg S_n(\mathbf{x}_n)$.)

For example, the program P in Example 2 has a unique fixpoint I on every input $\mathbf{R}$. We claim that the unique fixpoint of equation (5) captures the intuitive meaning of stratified GQLPs.

The natural question rises what to do with unstratified GQLPs, and how to properly define a semantics for such programs. Stratification can be violated in two ways: classically by a recursion through negation, or by recursion through GQ-predicates.

Example 4 *Consider the program*

$$\begin{aligned} C(x) &\leftarrow A(x), \neg C(x) \\ A(y) &\leftarrow Q[A] \end{aligned}$$

where Q is an arbitrary generalized quantifier. Here, the first clause violates stratification w.r.t. negation, and the second clause stratification w.r.t. GQ-predicates.

As pointed out above, recursion through GQ-predicates is a problem for the semantics, since GQ-atoms may behave nonmonotonically like negative literals. Stratification is a syntactical restriction which cautiously excludes such problems. Knowledge about semantical properties of the GQs, e.g., monotonicity, can be useful and help in finding a natural, intuitive semantics of GQLPs (see Section 3.3); in general, however, no such properties are available.

The common nonmonotonic nature of GQs and negative literals suggests a similar treatment in the definition of the semantics. Taking this point of view, different semantics $\mathcal{M}$ of ordinary logic programs can serve as a basis for the definition of semantics of GQLPs.

We focus here on semantics resorting to total models of logic programs, which are defined on every logic program. In particular, we consider the stable model semantics [21], which is to date the most well-accepted semantics having these properties. Similar considerations could be made for the well-founded semantics [49], partial stable models semantics [38], or regular semantics [52].

Our approach is to treat GQ-literals similar as negative literals; since GQs subsume negation, negation is implicitly treated with positive GQ-atoms. Under the assumption about truth of GQ-atoms and negative literals, a GQLP P is reduced similar as in [21] to an ordinary positive logic program and evaluated. If the unique minimal model of the latter program justifies the assumptions, it is considered an acceptable interpretation of P; we call such an interpretation a *GQ-stable model*.

We next provide a formalization of this concept. In what follows, we assume for simplicity that a domain and interpretations of logical constants are fixed. Moreover, we assume that no input relations are provided. Generalizations of the concepts to arbitrary structures and in presence of input relations are straightforward. We denote by $ground(P)$ the ground instance of P over the domain.

Definition 3.2 *Let P be a GQLP and I be an interpretation of the ordinary predicates τ. The* reduct of P by I, *denoted P^I, as the set of rules obtained from $ground(P)$ as follows.*

1. *Remove every rule r with a literal L in the body which is either negative or a GQ-literal and $I \not\models L$. (See Section 2.2 for the evaluation of GQ-literals.)*

2. Remove all negative literals and GQ-literals from the remaining rules.

Definition 3.3 *Let P be a GQLP. An interpretation M is a* GQ-stable model of P *iff M is the minimal model of the ordinary logic program P^M. We denote by $SM(P)$ the collection of all stable models of P.*

The meaning $\mathcal{M}^{st}$ *of P is the intersection of all GQ-stable models of P, i.e.,*

$$\mathcal{M}_P^{st} = \bigcap_{M \in SM(P)} M.$$

Example 5 *Consider the following program P, which uses the majority quantifier C_M and the modularity quantifier C_2:*

$$\begin{aligned} &W(a,b). \\ Q(x) &\leftarrow \neg S(x) \\ S(x) &\leftarrow \neg Q(x) \\ A(x) &\leftarrow C_M[Q,S], S(x) \\ Q(x) &\leftarrow C_2[A], S(x) \end{aligned}$$

Intuitively, the first two clauses choose complementary extensions for Q and S; the third clause assures that S implies A, if Q holds on more individuals than S; similarly, the fourth clause assures that S implies Q, if A holds true on an even number of elements.

Let $I_1 = \{W(a,b),\ Q(a),\ Q(b)\}$. This interpretation is a cGQ-stable model of P. Indeed, the reduct P^{I_1} consists of the clauses

$$\begin{aligned} &W(a,b). \\ Q(a) &\leftarrow \\ Q(b) &\leftarrow \\ A(a) &\leftarrow S(a) \\ A(b) &\leftarrow S(b) \\ Q(a) &\leftarrow S(a) \\ Q(b) &\leftarrow S(b) \end{aligned}$$

Clearly, I_1 is the minimal model of this program. Another stable model of P is $I_2 = \{W(a,b),\ S(a),\ S(b)\}$. The interpretation $I_3 = \{W(a,b),\ Q(a),\ S(b)\}$ is not a stable model of P: P^{I_3} contains the clauses $S(b) \leftarrow$ and $Q(b) \leftarrow S(b)$, which means that the minimal model of P^{I_3} contains $Q(b)$. It holds that I_1 and I_2 are all the stable models of P, i.e., $SM(P) = \{I_1,\ I_2\}$.

Let us note some interesting properties of GQ-stable models. Clearly, GQ-stable models conservatively extend the concept of stable models for ordinary logic programs.

Proposition 3.4 *Let P be a GQLP. If P has no GQ-literals, then M is a GQ-stable model of P iff M is a stable model of P.*

Therefore, we refer in the sequel to GQ-stable models simply as stable models. The next proposition tells that the term "model" is justified.

Proposition 3.5 *Let M be a stable model of a GQLP P. Then, M is a model of P, i.e., $M \models \forall^*(P)$, where $\forall^*$ is the universal closure of P.*

We also observe that, unlike in the case of ordinary logic programs, the stable models need not form an antichain. That is, a stable model of a GQLP is not necessarily a minimal model.

Example 6 *Consider the program P*

$$B(a) \leftarrow C_\forall[B]$$

and consider Herbrand interpretations. This program has two stable models: $M_1 = \{\}$ and $M_2 = \{B(a)\}$. Indeed, P^{M_1} is empty, and M_1 is the minimal model thereof; P^{M_2} consists of the clause $B(a) \leftarrow$, and hence M_2 is the minimal model of P^{M_2}.

This example rephrases the well-known theory $\{Lp \supset p\}$ of autoepistemic logic, and shows that intuitively weakly grounded stable models of GQLPs might exist. This can be handled by restricting attention to minimal stable models, in the spirit of ground logics [42, 11]. For programs like in the example, minimal stable models can be captured by means of a consequence operator, as shown in the next section. Notice that with respect to inference of positive literals, cautious reasoning from all stable models amounts to inference from the minimal ones; therefore, we do not further pursue this issue here.

On stratified GQLPs, the concept of stable model behaves as desired. In particular, the following properties hold.

Proposition 3.6 *Let P be a stratified GQLP. Then, $\mathcal{M}_P^* = \mathcal{M}_P^{st}$. Moreover, P has a unique stable model, which coincides with the unique fixpoint of the equation (6).*

Proof. The proof uses the following lemmata; the first is well-known (cf. [38]).

Lemma 3.7 *Let P be a stratified ordinary logic program. Then, P has unique stable model, which coincides with its unique stratified model.*

Lemma 3.8 *Let P be a GQLP on signature τ. Then, I is a stable model of P if and only if $I \cup GQ(I, LP)$ is a stable model of the ordinary LP $P' = P \cup GQ(I, P)$.*

Hence, I is a stable model of P

$\longleftrightarrow$ $I \cup GQ(I, P)$ is a stable model of P' (Lemma 3.8)

$\longleftrightarrow$ $I \cup GQ(I, P)$ is the stratified model of P' (P' is stratified, Lemma 3.7)

$\longleftrightarrow$ $I \cup GQ(I, P)$ is solution of $J = \mathcal{M}_{P'}(GQ(J, P'))$ (Proposition 3.2).

$\longleftrightarrow$ I is solution of $J = \mathcal{M}_P(GQ(J, P))$

Thus the stable models of P coincide with the fixpoints of equation (6). Consequently, P has a unique stable model, which coincides with the unique fixpoint of (6). This proves the result. ∎

3.3 Monotone Generalized Quantifiers

In lack of knowledge of any properties of GQs, we proposed above a treatment of GQ-predicates which is similar as for negation. In case some properties of the GQs involved are known, more sophisticated approaches to the semantics of a GQLP are imaginable.

Of particular interest is logic programming extended with monotone GQs. A GQ Q is *monotone*, if its truth monotonically increases with extensions of the input relations, i.e., if $(\mathcal{A}, \mathbf{b}) \models Q\mathbf{x}[\varphi](\mathbf{y})$, then $(\mathcal{B}, \mathbf{b}) \models Q\mathbf{x}[\varphi](\mathbf{y})$ for every structure $\mathcal{B}$ such that $\mathcal{A}$ is a substructure of $\mathcal{A}$. Denote the class of positive GQLPs which use monotone GQs by **Pos(Mon)**.

The semantics of programs P from **Pos(Mon)** can be naturally defined in terms of the least fixpoint of an immediate consequence operator. As in the previous section, we assume here that the domain and the interpretation of logical constants is fixed, and, furthermore, disregard input predicates. Recall that *ground*(P) is the ground instance of P over the domain.

Definition 3.4 *Let P be a program from* **Pos(Mon)**. *The operator T_P over interpretations I is defined by*

$$\begin{array}{rcl} T_P(I) & = & \{A \mid \text{ there is a rule } A \leftarrow B_1, \ldots, B_m \\ & & \qquad \text{ in } ground(P) \text{ such that } I \models B_i,\ 1 \leq i \leq m\} \end{array}$$

Lemma 3.9 *For every P from* **Pos(Mon)**, *$T_P(\cdot)$ is a monotone operator.*

Proof. Follows immediately from the fact that all GQs in P are monotone. ■

Corollary 3.10 *Let P be a program from* **Pos(Mon)**. *Then, the operator T_P has a least fixpoint T_P^∞, given by the limit of the sequence T_P^α for ordinals $\alpha \geq 0$, where*

$$\begin{array}{rcll} T_P^0 & = & \emptyset, & \\ T_P^{\alpha+1} & = & T_P(T_P^\alpha), & \text{for every successor ordinal } \alpha+1, \text{ and} \\ T_P^\alpha & = & \bigcup_{\beta<\alpha} T_P^\beta, & \text{for every limit ordinal } \alpha. \end{array}$$

Definition 3.5 *The least fixpoint semantics of a program P from* **Pos(Mon)**, *denoted $\mathcal{M}_P^{lfp}$ is defined by $\mathcal{M}_P^{lfp} = T_P^\infty$.*

Example 7 *Reconsider the program P from Example 6:*

$$B(a) \leftarrow C_\forall[B]$$

Here, $T_P^0 = \emptyset$, $T_P^1 = T_P(\emptyset) = \emptyset$; therefore, $T_P^\infty = T_P^0 = \emptyset$, and hence $\mathcal{M}_P^{lfp} = \emptyset$.

We remark that unlike with ordinary logic programming, the least fixpoint of T_P may not be reached in ω steps. The reason is that T_P is not finitizable, i.e., $T_P(I) \neq \bigcup_{|J|<\aleph_0} T_P(J)$ in general. To see this, consider the following example.

Example 8 *Consider the program P, which uses the GQ $C_\forall$:*

$$
\begin{aligned}
A(0)&.\\
A(f(x)) &\leftarrow A(x)\\
B &\leftarrow C_{\forall}[A]
\end{aligned}
$$

Suppose we consider Herbrand interpretations. Then, $T_P^1 = \{A(0)\}$, $T_P^2 = \{A(0), A(f(0))\}$, $\ldots$, $T_P^i = \{A(f^j(0)) \mid 0 \leq j < i\}$, $\ldots$, $T_P^\omega = \{A(f^i(0)) \mid 0 \leq i\}$ *and* $T_P^{\omega+1} = T_P^\omega \cup \{B\} = T_P^\infty$.

In case of ordinary logic programming, the definitions of the semantics of positive logic programs in terms of the least fixpoint of the immediate consequence operator and in terms of stable models are well known to coincide. In fact, the least fixpoint T_P^∞ of such a program P, which is given by T_P^ω, yields the unique stable model. A similar property holds on programs from **Pos(Mon)**.

Theorem 3.11 *Let P be a program from* **Pos(Mon)**. *Then, $\mathcal{M}_P^{st}$ and $\mathcal{M}_P^{lfp}$ coincide. In particular, P has a unique least stable model, which coincides with T_P^∞.*

Proof. Let M be a stable model of P. It is easy to see from Proposition 3.5 that $T_P(M) \subseteq M$ holds in this case. On the other hand, since M is the least model of the ordinary program P^M, it follows by well-known results that $M = T_{P^M}^\infty$. Therefore, for every atom $A \in M$, there is a rule r in $ground(P^M)$ which derives A. Consequently, there is a rule in $ground(P)$ which derives A w.r.t. M, i.e., $A \in T_P(M)$. Therefore $M \subseteq T_P(M)$ holds. This establishes the following.

Lemma 3.12 If M is a stable model of P, then $M = T_P(M)$.

Thus, in order to prove the result, it suffices by Corollary 3.10 to show that T_P^∞ is a stable model of P. That is, we have to check $T_P^\infty = T_{P'}^\infty$, where $P' = P^{T_P^\infty}$ is the reduct of P w.r.t. T_P^∞.

The following inclusions are easy to see:

$$
\begin{aligned}
T_P(I) &\subseteq T_{P'}(I), &&\text{for every } I \subseteq T_P^\infty; &(7)\\
T_{P'}(I) &\subseteq T_P(I \cup T_P^\infty), &&\text{for every } I. &(8)
\end{aligned}
$$

Therefore,

$$T_P^\infty = T_P(T_P^\infty) \subseteq T_{P'}(T_P^\infty) \subseteq T_P(T_P^\infty \cup T_P^\infty) = T_P^\infty$$

holds; this implies that T_P^∞ is a fixpoint of $T_{P'}$. In order to establish $T_P^\infty = T_{P'}^\infty$, it thus remains to show by transfinite induction that $T_P^\alpha \subseteq T_{P'}^\infty$, for every ordinal α. Clearly, $T_P^0 = \emptyset \subseteq T_{P'}^\infty$. For the induction step, for successor ordinals $\alpha + 1$,

$$
\begin{aligned}
T_P^{\alpha+1} &= T_P(T_P^\alpha) && \text{by definition of } T_P\\
&\subseteq T_{P'}(T_P^\alpha) && \text{by (7), as } T_P^\alpha \subseteq T_P^\infty\\
&\subseteq T_{P'}(T_{P'}^\infty) && \text{induction hypothesis}\\
&= T_{P'}^\infty && T_{P'}^\infty \text{ is fixpoint of } T_{P'};
\end{aligned}
$$

for limit ordinal α,

$$
\begin{aligned}
T_P^\alpha &= \textstyle\bigcup_{\beta<\alpha} T_P^\beta && \text{by definition of } T_P\\
&\subseteq \textstyle\bigcup_{\beta<\alpha} T_{P'}^\infty && \text{induction hypothesis}\\
&= T_{P'}^\infty.
\end{aligned}
$$

■

For example, the least fixpoints $T_P^\infty = T_P^0$ of the programs in Examples 6,7 are empty and coincide with their unique minimal stable models. The least fixpoint $T_P^\infty = T_P^{\omega+1}$ of the program P in Example 8 coincides with its unique stable model (notice that P is stratified).

Remark. With respect to inference of negative literals, the stable model semantics and the least fixpoint semantics are different even on **Pos(Mon)**, since a program from **Pos(Mon)** can have more than one stable model in general (cf. Example 6). However, the previous theorem tells us that the two approaches can be reconciled if we eliminate nonminimal stable models, i.e., retain only the stable models which do not properly contain any other stable model. In this case, both semantics infer the same sets of literals.

4 Modular Logic Programs

In this section, we demonstrate that generalized quantifiers are an appropriate tool for formalizing modular logic programming. Logic programming with generalized quantifiers facilitates an appealing and logically clean approach to define the semantics of logic programs which use program modules. This follows from the observation that a program module can be naturally seen as a generalized quantifier.

There is a number of approaches to modular logic programming, which greatly differ in syntax and semantics; see [5] for a comprehensive survey. We follow here the direction which promotes the use of modules at the extensional level, and avoid thus the intricacies of the use of intentional definitions of predicates.

We employ here the concept of a module familiar from imperative programming languages, which allows reuse of a piece of code under a notion of scope and visibility. A program module is syntactically described by its interface in terms of input and output parameters. Adapted to the logic programming paradigm, a logic programming module has an interface

$$LP[\mathbf{I}; \mathbf{O}],$$

where the $\mathbf{I} = I_1, \ldots, I_m$ are predicates which are imported into the module (i.e., input predicates) and the $\mathbf{O} = O_1, \ldots, O_n$ are predicates which are exported (i.e., output predicates). If we describe the semantics of a module at the extensional level (i.e., if we describe the extensions of the O_j as functions of the predicate extensions of the I_k rather than by the definitions of these predicates) and restrict the number of output relations to $n = 1$ it is clear that such a module is a generalized quantifier. The latter restriction is not essential, however. For each O_j, a particular module LP_j could be defined; using designated constants, the predicates O_j could be represented by a single predicate, etc.

Consequently, a module $LP[\mathbf{I}; O_1]$ can be abstractly seen as a GQ $Q_{LP}[\mathbf{I}](\mathbf{x})$. The semantics of a modular logic program is then inherited from the semantics of a GQLP. Notice that this approach is very flexible, and allows for easy incorporation of external predicates, e.g., averaging and counting predicates.

4.1 Syntax of modular logic programs

The syntax of *modular logic programs* (*MLPs*) is the one of GQLPs, with the difference that the GQ-literals are intended to refer to another logic program, the *logic program module*. We shall refer to the calling program as the *main program*, and the called module as the *subprogram*; the GQ-literals in a MLP are termed *call literals*, and the GQ-predicates *call predicates*. The similarity type of $LP[\mathbf{Q}]$ is the list of arities of predicates in $\mathbf{Q}$.

An atom (resp. literal) is a call atom (resp. literal) if its predicate is a call predicate.

To distinguish ordinary predicates, atoms, and literals from call predicates, call atoms, and call literals, we call the former standard ones.

Definition 4.1 *A* logic program module *is a pair* $\mu = (LP[\mathbf{Q}], P)$ *of a* module head $LP[\mathbf{Q}]$, *which has an associated integer* $n \geq 0$ (the arity), *and an ordinary logic program* P (the body), *in which the predicates* $\mathbf{Q}$ *are the input predicates and* LP *is the output predicate having arity* n*; syntactically, occurrences of the predicates* $\mathbf{Q}$ *in* P *are restricted to rule bodies. We require that each* LP *module is uniquely identified by its* name LP *and the list of the arities of the* Q_i *in* $\mathbf{Q}$ (*its* similarity type).

In this definition, the predicates $\mathbf{Q}$ serve as formal parameters of the module, which are substituted by actual parameters at a call. All predicates in a module $LP[\mathbf{Q}]$ are local; no predicates from outside are visible inside a module. Everything which should be visible must be input via a parameter Q_i. It is possible to extend module heads by allowing that besides predicates also individual variables occur as parameters. However, as we have seen for GQs, such parameters are not necessary, since they can be provided through designated predicates.

Note that the program P does not contain call literals, and hence any semantics of ordinary logic programming can be readily applied on P. This restriction can be removed and call- resp. GQ-literals may be admitted, provided that some basic constraints are obeyed; we shall address this issue in Section 4.3.

The relations in P are not visible from the main program, and vice versa. Thus, relations are imported to the subprogram only if they appear in the scope of a call literal, and exported only as the extension of a call literal.

Definition 4.2 *A* modular logic program *(MLP) is a finite collection* P *of rules*

$$A \leftarrow L_1, \ldots, L_m$$

where A *is a standard atom and the* L_i *are literals, plus a collection* $\mathcal{C}$ *of logic program modules such that for each call literal* $(\neg)LP[\mathbf{Q}](\mathbf{t})$ *occurring in* P*, there is a module* $LP[\mathbf{Q}^*]$ *in* $\mathcal{C}$ *where* LP *has the arity of* $\mathbf{t}$ *and each* $Q_i \in \mathbf{Q}$ *has the arity of* $Q_i^* \in \mathbf{Q}^*$.

Example 9 *Consider the following program:*

$$\begin{aligned} Q &\leftarrow 3Uncol[G_1], \neg 3Uncol[G_2] \\ S(x,y) &\leftarrow Q, TC[G_2](x,y), \neg TC[G_1](x,y) \end{aligned}$$

Here $3Uncol[G]$ *and* $TC[G]$ *are modules for deciding uncolorability of a graph with three colors and computing its transitive closure, respectively.*

The program should intuitively accomplish the following: If the graph G_1 is not 3-colorable and the graph G_2 is 3-colorable, then S is assigned the difference of the transitive closures of G_2 and G_1; otherwise, S is empty.

The body of $TC[G]$ resp. $3Uncol[G]$ could be as follows.

$$
\begin{array}{ll}
TC[G]: & 3Uncol[G]: \\
\quad TC(x,y) \leftarrow G(x,y) & \quad Green(x) \leftarrow \neg Red(x), \neg Blue(x) \\
\quad TC(x,y) \leftarrow TC(x,z), G(z,y) & \quad Red(x) \leftarrow \neg Green(x), \neg Blue(x) \\
 & \quad Blue(x) \leftarrow \neg Red(x), \neg Green(x) \\
 & \quad 3Uncol \leftarrow G(x,y), Red(x), Red(y) \\
 & \quad 3Uncol \leftarrow G(x,y), Green(x), Green(y) \\
 & \quad 3Uncol \leftarrow G(x,y), Blue(x), Blue(y)
\end{array}
$$

The stable models of the body of $3Uncol[G]$ correspond 1-1 to the assignments of one of three colors to each vertex in G. Hence, under the stable semantics, $3Uncol$ is true precisely if the graph G is not 3-colorable.

4.2 Semantics of modular logic programs

As discussed above, we can see LP modules as generalized quantifiers. Thus, by exploiting this correspondence, we can define the semantics of MLPs in terms of the semantics of GQLPs.

Every LP module $\mu = (LP[\mathbf{Q}], P)$ can be seen under a semantics $\mathcal{M}$ as a GQ $Q_{C(\mu;\mathcal{M})}$ which is associated with the collection $C(\mu;\mathcal{M})$ of all structures $\mathcal{A} = (A, \mathbf{Q}^*, R^*)$, where the $\mathbf{Q}^*$ are relations for the predicates $\mathbf{Q}$ on A and $R^* = \{\mathbf{a}\}$, for any tuple $\mathbf{a}$ over A such that P derives the fact $LP(\mathbf{t})$ on $\mathcal{A}$; i.e.,

$$C(\mu;\mathcal{M}) = \{(A, \mathbf{Q}^*, \{\mathbf{a}\}) \mid \mathcal{M}_P((A, \mathbf{Q}^*)) \models LP(\mathbf{a})\}$$

(Notice that $C(\mu;\mathcal{M})$ is closed under isomorphism, if we consider logic program over general structures). We call $Q_{C(\mu;\mathcal{M})}$ the *module quantifier of μ under $\mathcal{M}$.*

Definition 4.3 *Let P be an MLP. The meaning of P under $\mathcal{M}$ is defined as the meaning of P, viewed as GQLP over the collection of GQs associated with the LP modules used by P.*

In the rest of this paper, we focus on $\mathcal{M}^{st}$ and $\mathcal{M}^{lfp}$ for positive programs with monotone associated GQs.

Example 10 *Let I be an interpretation of a binary predicate E and a unary predicate U, e.g. $I = \{E(1,2), E(2,3), E(3,1), E(2,4), U(1)\}$, and consider the module $TC[G]$ from above.*

$$
\begin{tikzcd}
1 \arrow[r] & 2 \arrow[dl] \\
3 \arrow[u] & 4 \arrow[u]
\end{tikzcd}
$$

For evaluating this module on input I under $\mathcal{M}^{lfp}$, we augment the body by the ground facts for the input according to I, i.e., the facts $G(1,2)$, $G(2,3)$, $G(3,1)$, and $G(2,4)$; the resulting program is:

$$\begin{array}{rlr} TC(x,y) & \leftarrow G(x,y) & G(2,3) \leftarrow \\ TC(x,y) & \leftarrow TC(x,z), G(z,y) & G(2,4) \leftarrow \\ G(1,2) & \leftarrow & G(3,1) \leftarrow \end{array}$$

The least fixpoint of this program contains in addition the facts $TC(i,j)$, $1 \leq i,j \leq 3$ and $TC(k,4)$, $1 \leq k \leq 3$.

Remark: Certainly, not all generalized quantifiers used have to be program quantifiers. In fact, calls to library modules and external modules, formalized as generalized quantifiers, can be incorporated easily. The evaluation adds then the ground facts $P[\mathbf{Q}](\mathbf{t})$ which are provided by the library.

Example 11 *Consider again the module $TC[G]$, together with the following program P:*

$$\begin{array}{rl} R(x,y) & \leftarrow E(x,y) \\ R(x,y) & \leftarrow E(y,x) \\ C & \leftarrow \neg C, E(a,y), E(b,y), a \neq b \\ D & \leftarrow \neg TC[R](x,x), \neg D \\ U(x) & \leftarrow TC[E](x,y), U(y) \\ U(x) & \leftarrow \neg U^-(x), R(x,y) \\ U^-(x) & \leftarrow \neg U(x), R(x,y) \end{array}$$

The first two rules essentially assign the shade of graph E (i.e., the undirected graph induced by E) to R. Now, in any interpretation I, $TC[E]$ and $TC[R]$ hold the transitive closures of E and R, respectively. Since the main program guarantees that $TC[R]$ is the transitive closure of the shade of E, it is easy to see that the stable models of LP are those interpretations where E is a tree, and U is a subtree of E.

We conclude the exposition of the stable modular LP semantics by some notational convention. We introduce denial clauses

$$\perp \leftarrow B_1, \ldots, B_m$$

which are a shorthand notation for

$$C \leftarrow B_1, \ldots, B_m, \neg C$$

where C is a fresh propositional letter that does not occur elsewhere in the program. Such a clause does not admit any stable model in which all of $B_1, \ldots, B_m$ are true. In fact, if all B_i are true, then the clause basically reduces to the unstratified clause $C \leftarrow \neg C$; for the latter, no stable model exists.

4.3 Nested LP modules

So far, we have considered modular logic programs in which the modules used are ordinary logic programs. If we think in terms of subprograms, such programs have only one level of calls to subprograms. However, in real life programming there are multiple levels of calls to subprograms possible, and they are heavily used in practice. Therefore, it is necessary to extend our concept of MLPs by allowing that in the bodies of modules calls to other modules occur.

In principle, this is not a problem and a semantics can be given to nested MLPs based on the semantics for logic programs with GQs. This is straightforward, provided a hierarchic use of modules is obeyed. Formally, this is expressed by the following condition.

Define the *call graph* of a modular logic program P with modules $\mu_1, \dots, \mu_k$ as follows: the vertices are the main program, seen as module μ_0, and the modules $\mu_i = (LP_i[\mathbf{Q}_i], P_i)$, $i > 0$. There is an arc from μ_i to μ_j if the program P_i in the body of μ_i contains a call literal $LP_j[\mathbf{Q}_i^*](\mathbf{t})$.

We say that a program P is *hierarchical* if its call graph is acyclic. The *nesting depth* of P is the length of the longest path in its call graph starting from M_0; P is *finitely nested*, if its nesting depth is finite.

For finitely nested, hierarchical programs, a semantics can be inductively defined, based on the semantics of programs having nesting depth 2 (i.e., unnested MLPs). In particular, stable models of such programs can be naturally defined, as well as a stratified semantics and a semantics for positive nested programs (where all programs in all bodies are positive). We do not carry out this here; the formalization should be clear.

A semantics for nested MLPs with cyclic call graph (i.e., with mutual recursive module calls) is not straightforward.

5 A Taxonomy of Program Classes

As we have seen in Section 3, enhancing logic programming with GQs may lead to different violations of stratification. In particular, we have seen classical violation in terms of recursion involving negation, and violation by recursion through GQ literals. These different forms of violation motivate a more careful consideration of interdependencies between predicates in a logic program, which refines the simple stratified vs. unstratified paradigm. Properly distinguishing between ordinary predicates and GQ-predicates, we generalize the concept of dependency graph from ordinary LPs to GQLPs. The dependency graph serves as a basis for the definition of a number of natural syntactic subclasses of logic programs with GQs, which raise interesting questions concerning their interrelationships in terms of semantical and computational properties. These issues will be addressed in Section 6.

5.1 Dependency graph

Recall that for a GQLP LP, we denote by τ_{LP} (or simply τ) the signature of ordinary predicates of LP, and by τ^*_{LP} the extension of τ with the GQ-predicates in LP.

Definition 5.1 *Let LP be a GQLP. The* dependency graph *of LP, $D(LP)$, is the directed graph constructed as follows. The vertices of $D(LP)$ are the predicates in τ^*_{LP}; an arc from P_1 to P_2 is present in $D(LP)$ if one of 1.–3. applies:*

1. *there is a rule $P_1(\mathbf{t}_1) \leftarrow \dots, \neg P_2(\mathbf{t}_2), \dots$ in LP.*

2. *P_1 is a call predicate $P[\mathbf{Q}^*]$ and P_2 occurs in $\mathbf{Q}^*$.*

3. there is a rule $P_1(\mathbf{t}_1) \leftarrow ..., P_2(\mathbf{t}_2), ...$ in LP;

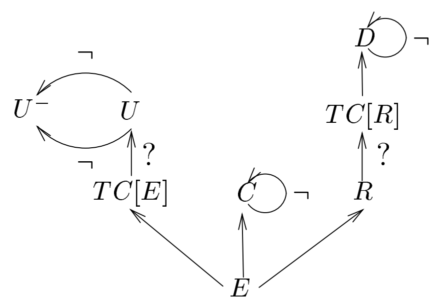
We color the arc

red ,	*if 1. applies and both P_1, P_2 are ordinary predicates;*
yellow ,	*if P_1 or P_2 is a GQ-predicate;*
green ,	*otherwise (i.e., 3. applies and both P_1 and P_2 are ordinary predicates).*

The color of an arc in $D(LP)$ reflects possible nonmonotonic dependency of a P_1 on P_2. A red arc intuitively represents that P_1 is nonmonotonically dependent on P_2, i.e., increasing P_2 decreases P_1; a yellow arc represents that P_1 possibly is nonmonotonically dependent on P_2 (this depends on the properties of the respective module involved); finally, a green arc represents that P_1 is monotonically dependent on P_2.

A predicate P_1 *depends* on P_2, if P_1 reaches P_2 in $D(LP)$. If P_1 and P_2 mutually depend on each other, they are called *mutually recursive*; if neither P_1 depends on P_2 nor vice versa, they are *independent*.

Example 12 *The dependency graph of the program in Example 11 is as follows (red arcs are symbolized by label "$\neg$", yellow arcs by "?"; green arcs are unlabeled).*



According to syntactic conditions on the dependency graph $D = D(LP)$, several classes of GQLPs are defined in Table 1.

Class	*Symbol*	*Description*
unrestricted	**LP**	arbitrary D
stratified	**Strat**	cycles in D have only green arcs
semistratified	**SStrat**	no cycle in D has a red arc
call recursion free	**CRF**	no cycle in D has a yellow arc
call independent	**CI**	no path in D has two yellow arcs
stratified call independent	**SCI**	cycles in D have only green arcs and no path in D has two yellow arcs
recursion-free	**RF**	D is acyclic
positive	**Pos**	D has no red arcs

Table 1: Classes of GQLPs

Notice that the classification in Table 1 also applies to ordinary logic programs; therefore, it provides implicitly syntactic classes of LP-modules $\mu = (LP[\mathbf{Q}], P)$ according to the class of P.

Since the dependency graph of an ordinary logic programs does not have yellow arcs, the classification in this case essentially gives rise to the familiar classes **Strat, RF, Pos, LP**.

Based on the classes of ordinary logic programs, we can further refine the above classification of MLPs, by taking into account restrictions on the subprograms. We use the following notation. Let $\mathcal{C}_M$ be a class of main programs, and let $\mathcal{C}_S$ be a class of subprograms. Then, $\mathcal{C}_M(\mathcal{C}_S)$ denotes the class of MLPs with main programs from $\mathcal{C}_M$ and subprograms from $\mathcal{C}_S$. According to this convention, each class $\mathcal{C}$ in Table 1 amounts to $\mathcal{C}(\mathbf{LP})$.

An extension of this notation to nested MLPs is at hand, by simply relaxing the class D to MLPs rather than ordinary logic programs. Thus, we can inductively define classes like **Strat**(**LP**(**Strat**)) and so on. Accordingly, $\mathbf{LP^k}$ is the class of all modular logic programs with nesting depth bounded by k, and $\mathbf{LP}^* = \bigcup_{k\geq 1} \mathbf{LP^k}$. In particular, $\mathbf{LP^1}$ is the class of ordinary logic programs. For the rest of this paper, we adopt this notation for the respective classes of hierarchical nested MLPs.

The concept of stratified modular program naturally extends the familiar notion of stratified logic program, while retaining some of its appealing properties, as shown in Section 3. Semistratified modular programs allow recursion through module calls, which in general means nonmonotonic recursion like recursion through negation; however, if the modules are monotone, we have positive recursion, which intuitively seems easier to deal with from the semantical point of view than arbitrary recursion. In order to account for this interesting issue, we shall also consider the use of subprograms from the class **Mon** of all monotone ordinary LPs in our study, and investigate the effect on computational properties.

Notice that **Mon** is defined semantically and not very interesting from the syntactical side since telling whether a (already stratified) LP is monotone is undecidable. However, in many cases the programmer knows that the modules used are monotone or can easily conclude this (e.g., if the program in the body is positive). Moreover, for fixed domains a syntactic characterization of **Mon** under the stable semantics can be given in terms of second-order logic [47, 46].

Several interesting issues come up with the classes of programs we have presented above, which will be tackled in the subsequent sections:

- What can be expressed by programs from these classes ?
- Do there exist (hopefully effective) translations from one class of programs to the other ?
- Do the classes have particular structural properties of classes? For example, does there exist a specific normal form for programs from a class ?

6 Expressiveness of Modular Logic Programs

In this section, we address the expressiveness of modular logic programming using stable model semantics. We focus in our analysis on finite relational structures, and leave other domains for future work, apart from a brief comment to expressiveness over the integers.

6.1 Preliminaries on expressive power

We assume some familiarity with the basic concepts of complexity theory, such as P, NP, L (logspace), oracle computations, and the polynomial hierarchy (PH). Consult [18, 36] for an introduction.

Intertwined with the levels Σ_k^p, Π_k^p of PH are the classes Θ_k^p [51] ($\Sigma_k^p \subseteq \Theta_{k+1}^p \subseteq \Sigma_{k+1}^p$), which are now considered constituting intermediate classes of PH. It turns out that the Θ_{k+1}^p classes have natural complete problems which are typically obtained as optimization variants of Σ_k^p-complete problems.

Since oracles can be seen as a logical concept of subprograms, the following identity due to Wagner will be most useful for our investigations. The class $\mathrm{P}_{\|}^{\mathrm{NP}}$ is the class of problems decidable by polynomial time Turing machines whose oracle queries are first collected, and then all queries are forwarded to the oracle together. Thus, the queries are independent of each other.

Proposition 6.1 [51] $\Theta_{k+1}^p = \mathrm{P}_{\|}^{\Sigma_k^p} = \mathrm{L}^{\Sigma_k^p} = \mathrm{P}^{\Sigma_k^p}[O(\log n)]$.

Here, the term $O(\log n)$ in the brackets indicates that the number of queries posed to the oracle is bound by $O(\log n)$, where n is the size of the input.

A Turing machine which computes an output rather than only telling yes or no is known as a *transducer*. The output of a transducer T_1 may be provided as input to another machine T_2; the composed machine is denoted by $T_1 \circ T_2$. This induces naturally analogous notation $C_1 \circ C_2$ for machine-based complexity classes C_1 and C_2. For the composition of Θ_i^p computations, we note the following easy lemma.

Lemma 6.2 *For any constants $k, i > 0$, $\underbrace{\Theta_i^p \circ \ldots \circ \Theta_i^p}_{k \text{ times}} = \Theta_i^p$.*

Proof. Immediate from the fact that $\Theta_{k+1}^p = \mathrm{P}^{\Sigma_k^p}[O(\log n)]$. ■

Let $\mathcal{M}$ be a semantics as defined in section , let C be a complexity class, and let $\mathcal{K}$ be a class of finite structures over a relational signature τ.

We say that $\mathcal{M}$ *has expressivity* C (or, equivalently, $\mathcal{M}$ *captures* C) *over* $\mathcal{K}$, if

1. for every program LP and structure $\mathcal{A}$ in $\mathcal{K}$, $\mathcal{M}_{LP}(\mathcal{A})$ is computable in C, and
2. every generic (i.e., isomorphism preserving) mapping (also called query) $f : \mathcal{K} \to \mathcal{K}$ computable in C is expressible by a program LP under the semantics $\mathcal{M}$, i.e., $f = \mathcal{M}_{LP}$.

Here computability of a mapping f in C means that deciding whether a relation in $f(\mathcal{A})$ holds a specific tuple is in C. Semantical restrictions on the maps f may be imposed. E.g., monotone co-NP is the class of all monotone maps f computable in co-NP.

Our interest is here in the class $\mathcal{F}$ of all finite structures and in $\mathcal{OF}$, the class of all *ordered* finite structures, which provide an enumeration of the domain A in a designated relation.

The expressive power of ordinary logic programming over finite structures has been investigated e.g. in [41, 14, 40]. Stable model semantics was shown to be quite expressive.

Proposition 6.3 ([41]) **LP** *captures* co-NP*, or equivalently, all* Π_1^1 *definable queries over finite structures.*

The *data complexity* of an LP class, which is the complexity of the mappings $\mathcal{M}_{LP}$ for fixed LP, follows immediately if it captures a class C over all (ordered) finite structures, and is C-completeness in this case. However, a class can be C-complete while it does not capture C; therefore, capturing C is a stronger concept than C-completeness. For example, **Pos** is known to be P-complete, while it does provably not capture P.

Besides data complexity, Vardi [50] further introduced *expression (resp. combined) complexity*, which amounts to the complexity of $\mathcal{M}_{LP}$ for varying LP over a fixed (resp. arbitrary) structure Struct; typically, expression and combined complexity coincide.

6.2 Results

Table 2 compactly lists the expressivity results that we derive in this paper, based on the stable model semantics $\mathcal{M}^{st}$. In the rest of this sections, we tacitly assume this semantics for all classes of programs considered. The definition of class **2Pos(Mon)** is deferred to section 6.5. An asterisk "*" indicates that the proof establishes capturing over ordered finite structures. Note that capturing of Θ_2^p over all finite structures is not possible unless some strongly believed hypotheses of complexity theory are false [7].

The results also indicate the possibility of intertranslations between different classes $\mathcal{C}_1$ and $\mathcal{C}_2$ of modular programs. If $\mathcal{C}_1$ has a complexity C_1 and $\mathcal{C}_2$ captures C_2, then $\mathcal{C}_1$ is translatable into $\mathcal{C}_2$ if C_1 is included in C_2, such that every program and its transform are equivalent on the respective class of structures (in fact, in all cases effective translations exist); in particular, if $\mathcal{C}_1$ and $\mathcal{C}_2$ capture the same class C, then mutual such translations exist. otherwise, this is not the case unless some strongly believed hypotheses of complexity theory are false.

6.3 Expressive power of MLPs with LP modules

The expressive power of modular logic programming under stable semantics is located at the second level of the polynomial hierarchy.

Theorem 6.4 LP(LP) *captures* Π_2^p *over finite structures.*

Proof. It is easily seen that the query computed by any program LP from **LP(LP)** is in Π_2^p. Indeed, the call literals can be evaluated in polynomial time using an NP-oracle, and thus the reduct LP^I an be computed in polynomial time with an NP oracle. Hence, a proper guess for a stable model of LP on D refuting an atom B can be verified in polynomial time with an NP oracle. As a consequence, the queries in **LP(LP)** are in Π_2^p.

	Expressiveness		Expression Complexity
Language	*(data) complexity*	*order*	
LP(LP), LP(Mon), CI(LP), CI(Mon), CRF(LP), CRF(Mon), 2Pos(LP), SStrat(LP)	Π_2^p		$E\Pi_2^p = \mathrm{NEXP}^{\mathrm{NP}}$
2Pos(Mon)	Δ_2^p	*	$E\Delta_2^P = \mathrm{EXP}^{\mathrm{NP}}$
SCI(LP), SCI(Mon), Strat(LP), Strat(Mon), RF(LP), RF(Mon), SStrat(Mon)	Θ_2^p	*	$E\Theta_2^P = \mathrm{PSPACE}^{\mathrm{NP}}$
LP, Pos(LP)	coNP		coNEXP
Pos(Mon)	monotone coNP		coNEXP
LP^k	Π_k^P		$E\Pi_k^P$
LP*	PH		EXPSPACE

Table 2: Expressiveness of MLP under $\mathcal{M}^{st}$ over finite structures

For the other direction, consider an arbitrary Π_2^p computable $n-$ary relation A over signature $\tau = (\mathbf{Q})$. Let $\tau' = (\mathbf{Q}, B)$ be the extension of τ by a new relation symbol B that has the same arity as A. Then, the Boolean query on τ' which is true on a τ'-structure D iff $D|B \supseteq A(D|\mathbf{Q})$, is in Π_2^p. (Here $D|\mathbf{R}$ is the restriction of structure D to the relation symbols in $\mathbf{R}$.)

Hence, by the extension of Fagin's theorem [48], there exists a second-order sentence Φ over τ' of the form

$$\forall R \exists S.\varphi$$

such that for every τ'-structure D, $D \models \Phi$ if and only if $D|B \supseteq A(D|\mathbf{Q})$. Moreover, deciding $(D, R) \models \forall S \neg\varphi$ is in co-NP. By Theorem 6.3, there exists an *LP* module $P_S[\mathbf{Q}, B, R]$ which performs this test.

Consider now the following program LP_Ψ:

$$\begin{aligned} B(\mathbf{x}) &\leftarrow \neg B^-(\mathbf{x}) \\ B^-(\mathbf{x}) &\leftarrow \neg B(\mathbf{x}) \\ R(\mathbf{y}) &\leftarrow \neg R^-(\mathbf{y}) \\ R^-(\mathbf{y}) &\leftarrow \neg R(\mathbf{y}) \\ \perp &\leftarrow P_S[\mathbf{Q}, B, R] \\ A(\mathbf{x}) &\leftarrow B(\mathbf{x}) \end{aligned}$$

The first two clauses select a relation B, and the following two clauses select a relation R. The denial clause checks whether the τ'-structure D given by the relations for $\mathbf{Q}$ and the chosen relation B, extended by R fulfills $(D, R) \not\models \forall S \neg\varphi$, i.e., whether $(D, R) \models \exists S\varphi$. The last clause serves to transfer the value of B to the output relation A.

If the choice of B and R leads to a stable model, and only in this case, B is a superset of $A(D|\mathbf{Q})$. Hence, the cautious stable model semantics on that program, which takes the intersection of all stable models, computes precisely the relation A. ∎

Remark. The previous theorem shows that modular logic programming has the same expressive power as disjunctive logic programming over finite structures [14]. Thus, it appears that modular logic programming can be translated into disjunctive logic programming and vice versa. However, such translations are only available as long as finite structures are concerned. As shown recently [12], over infinite Herbrand models resp. the integers, DLP under the stable semantics can express precisely all Π^1_1 definable relations, and has the same expressive power as normal logic programming [41]. The proof of Theorem 6.4 works over infinite Herbrand models resp. the integers as well and establishes that MLP can express all Π^1_2 definable relations. Since, as well-known, there are Π^1_2 definable relations on these domains which are not Π^1_1 definable, it follows that MLP is there strictly more expressive than DLP.

The following corollaries demonstrate that either ordinary or call recursion can be restricted, without losing expressibility:

Corollary 6.5 **CRF(LP)** *and* **CI(LP)** *capture* Π^p_2 *over finite structures.*

Proof. Follows from the fact that LP_Ψ in the proof of Theorem 6.4 is call recursion free and call independent. ∎

Corollary 6.6 **SStrat(LP)** *captures* Π^p_2 *over finite structures.*

Proof. It suffices to show that the complement Quantifier $C_\sim$ (of arbitrary arity k) can be expressed in **SStrat(LP)**: To this end, we use additional modules N^k:

$$\begin{array}{l} N^k[S] \quad : \\ \quad N^k(\mathbf{x}) \leftarrow \neg S(\mathbf{x}) \end{array}$$

(The existence of such a module follows immediately from Proposition 6.3 as well.) ∎

Now we turn to **RF(LP)** and **SCI(LP)**.

Theorem 6.7 **SCI(LP)** *captures* Θ^p_2 *over all ordered finite structures.*

Proof. For membership of **SCI(LP)** in Θ^p_2, observe that the oracle queries for evaluating the subprograms can be made in parallel, since they are independent of each other by assumption. The computation of the (unique) stratified model of the main program is after that is possible in polynomial time. Hence, it lies in $P^{NP}_{||}$. Since by Theorem 6.1 $P^{NP}_{||} = \Theta^p_2$, it follows that query evaluation is in Θ^p_2.

For expressiveness, we make use of the *Stewart Normal Form*, introduced in [43, 44], and generalized in [23]. It states that, on ordered finite structures, every Θ^p_2 relation is definable by an existential first order formula with generalized quantifiers of the following form:

$$\exists \mathbf{x}. P_1[\mathbf{Q}](\mathbf{x},\mathbf{y}) \wedge P_2[\mathbf{Q}](\mathbf{x},\mathbf{y})$$

where P_1 (resp. P_2) is an co-NP-computable (NP-computable) generalized quantifier. Concluding again from Proposition 6.3, there exist modules computing P_1 and the complement of P_2. Hence, the following main program computes the relation:

$$A(\mathbf{y}) \leftarrow P_{P1}[\mathbf{Q}](\mathbf{x},\mathbf{y}), \neg P_{P2}[\mathbf{Q}](\mathbf{x},\mathbf{y}) \tag{9}$$

■

We say that a language $C(D)$ obeys the Stewart Normal Form, if the main programs in $C(D)$ can be written in the form (9). Surprisingly, the expressive power does not increase if we move from **SCI(LP)** to **Strat(LP)**.

Theorem 6.8 *Over ordered finite structures,* **SCI(LP)** $=$ **Strat(LP)** $=$ **RF(LP)**.

Proof. Every stratified program P can be partitioned into a finite number k of strata $S_1, \ldots, S_k$. Each stratum S_i can be considered as a **SCI(LP)** program over the predicates of the lower strata S_j, $j < i$, and thus can be evaluated by a Θ_2^p machine M_i. Hence, the evaluation of the program can be done by the composition $M_1 \circ \cdots \circ M_k$ of the machines M_i. By Lemma 6.2, this composed computation is in Θ_2^p. This proves that all mappings in **Strat(LP)** are in Θ_2^p. Expressiveness for Θ_2^p over ordered databases follows from Theorem 6.7.

The result for **RF(LP)** follows immediately from the result about **Strat(LP)** and the fact that the program in the proof of Theorem 6.7 is actually also from this class. ■

Corollary 6.9 **SCI(LP), Strat(LP)**, *and* **RF(LP)** *obey the Stewart Normal Form.*

6.4 Expressive power with monotone modules

From Theorem 3.11, we can also infer that a positive program using only monotone modules that compute coNP-relations does itself only compute a coNP-relation; in other words, glueing monotone co-NP-modules together with a positive logic program does not lead to an increase in expressibility.

Theorem 6.10 **Pos(Mon)** $=$ **Mon**, *i.e.* **Pos(Mon)** *captures the class of all monotone co*NP *queries.*

Proof. Monotonicity of **Pos(Mon)** queries is easy to see from the fixpoint characterization of the unique stable model (Theorem 3.11).

co-NP-computability holds for the following reason. A fact $Q(t)$ is computed true by a **Pos(Mon)** program P, if and only if for every sequence S^η such that $S^\eta \supseteq T_P^\eta$ it holds that $Q(t)$ is contained in $S^{\eta*}$, where $\eta*$ is the least fixpoint ordinal. Over finite databases, this is equivalent to membership of $Q(t)$ in S^m, where m is the size of the Herbrand base of P.

To prove that $Q(t)$ is not computed true, it is sufficient to exhibit a sequence S_i, $i = 0, \ldots, m$ such that $S_{i+1} \supseteq T_P(S_i, P)$ and $Q(t)$ is not contained in S_m; on the other hand, the existence of such a sequence is clearly also necessary.

The fact $S_{i+1} \supseteq T_P(S_i, P)$ has a polynomial size proof, given by any superset $S'_i \supseteq \mathsf{comp}(S_i, P)$ such that $T_P(S'_i) \subseteq S_{i+1}$. The set S'_i in turn has a polynomial size proof as well, since the fact that a call atom L is false in $\mathsf{comp}(S_i, P)$ has a polynomial size proof (as deciding this is in NP). Hence, the fact $S_{i+1} \supseteq T_P(S_i, P)$ has a polynomial size proof that can be checked in polynomial time, and the same holds for the sequence $S_1, \ldots, S_m$.

This shows that deciding whether $Q(t)$ is not computed true is in NP. From this, co-NP-computability of the query follows. ■

Note that telling whether a given program P belongs to **Pos(Mon)** is undecidable (since deciding whether an lp module LP is monotone is undecidable).

Before we proceed, we need the following Lemma:

Lemma 6.11 *There exists a subclass* **R** *of* (**CI** ∩ **RF**)*, such that* **R(Mon)** = **LP**.

Proof. An immediate consequence of the fact that there exist monotone problems which are NP complete under quantifier-free reductions with successor [46, 47]. ■

Theorem 6.12 *Over ordered finite structures,* **Strat(Mon)**, **SStrat(Mon)**, *and* **RF(Mon)** *capture* Θ_2^p.

Proof. With monotone modules, all yellow edges become "monotone", and thus all cycles in the given program classes become monotone, and therefore by Theorem 6.10 can be replaced by single calls to a monotone module. Collapsing the cycles into single calls we obtain a **RF(Mon)** program. Since **RF**(**LP**) is in Θ_2^p, membership follows.

For expressiveness, we use the Stewart Normal Form again, however we are now confined to monotone modules. Thus, we take the program (9) from the proof of Theorem 6.7 and define the modules in **RF(Mon)** which is possible according to Lemma 6.11. Since the resulting program is in **RF(RF(Mon))**, which amounts to **RF(Mon)**, the result follows. ■

Corollary 6.13 **LP(Mon)**, **CRF(Mon)**, *and* **CI(Mon)** *capture* Π_2^p.

Proof. Membership follows from Theorem 6.4. For expressiveness, we extend the idea of the proof of Theorem 6.12. Thus, we employ the program LP_Ψ introduced in Theorem 6.4, and provide the **LP** modules by Lemma 6.11.

The only obstacle remaining is that the proof of Lemma 6.11 requires ordered domains. Consider the following program LP_{succ}:

$$\begin{aligned}
S(x,y) &\leftarrow \neg S^-(x,y) \\
S^-(x,y) &\leftarrow \neg S(x,y) \\
L(x,y) &\leftarrow \neg L^-(x,y) \\
L^-(x,y) &\leftarrow \neg L(x,y) \\
L(x,z) &\leftarrow L(x,y), L(x,z) \\
\perp &\leftarrow L(x,x) \\
\perp &\leftarrow L^-(x,y), L^-(y,x), x \neq y
\end{aligned}$$

$$S^-(x,y) \leftarrow L^-(x,y)$$
$$S^-(x,z) \leftarrow L(x,y), L(y,z)$$

In every stable model of LP_{succ}, S defines a successor relation on the domain, and thus, every stable model of $LP_{\text{succ}} \cup LP_{\Psi}$ is ordered by S. Since the complexity of **LP(RF(LP))** is in $\text{coNP}^{\Theta_2^p} = \Pi_2^p$, the result follows. ■

6.5 2Pos(*) programs

Table 2 also includes results for classes **2Pos(*)** of MLPs. These classes are natural generalizations of the respective **Pos(*)** class, by allowing that a module may be based either on the skeptical or the brave variant of stable model semantics (i.e., inference from either the intersection or the union of all stable models). This reflects programming with modules under different calling (or reasoning) modes.

Syntactically, we distinguish the calling mode to a module by prefixing a call-predicate with either "$\Box$" or "$\Diamond$", where the former represents skeptical mode and the latter brave mode. Simlar reasoning modes are employed in epistemic logic programming [3].

Clearly, **2Pos(LP)** and **LP(LP)** have the same expressive power since negation can be provided by a module (cf. proof of Corollary 6.6). What happens with **2Pos(Mon)** ? The brave modality can be seen as a kind of double negation, similar to $\Diamond = \neg\Box\neg$ in modal logic. Therefore, brave calls to a monotone module are monotone; this means that **2Pos(Mon)** programs amount to positive LPs with monotone GQs, on which the least fixpoint semantics (cf. Section 3.3) can be applied. The expressiveness must be higher than co-NP, since both co-NP-hard and NP-hard modules can be employed. In fact, we obtain the following.

Theorem 6.14 2Pos(Mon) *captures Δ_2^p over ordered finite structures.*

Proof. Since the domain is ordered, we may without loss of generality suppose that constants $0, max$ are available. Membership in Δ_2^p follows from the fact that the modules are monotone by assumption, and therefore, ordinary inflationary evaluation of the main program is possible. Since the closure ordinal is polynomial in the domain size, a polynomial time algorithm with an NP oracle will suffice.

For expressiveness, we use [45, Corollary 4] which says that over ordered finite structures, every Δ_2^p relation $A(\mathbf{y})$ over $\tau = \mathbf{Q}$ is expressible as $R^*[\mathbf{Q}](\mathbf{max}, \mathbf{y})$, where $R^*[\mathbf{Q}]$ is the least fixed point of the monotone mapping

$$R \rightarrow \{\mathbf{x} \mid H_1[\mathbf{Q}, R](\mathbf{x}, \mathbf{y}) \vee H_2[\mathbf{Q}, R](\mathbf{x}, \mathbf{y})]\}$$

Here, $H_1[\mathbf{Q}, R]$ (resp. $H_2[\mathbf{Q}, R]$) is an NP-computable relation (co-NP computable relation) over signature $\tau \cup \{R\}$ which is monotone in R.

By Proposition 6.3, we conclude that there exist two modules P_{H1} and P_{H2} which compute H_1 and H_2, under brave and cautious semantics, respectively.

Now the main program becomes fairly easy:

$$R(\mathbf{x}, \mathbf{y}) \leftarrow \Box P_{H1}[\mathbf{Q}, R](\mathbf{x}, \mathbf{y})$$

$$R(\mathbf{x},\mathbf{y}) \leftarrow \Diamond P_{H2}[\mathbf{Q},R](\mathbf{x},\mathbf{y})$$
$$A(\mathbf{y}) \leftarrow R(\mathbf{max},\mathbf{y})$$

■

6.6 Nested modular logic programs

Not much surprisingly, by nesting modules the expressiveness of modular logic programs increases. For the class of all nested MLPs, $\mathbf{LP}^*$, and the class of MLPs with nesting depth bounded by k, we obtain the following result.

Theorem 6.15 $\mathbf{LP^k}$ *captures the class* Π^p_k *of PH, and* $\mathbf{LP}^*$ *captures PH.*

Proof. Analogous to Theorem 6.4. ■

Thus, modular logic programming is as expressive as full second-order logic over finite structures (cf. [48]).

Expressiveness results for restricted classes of nested MLPs can be derived in the same manner. However, the number of different classes grows exponentially with the nesting depth, and we therefore refrain from considering all of them here. Basically, the capturing results from Table 2 can be lifted all higher levels of PH by appropriate MLP classes. For example, **Strat(SStrat(LP))** captures Θ^p_3, and obeys the Stewart Normal Form in the main program. Note that **LP(Strat(LP))** collapses to **LP(LP)** since the logspace computations in the medium module can be transferred to the main program.

7 Further Issues and Conclusion

7.1 Summary

Modularity has been an important issue in logic programming for more than a decade now, and a large number of different approaches to programming with modules has been proposed. The dichotomy of requirements to be met by a programming language and by a logical system makes it hard to define a general framework for modular logic programming.

Among the large number of works on the subject, three main directions have been followed in modular logic programming. [2]

Programming-in-the-large: Composition of ordinary logic programs by means of (algebraic) extra-logical composition operators without extending the semantics of the underlying programs, e.g. in [35], [16].

Programming-in-the-small: Extension of logic languages by scoping and abstraction mechanisms, e.g. embedded implications in [32].

Programm-Splitting Introducing modularity by exploiting properties which are implicitly provided by non-modular logic programs, e.g. the relevance principle in [8, 9] or modular construction of models [26, 14].

[2] We borrow here from the classification and extensive bibliography in [5], and confine ourselves to some seminal papers.

In this paper, we have presented a new approach to modular logic programming, which is based on generalized quantifiers. We have shown how logic programs can be extended by incorporation of generalized quantifiers. We have defined a semantics for such programs which properly extends the semantics of Horn logic programs and of stratified logic programs. In this course, we have extended the concept of stratification. Furthermore, we have presented for unstratified programs a conservative extension of the concept of stable model, and thus provide a semantics for arbitrary logic programs using modules. Based on syntactic and semantics criteria, we have presented a taxonomy of modular program classes, whose property have been investigated. In particular, the expressive capability of these classes over finite structures have been thoroughly studied.

Our approach to modular programming, which is based on generalized quantifiers, is quite flexible and allows for easy integration of abstractly defined predicates into the language. It aims at combining Programming-in-the-large and Programming-in-the-small:

- The semantics of the modules can be defined as for ordinary logic programs (with the possibility to go beyond that), yet the framework still retains declarativity and logical purity even for the main program.
- The semantics of the main programs is a conservative extension of standard logic programming semantics which avoids the difficulties (to the real-life programmer) of Kripke semantics for intuitionistic and modal logic.
- The semantic coincidence of program classes, like $\mathbf{Pos}(\mathbf{Pos}(..(\mathbf{Pos}(\mathbf{Mon}))..)) = \mathbf{Mon}$, introduces modularity without increasing the expressivity, i.e., without increasing the evaluation complexity, yet it provides comfortable features for information hiding and scoping.

7.2 Related Work

Briefly, our approach compares to other approaches as follows. An important difference to O'Keefe's concept [35] and to Gaifman's and Shapiro's work [17] is that our approach is *extensional*, in that the main program does not operate on the clauses of the subprograms (O'Keefe's union operator takes the union of two sets of clauses) but refers to a subprogram as an independent semantical entity; we feel that this view fulfills the needs of real-life programming and real-life programmers much better, in particular it facilitates the reuse of modules as well as structured program development in large scale and inhomogeneous projects in general.

The extensional view of Fitting [16] is the closest among the semantics we encountered, yet Fitting is rather concerned with recursion theoretic questions and refrains from using logic main programs; instead his approach allows to "do logic programming in a functional programming style" [16]. Still, our expressiveness proofs suggest that subprograms are in a natural correspondence to first principles from recursion and complexity theory.

In our approach, main programs and subprograms are syntactically and semantically clearly distinguished; thus, compilation and partial evaluation of subprograms can be adressed quite easily, as postulated in [32]. It is evident that our semantics can be adapted to the three-valued case, and thus to well-founded and partial stable semantics.

An extension of logic programming by considering literals in rule bodies whose truth is determined by inference from a set of interpretations has been undertaken by Gelfond, who considered *epistemic logic programs* ([19]; see [20, 3]). In such programs, a literal L in a rule body may be preceded by an epistemic operator K or M; intuitively, KL means that L is known, and ML that L is believed. For example,

$$innocent(X) \leftarrow \neg K guilty(X)$$

states that if X is not known to be guilty, we infer that (s)he is innocent. Evaluation of a "subjective" literal KL resp. ML basically refers to inference of L from a set of interpretations, rather than to membership of L in an interpretation; the semantics of epistemic logic programs is defined in terms of world views, which are sets of interpretations (to be precise, sets of answer sets; see [3] for details).

While epistemic logic programming allows to speak about provability of literals in rules, the concern is knowledge representation rather than modular programming; in particular, evaluation of a subjective literal refers always to the whole program, and modular entities which allow reuse of programs are not available. Thus, epistemic logic programming is rather weakly related to our approach of modular logic programming.

7.3 Further work

It appears that our approach to modular logic program rises a number of interesting issues for further research.

- Revision logic programming as introduced in [30, 31, 39] can be easily incorporated in subprograms [15]; thus a logic program can do counterfactual reasoning about logic programs.
- Since disjunctive logic programming is as expressive as modular logic programming over finite domains, the feasibility of knowledge representation by modular logic programs can be investigated.
- Similarly, abduction from logic programs [13] can be incorporated into our framework.
- Provide a proof theory for full modular logic programs.
- Identify fragments of modular logic programming which have a zero-one law like datalog.
- Relate our approach to the other proposed languages for modular logic programming, in particular to embedded implications. In particular, find a modular translation of modular logic programming into embedded implications and vice versa.
- Finally, we think that existing implementations can be adapted to modularity comparatively easy with our approach.

References

[1] K. Apt, H. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In Minker [33], pages 89–148.

[2] A. Badia, M. Gyssens, and D. V. Gucht. Query languages with generalized quantifiers. In R.Ramakrishnan, editor, *Applications of Logic Databases*, pages 235–258. Kluwer Academic Publishers, 1995.

[3] C. Baral and M. Gelfond. Logic Programming and Knowledge Representation. *Journal of Logic Programming*, 19/20:73–148, 1994.

[4] J. Barwise and R. Cooper. Generalized Quantifiers and Natural Language. *Linguistic and Philosophy*, 4:159–219, 1981.

[5] M. Bugliesi, E. Lamma, and P. Mello. Modularity in Logic Programming. *Journal of Logic Programming*, 19/20:443–502, 1994.

[6] A. Dawar. Generalized Quantifiers and Logical Reducibilities. *Journal of Logic and Computation*, 5(2):213–226, 1995.

[7] A. Dawar, G. Gottlob, and L. Hella. Capturing Relativized Complexity Classes without Order. Technical Report CD-TR 96/105, Christian Doppler Laboratory for Expert Systems, TU Vienna, September 1996.

[8] J. Dix. A Classification Theory of Semantics of Normal Logic Programs: Strong Properties. *Fundamenta Informaticae*, 22:227–255, 1995.

[9] J. Dix. A Classification Theory of Semantics of Normal Logic Programs: Weak Properties. *Fundamenta Informaticae*, 22:257–288, 1995.

[10] J. Dix. Semantics of Logic Programs: Their Intuitions and Formal Properties. An Overview. In *Logic, Action and Information. Proc. of the Konstanz Colloquium in Logic and Information (LogIn'92)*, pages 241–329. DeGruyter, 1995.

[11] F. Donini, D. Nardi, and R. Rosati. Ground Nonmonotonic Modal Logics for Knowledge Representation. In *Proceedings World Congress for AI (WOCFAI-95)*, 1995.

[12] T. Eiter and G. Gottlob. Expressiveness of Stable Model Semantics for Disjunctive Logic Programs with Functions. Technical Report CD-TR 96/103, Christian Doppler Laboratory for Expert Systems, TU Vienna, Austria, July 1996. Submitted fo publication.

[13] T. Eiter, G. Gottlob, and N. Leone. Abduction From Logic Programs: Semantics and Complexity. Technical Report CD-TR 94/72, Christian Doppler Laboratory for Expert Systems, TU Vienna, October 1994. *Theoretical Computer Science*, to appear.

[14] T. Eiter, G. Gottlob, and H. Mannila. Adding Disjunction to Datalog. In *Proceedings of the Thirteenth ACM SIGACT SIGMOD-SIGART Symposium on Principles of Database Systems (PODS-94)*, pages 267–278, May 1994.

[15] T. Eiter, G. Gottlob, and H. Veith. Logic Programming: Modularity and Revision. In *Abstract presented at the Workshop* Logic Databases: The Meaning of Change*, Dagstuhl, September 1996*, 1996.

[16] M. Fitting. Enumeration Operators and Modular Logic Programming. *Journal of Logic Programming*, 4:11–21, 1987.

[17] H. Gaifman and E. Shapiro. Fully Abstract Compositional Semantics for Logic Programs. In *Proceedings 16th Annual ACM Symposium on Principles of Programming Languages*, pages 134–142. ACM Press, New York, 1989.

[18] M. Garey and D. S. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979.

[19] M. Gelfond. Epistemic Approach to Formalization of Commonsense Reasoning. Technical Report TR-91-2, University of Texas at El Paso, 1994.

[20] M. Gelfond. Logic Programming and Reasoning with Incomplete Information. *Annals of Mathematics and Artificial Intelligence*, 12:89–116, 1994.

[21] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Logic Programming: Proceedings Fifth Intl Conference and Symposium*, pages 1070–1080, Cambridge, Mass., 1988. MIT Press.

[22] L. Giordano and A. Martelli. Structuring Logic Programs: A Modal Approach. *Journal of Logic Programming*, 21:59–94, 1994.

[23] G. Gottlob. Relativized Logspace and Generalized Quantifiers over Finite Structures. In *Proceedings of the Tenth Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 65–78, 1995. Extended version to appear in *Journal of Symbolic Logic*.

[24] L. Henkin. Some Remarks on Infinitely Long Formulas. In *Infinitistic Methods, Proc. Symp. on Foundations of Mathematics*, pages 167–183. Warsaw, Panstwowe Wydawnictwo Naukowe and Pergamon Press, 1961.

[25] P. Hsu and D. Parker. Improving SQL with Generalized Quantifiers. In *Proceedings 11th Intl Conference on Data Engineering*, 1995.

[26] V. Lifschitz and H. Turner. Splitting a Logic Program. In *Proceedings ICLP-94*, pages 23–38, Santa Margherita Ligure, Italy, June 1994. MIT-Press.

[27] P. Lindström. First Order Predicate Logic with Generalized Quantifiers. *Theoria*, 32:186–195, 1966.

[28] J. Lloyd. *Foundations of Logic Programming*. Springer, Berlin, 1984.

[29] P. Mancarella and D. Pedreschi. An Algebra of Logic Programs. In *Proceedings ICLP '88*, pages 1006–1023. MIT-Press, 1988.

[30] W. Marek and M. Truszczyński. Revision specifications by means of revision programs. In *Logics in AI. Proceedings JELIA '94*, LNAI, 1994.

[31] W. Marek and M. Truszczyński. Revision Programming, database updates and integrity constraints. In *Proceedings ICDT '95*, volume 893 of *LNCS*, pages 368–382, 1995.

[32] D. Miller. A Theory of Modules in Logic Programming. In *Proceedings of the International Logic Programming Symposium (ILPS '86)*, pages 106–114. MIT Press, 1986.

[33] J. Minker, editor. *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufman, Washington DC, 1988.

[34] A. Mostowski. On a Generalization of Quantifiers. *Fundamenta Mathematicae*, 44:12–36, 1957.

[35] R. O'Keefe. Towards an Algebra for Constructing Logic Programs. In *Proceedings of the IEEE Symposium on Logic Programming*, pages 152–160. IEEE CS Press, 1985.

[36] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[37] T. Przymusinski. On the Declarative and Procedural Semantics of Stratified Deductive Databases. In Minker [33], pages 193–216.

[38] T. Przymusinski. Stable Semantics for Disjunctive Programs. *New Generation Computing*, 9:401–424, 1991.

[39] T. Przymusiński and H. Turner. Update by means of inference rules. In *Proceedings LPNMR '95*, volume 928 of *LNCS*, pages 156–174, 1995.

[40] D. Saccá. Deterministic and Nondeterministic Stable Model Semantics for Unbound DATALOG Queries. In *Proceedings 5th Intl. Conference on Database Theory (ICDT-95)*, LNCS 893, pages 353–367, January 1995.

[41] J. Schlipf. The Expressive Powers of Logic Programming Semantics. *Journal of Computer and System Sciences*, 51(1):64–86, 1995. Abstract in Proc. PODS 90, pp. 196–204.

[42] G. Schwarz and M. Truszczyński. Minimal Knowledge Problem: A New Approach. *Artificial Intelligence*, 67:113–141, 1994.

[43] I. Stewart. Logical Characterizations of Bounded Query Classes I: Logspace oracle machines. *Fundamenta Informaticae*, 18:65–92, 1993.

[44] I. Stewart. Logical Characterizations of Bounded Query Classes II: Logspace oracle machines. *Fundamenta Informaticae*, 18:93–105, 1993.

[45] I. Stewart. Incorporating generalized quantifiers and the least fixed point operator. In *Proc. Computer Science Logic*, number 832 in LNCS, pages 318–333, 1994.

[46] I. Stewart. Complete problems for monotone NP. *Theoretical Computer Science*, 145:147–157, 1995.

[47] I. Stewart. Logical Descriptions of monotone NP problems. *Journal of Logic and Computation*, 4:337–357, 1995.

[48] L. J. Stockmeyer. The Polynomial-Time Hierarchy. *Theoretical Computer Science*, 3:1–22, 1977.

[49] A. van Gelder, K. Ross, and J. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.

[50] M. Vardi. Complexity of relational query languages. In *Proceedings 14th STOC*, pages 137–146, San Francisco, 1982.

[51] K. Wagner. Bounded Query Classes. *SIAM J. Comp.*, 19(5):833–846, 1990.

[52] J.-H. You and L. Yuan. A Three-Valued Semantics for Deductive Databases and Logic Programs. *Journal of Computer and System Sciences*, 49:334–361, 1994.