Modular Logic Programming and Generalized Quantifiers*

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

[+] AG Informatik, University of Giessen
Arndtstraße 2, D-35392 Gießen, Germany.
Email: eiter@informatik.uni-giessen.de
[*] Information Systems Department, TU Vienna
Paniglgasse 16, A-1040 Wien, Austria.
Email: (gottlob|veith)@dbai.tuwien.ac.at

Abstract. The research on systems of 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 this paper, we present a general model theoretic approach to modular logic programming 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. 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.
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 analyzed.

Keywords: generalized quantifiers, modular logic programming, stratification, stable models, expressive power

1 Introduction

Structured Logic Programming requires the possibility to define subprograms along with scoping and abstraction mechanisms. This need has initiated a large number of investigations, which evolved into the area of modular logic programming, e.g. [26, 29, 23, 14, 17, 4]; see [4] for a comprehensive survey.

The research on systems of logic program modules has followed two mainstreams [4]. The one is programming-in-the-large, where compositional operators are provided for combining separate and independent modules, e.g. [29, 23, 14].

* Extended Abstract, with most proofs omitted. An extended technical report including all proofs is available from the authors.

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 mechanisms available in other programming paradigms, e.g. [26, 17]. This approach fosters the introduction of new logical connectives in an extended logical language. Thus, the two mainstreams are quite divergent.

Both approaches suffer from certain problems, and a number of important issues in modular logic programming remains to be satisfactorily answered (cf. [4]). For example, as pointed out in [4, 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.

The following example is intended to give a first feeling of what a modular logic program with generalized quantifiers can look like. Exact definitions will be given in Sections 2–4.

Example 1. Consider the following program:

$$Col_ok \leftarrow 3Uncol[G_1], \neg 3Uncol[G_2]$$
$$Diff_TC(x,y) \leftarrow Col_ok, TC[G_2](x,y), \neg TC[G_1](x,y)$$

Here, $3Uncol[G]$ and $TC[G]$ are calls to program modules $3Uncol$ and TC which intuitively decide uncolorability of a graph G with three colors and compute the transitive closure of G, respectively. The use of these modules amounts to invoking generalized quantifiers for 3-uncolorability and for transitive closure, respectively.

Intuitively, the program should do the following: *If the graph G_1 is not 3-colorable and the graph G_2 is 3-colorable, then $Diff_TC$ is assigned the difference of the transitive closures of G_2 and G_1; otherwise, $Diff_TC$ is empty.* ■

Generalized quantifiers [28, 21] 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 its expressive capability. Essentially, a generalized quantifier checks properties of relations definable in the underlying logic (see Section 2 for details). Mostowski [28] gave a formalization of generalized quantifiers, which served as a basis for the seminal work of Lindström [21].

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 [19], which have been studied extensively in logic. Generalized quantifiers have applications in various areas including linguistics, finite model theory, Artificial Intelligence and databases, cf. [3, 5, 2, 20].

For this reason, we believe that this extension of logic programming has high potential for applications beyond modular logic programming, which originally has driven this research on generalized quantifiers; in this paper, we introduce the technical framework of logic programming with generalized quantifiers, and develop modular logic programming as an application thereof. Epistemological and linguistic issues, which are intriguing and deserve further investigation, are beyond the scope of the present paper and remain for further work.

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 of generalized quantifier, only well-known concepts from logic programming are used.
- It conservatively 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 our modular LP approach is well understood.
- It supports information hiding. This feature is naturally provided by the scope of a generalized quantifier.

After introducing the main concepts 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 generalized 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 is 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.

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

2 Preliminaries

We assume that the reader is familiar with the concepts and terminology of logic programming; see [22] for background. P denotes a logic program, letters $Q, R, \ldots$ denote predicates, lower case letters x, y, z variables, $a, b, c, \ldots$ constants and $f, g \ldots$ functions. The bold face version $\mathbf{R}$ of a predicate symbol R denotes a list $R_1, \ldots, R_m$ 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.

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

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

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 $\mathcal{M}_P(\mathcal{A}) = (A, \mathbf{f}, \mathbf{c}, \mathbf{S})$ 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 foresee 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.

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, 31].*

A number of other properties of semantics are desirable, e.g. relevance and modularity; see [9, 7, 8].

Generalized Quantifiers The extension of first-oder logic by generalized quantifiers (GQs) was studied in depth in the seminal work by Lindström [21]; particular GQs, e.g. Henkin quantifiers [19], have been considered earlier. Formally, 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.

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*

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

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 the remaining free variables $\mathbf{y}$ after a formula (2).

The semantics of Q_C is defined as follows. Consider the formula $\Theta(\mathbf{y})$ 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 $\varphi_i^{\mathcal{A},\mathbf{b}} = \{\mathbf{a} \mid \mathcal{A} \models \varphi_i[\mathbf{a}, \mathbf{b}]\}$.

Example 2. The following GQs are well-known; the signature σ has in 1–3 a single, unary relation, in 4 two unary relations, and in 5–6 two binary relations, respectively.

1. The *universal quantifier*: $C_\forall = \{(A, U) \mid A = U\}$.
2. The *majority quantifier*: $C_M = \{(A, U) \mid 2|U| > |A|\}$.
3. The *modularity quantifier*: $C_k = \{(A, U) \mid |U| = 0 (\mathrm{mod}\ k)\}$.
4. The *isomorphism quantifier*: $C_\cong = \{(A, E, F) \mid (A, E), (A, F)$ are isomorphic graphs$\}$.

Semantically, the quantifier $C_\forall$ amounts to the standard universal first-order quantifier "$\forall$"; any formula $\forall x.\varphi(x)$ where φ is quantifier-free, is equivalent to $C_\forall x[\varphi(x)]$. Thus, consider a first order formula $\forall x \exists y R(x, y)$. Using the universal quantifier $\mathcal{C}_\forall$, we can write this formula as $C_\forall x[\exists y R(x, y)]$, and thus as

$$C_\forall x[\neg C_\forall y[\neg R(x, y)]].$$

Clearly, quantifier-free logic extended with $C_\forall$ yields first order logic. ■

Since the above definition of GQs has only relational signatures, no parameters like constants can be accessed for the semantical definition of a GQ. However, this is no shortcoming since such parameters can be provided by designated relations. This is exemplified by the next GQs.

Example 3. Consider the following important GQs, where σ consists of two binary and two unary relations, respectively.

5. The *transitive closure quantifier*: $C_{TC} = \{(A, E, \{(u, v)\}) \mid$ there is a path from u to v in the graph $E \subseteq A \times A\}$.

6. The *complement quantifier*: $C_\sim = \{(A, U, \{v\}) \mid v \in A - U\}$ ■

Here, the last relation in a structure from the class holds a single tuple, which amounts to the desired parameter. In a formula, the parameter can be assigned to the last relation via the equality predicate.

For example, the formula $\Xi = \forall xy.x \neq y \supset C_{TC}uvwz[E(u,v) \vee E(v,u), (w = x \wedge z = y)](x,y)$ expresses that a graph is weakly connected. Indeed, observe that Ξ is true on (A, E) iff there is an undirected path between every distinct pair of vertices x and y in the graph.

We make explicit use of this technique in the definition (3) of a GQ-atom in Section 3.1.

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 [16] 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 $a_1, \ldots, a_n, a_{n+1} = l$, and that $S_1, \ldots, S_n$ are predicates from the signature of an ordinary logic program P. Like in the definition of C_{TC}, R_{n+1} is intended to be a singleton. Then, the formula

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

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 relation defined by the formula $\mathbf{x}_{n+1} = \mathbf{v}$ is the singleton relation $\{\mathbf{v}\}$; informally, this formula transfers $\mathbf{v}$ into the quantifier such that it becomes accessible in its evaluation. For example, the transitive closure quantifier C_{TC} is used in a GQLP as a GQ atom $C_{TC}x_1x_2x_3x_4[G(x_1, x_2), x_3 = v_1 \wedge x_{=}v_2](v_1, v_2)$.

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

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

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

Syntactically, a logic program with generalized quantifiers can be regarded 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 called 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 4. Consider the following program P, which uses the isomorphism GQ $C_{\cong}$ and the transitive closure GQ C_{TC}. Its purpose is to compute, assuming some graph with edges E is provided, the atom Iso true if the vertices a and b lie in isomorphic strongly connected components of the graph:

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

Here, $\tau = (S, E, G_a, G_b, Iso)$, and $\tau^*_{\{C_{TC}, C_{\cong}\}}$ in addition contains the relation symbols $C_{TC}[E], C_{TC}[S], C_{TC}[G_a]$, and $C_{TC}[G_b]$, as well as the relation symbols $C_{\cong}[X,Y]$ for all $X, Y \in \{S, E, G_a, G_b\}$.

The first clause checks whether x and y lie in the same strongly connected component of the graph. The next two clauses determine in G_a and G_b the strongly connected components of a and b, respectively. The last clause assigns Iso value true if G_a and G_b are isomorphic, which is tested by using the isomorphism quantifier. Notice that graph isomorphism is not known to be a problem in co-NP. As a consequence, a normal logic program which decides this problem under standard cautious stable semantics is hard to find (and might not exist). ■

3.2 Semantics

Comparing logic programming to first-order logic, it is immediately clear that new 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, and thus a generally accepted semantics for non-stratified GQLPs may not be expected.

We start by extending the concept of stratification to 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 GQLPs 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 GQLPs is as follows. For any GQLP P over a structure $\mathcal{A}$ for its signature τ, $\mathcal{M}_P^*$ 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 is seen as an ordinary logic program and evaluated on the structure $\mathcal{A}$ extended by relations for the GQ-literals of P w.r.t. *$\mathcal{B}$, yields the structure $\mathcal{B}$. (Here, 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 (extensional) input relations.

Example 5. Reconsider the program P in Example 4. Suppose E is regarded as an extensional input relation $E = \{(a,b),(a,c),(c,a),(b,b)\}$, 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:

$$C_{TC}[E] = \{(a,b),(a,c),(a,a),(c,a),(c,c),(c,b),(b,b)\},$$
$$C_{\cong}[G_a, G_b] = \mathit{true} \text{ (formally, } = \{()\})$$

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 as 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.*

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 as above. (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 4 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.

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 [16] 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, is 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 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.,*

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

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

Proposition 3.3 *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 3.4 *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}. ■

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

Proposition 3.5 *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).*

3.3 Monotone Generalized Quantifiers

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 $\mathcal{B}$. 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.

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

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

Lemma 3.6 *For every P from* **Pos(Mon)**, *$T_P(\cdot)$ is a monotone operator.*

Corollary 3.7 *Let P be a program from* **Pos(Mon)**. *Then, T_P has a least fixpoint T_P^∞, given by the limit of the sequence T_P^α for ordinals $\alpha \geq 0$, where*

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

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 $C_\forall$: $A(0).$; $A(f(x)) \leftarrow A(x)$, ; $B \leftarrow C_\forall[A]$. Suppose we consider Herbrand interpretations. Then,

$$\begin{aligned} T_P^1 &= \{A(0)\}, \\ T_P^2 &= \{A(0), A(f(0))\}, \\ &\vdots \\ T_P^i &= \{A(f^j(0)) \mid 0 \leq j < i\} \\ &\vdots \\ T_P^\omega &= \{Q(f^i(0)) \mid 0 \leq i\}, \\ T_P^{\omega+1} &= T_P^\omega \cup \{B\} = T_P^\infty. \end{aligned}$$

■

In case of ordinary logic programming, the least fixpoint T_P^∞ of a positive program P, which is given by T_P^ω, yields the unique stable model. A similar property holds on programs from **Pos(Mon)**.

Theorem 3.8 *Let P be 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^∞.*

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).

4 Modular Logic Programs

Adapted to the logic programming paradigm, a program 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) 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.

Consequently, a module $LP[\mathbf{I}; O_1]$ can be abstractly seen as a GQ $Q_{LP}[\mathbf{I}](\mathbf{x})$, where the length of $\mathbf{x}$ is the arity of O_1. 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 an 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.

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 $\mathbf{Q}$ 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*).*

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.

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. Let us return to the program from Example 1:

$$Col_ok \leftarrow 3Uncol[G_1], \neg 3Uncol[G_2]$$
$$Diff_TC(x,y) \leftarrow Col_ok, TC[G_2](x,y), \neg TC[G_1](x,y)$$

Now, $3Uncol[G]$ and $TC[G]$ are defined by modules which compute their specific properties. The body of $TC[G]$ resp. $3Uncol[G]$ could be as follows.

$TC[G]$:
$$TC(x,y) \leftarrow G(x,y)$$
$$TC(x,y) \leftarrow TC(x,z), G(z,y)$$

$3Uncol[G]$:
$$Green(x) \leftarrow \neg Red(x), \neg Blue(x)$$
$$Red(x) \leftarrow \neg Green(x), \neg Blue(x)$$
$$Blue(x) \leftarrow \neg Red(x), \neg Green(x)$$
$$3Uncol \leftarrow G(x,y), Red(x), Red(y)$$
$$3Uncol \leftarrow G(x,y), Green(x), Green(y)$$
$$3Uncol \leftarrow G(x,y), Blue(x), Blue(y)$$

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

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 programs 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.*

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.

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:

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

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

Example 11. Consider again the module $TC[G]$, which is used in the following program P:

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

The first two rules assign R the undirect version of E. Thus, in any interpretation I, $TC[E]$ and $TC[R]$ hold the transitive closures of E and R, respectively. The next two clauses guarantee that no vertex in E has indegree greater than one and that each pair of vertices in E is weakly connected. The next three clauses check that there is some vertex in E having outdegree 0. These conditions amount to the property that E is a rooted tree.

The last three clauses select a subset of the vertices, such that they induce a subtree of E which has the same root. (Note that $TC[E](x,y)$ could be replaced by $E(x,y)$). Hence, given a graph E, the stable models correspond to those subtrees of E; if E is not a rooted tree, no stable model exists. ■

4.3 Nested LP modules

Modular programs as we have defined so far allow only one level of calls to subprograms. However, this restriction can be relieved in a straightforward way, provided a hierarchic use of modules is obeyed.

Define the *call graph* of a modular logic program P with modules $\mu_1, \ldots, \mu_k$ 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. 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

Properly distinguishing between ordinary predicates and GQ-predicates, we generalize the concept of dependency graph from ordinary LPs to GQLPs.

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:*

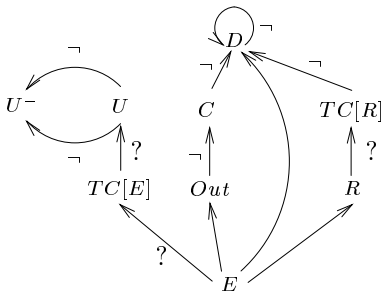
1. *there is a rule $P_1(\mathbf{t}_1) \leftarrow \ldots, \neg P_2(\mathbf{t}_2), \ldots$ in LP.*
2. *P_1 is a GQ predicate $P[\mathbf{Q}^*]$ and P_2 occurs in $\mathbf{Q}^*$.*
3. *there is a rule $P_1(\mathbf{t}_1) \leftarrow \ldots, P_2(\mathbf{t}_2), \ldots$ in LP;*

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.

Example 12. The dependency graph of the program in Example 11 is as follows (red arcs are symbolized by label "¬", 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 $\mathbf{Strat}(\mathbf{LP}(\mathbf{Strat}))$ 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.

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.

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.

We assume some familiarity with the basic concepts of complexity theory, in particular of the polynomial hierarchy (PH). Consult [15, 30] for an introduction. Intertwined with the levels of PH are the classes $\Theta_k^p = P_{\|}^{\Sigma_k^p}$ [37], where $\Sigma_k^p \subseteq \Theta_{k+1}^p \subseteq \Sigma_{k+1}^p$.

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

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 [34, 11, 33]; as shown in [34], **LP** captures coNP.

Table 2 compactly lists the expressivity results that we derive in the full paper, based on the stable model semantics $\mathcal{M}^{st}$. Here, **Mon** are the programs defining monotone modules (cf. Section 3.3), and **2PosMon(Mon)** are positive programs which may use monotone modules under cautious or brave stable semantics. 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 [6].

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

	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
$\mathbf{LP}^k$	Π_k^P		$E\Pi_k^P$
$\mathbf{LP}^*$	PH		EXPSPACE

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

same class C, then mutual such translations exist. Otherwise, this is not the case unless some strongly believed hypotheses of complexity theory are false.

The results in table 2 show that modular logic programming has the same expressive power as disjunctive logic programming over finite structures [11]. 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 [34]. The proof of the Theorem for MLP works over infinite Herbrand models resp. the integers as well and establishes that MLP can express all Π_2^1 definable relations. Since, as well-known, there are Π_2^1 definable relations on these domains which are not Π_1^1 definable, it follows that MLP is there strictly more expressive than DLP.

There is no space here for discussing how these expressiveness results are obtained. However, we sketch a proof for **SCI(LP)** directed to the reader familiar with the concepts, as it unveils further interesting syntactical properties of this fragment.

Theorem 6.1 SCI(LP) *captures Θ_2^p over all ordered finite structures.*

Proof. (Sketch) For membership of **SCI(LP)** in Θ_2^p, 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 possible in polynomial time. Hence, it lies in $P_{\|}^{NP}$. Since $P_{\|}^{NP} = \Theta_2^p$, query evaluation is in Θ_2^p.

For expressiveness, we make use of the *Stewart Normal Form*, introduced in [35, 36], and generalized in [18]. It states that, on ordered finite structures, every Θ_2^p 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. Since stable semantics of normal LPs expresses co-NP [34], there exist modules computing P_1 and the complement of P_2. Hence, the following main program computes the relation:

$$A(\mathbf{y}) \leftarrow P_{P1}[\mathbf{Q}](\mathbf{x}, \mathbf{y}), \neg P_{P2}[\mathbf{Q}](\mathbf{x}, \mathbf{y}) \tag{7}$$

■

Note that the proof shows that this fragment has a very simple normal form over finite structures: every program is equivalent to a single rule which uses two LP modules.

7 Further Issues and Conclusion

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 a conservative extension of the concept of stable model for unstratified programs, and thus we provide a semantics for arbitrary logic programs using modules. Based on syntactic and semantical criteria, we have presented a taxonomy of modular program classes, whose properties have been investigated. In particular, the expressive capability of these classes over finite structures have been thoroughly studied.

Our approach to modular programming 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 main programs avoids the difficulties (to the real-life programmer) of Kripke semantics for intuitionistic and modal logic.

A detailed comparison to related work will be given in the full paper.

A lot of interesting topics remains to be addressed; these issues are part of our current research.

- Revision logic programming as introduced in [24, 25, 32] can be easily incorporated in subprograms [13]; 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 [10] can be incorporated into our framework.
- Provide a proof theory for full modular programs.
- Relate our approach to the other proposed languages for modular logic programming, in particular to embedded implications.

Acknowledgement. The authors would like to thank the anonymous referees for their useful comments which helped to improve the presentation of this paper.

References

1. K. Apt, H. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In Minker [27], pp. 89–148.
2. A. Badia, M. Gyssens, and D. V. Gucht. Query languages with generalized quantifiers. In R.Ramakrishnan, editor, *Applications of Logic Databases*, pp. 235–258. Kluwer Academic Publishers, 1995.
3. J. Barwise and R. Cooper. Generalized Quantifiers and Natural Language. *Linguistic and Philosophy*, 4:159–219, 1981.
4. M. Bugliesi, E. Lamma, and P. Mello. Modularity in Logic Programming. *J. Logic Programming*, 19/20:443–502, 1994.
5. A. Dawar. Generalized Quantifiers and Logical Reducibilities. *J. Logic and Computation*, 5(2):213–226, 1995.
6. A. Dawar, G. Gottlob, and L. Hella. Capturing Relativized Complexity Classes without Order. *Mathematical Logic Quarterly*, to appear. Technical Report CD-TR 96/105, CD Lab for Expert Systems, TU Vienna, 1996.
7. J. Dix. A Classification Theory of Semantics of Normal Logic Programs: Strong Properties. *Fundamenta Informaticae*, 22:227–255, 1995.
8. J. Dix. A Classification Theory of Semantics of Normal Logic Programs: Weak Properties. *Fundamenta Informaticae*, 22:257–288, 1995.
9. J. Dix. Semantics of Logic Programs: Their Intuitions and Formal Properties. An Overview. In *Logic, Action and Information. Proc. Konstanz Colloquium in Logic and Information (LogIn'92)*, pp. 241–329. DeGruyter, 1995.
10. T. Eiter, G. Gottlob, and N. Leone. Abduction From Logic Programs: Semantics and Complexity. *Theoretical Computer Science*, to appear.
11. T. Eiter, G. Gottlob, and H. Mannila. Adding Disjunction to Datalog. In *Proc. PODS '94*, pp. 267–278, 1994.
12. T. Eiter and G. Gottlob. Expressiveness of Stable Model Semantics for Disjunctive Logic Programs with Functions. *J. Logic Programming*, to appear. CD-TR 96/103, CD Lab for Expert Systems, TU Vienna.
13. 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.*

14. H. Gaifman and E. Shapiro. Fully Abstract Compositional Semantics for Logic Programs. In *Proc. 16th ACM Symp. POPL*, pp. 134–142. 1989
15. M. Garey and D. S. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979.
16. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Proc. Fifth Intl CSLP*, pp. 1070–1080, 1988.
17. L. Giordano and A. Martelli. Structuring Logic Programs: A Modal Approach. *J. Logic Programming*, 21:59–94, 1994.
18. G. Gottlob. Relativized Logspace and Generalized Quantifiers over Finite Structures. In *Proc. IEEE LICS '95*, pages 65–78, 1995. Extended version *Journal of Symbolic Logic*, to appear.
19. L. Henkin. Some Remarks on Infinitely Long Formulas. In *Infinitistic Methods, Proc. Symp. on Foundations of Mathematics*, pp. 167–183. Warsaw, Panstwowe Wydawnictwo Naukowe and Pergamon Press, 1961.
20. P. Hsu and D. Parker. Improving SQL with Generalized Quantifiers. In *Proc. 11th Intl Conference on Data Engineering*, 1995.
21. P. Lindström. First Order Predicate Logic with Generalized Quantifiers. *Theoria*, 32:186–195, 1966.
22. J. Lloyd. *Foundations of Logic Programming*. Springer, Berlin, 1984.
23. P. Mancarella and D. Pedreschi. An Algebra of Logic Programs. In *Proc. ICLP '88*, pp. 1006–1023. MIT-Press, 1988.
24. W. Marek and M. Truszczyński. Revision specifications by means of revision programs. In *Logics in AI. Proc. JELIA '94*, LNAI, 1994.
25. W. Marek and M. Truszczyński. Revision Programming, database updates and integrity constraints. In *Proc. ICDT '95*, LNCS 893, pp. 368–382, 1995.
26. D. Miller. A Theory of Modules in Logic Programming. In *Proc. ILPS '86*, pp. 106–114. 1986.
27. J. Minker, editor. *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufman, 1988.
28. A. Mostowski. On a Generalization of Quantifiers. *Fundamenta Mathematicae*, 44:12–36, 1957.
29. R. O'Keefe. Towards an Algebra for Constructing Logic Programs. In *Proc. IEEE Symposium on Logic Programming*, pp. 152–160. IEEE CS Press, 1985.
30. C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
31. T. Przymusinski. On the Declarative and Procedural Semantics of Stratified Deductive Databases. In Minker [27], pp. 193–216.
32. T. Przymusiński and H. Turner. Update by Means of Inference Rules. In *Proc. LPNMR '95*, LNCS 928, pp. 156–174, 1995.
33. D. Saccá. Deterministic and Nondeterministic Stable Model Semantics for Unbound DATALOG Queries. In *Proc. ICDT-95*, LNCS 893, pp. 353–367, 1995.
34. J. Schlipf. The Expressive Powers of Logic Programming Semantics. *J. Computer and System Sciences*, 51(1):64–86, 1995. Abstract in Proc. PODS 90, pp. 196–204.
35. I. Stewart. Logical Characterizations of Bounded Query Classes I: Logspace oracle machines. *Fundamenta Informaticae*, 18:65–92, 1993.
36. I. Stewart. Logical Characterizations of Bounded Query Classes II: Logspace oracle machines. *Fundamenta Informaticae*, 18:93–105, 1993.
37. K. Wagner. Bounded Query Classes. *SIAM J. Comp.*, 19(5):833–846, 1990.

This article was processed using the LaTeX macro package with LLNCS style