Generalized Quantifiers in Logic Programs*

Thomas Eiter[1] Georg Gottlob[2] Helmut Veith[2,3]

[1] Institut für Informationssysteme, Abteilung Wissensbasierte Systeme, TU Wien, A-1040 Wien, Austria. Email: eiter@kr.tuwien.ac.at
[2] Institut für Informationssysteme, Abteilung Datenbanken und AI, TU Wien, A-1040 Wien, Austria. Email: (gottlob|veith)@dbai.tuwien.ac.at
[3] School of Computer Science, Carnegie Mellon University, Pittsburgh, USA. Email: veith@cs.cmu.edu

Abstract. Generalized quantifiers are an important concept in model-theoretic logic which has applications in different fields such as linguistics, philosophical logic and computer science. In this paper, we consider a novel application in the field of logic programming, which has been presented recently. The enhancement of logic programs by generalized quantifiers is a convenient tool for interfacing extra-logical functions and provides a natural framework for the definition of modular logic programs. We survey the expressive capability of syntactical classes of logic programs with generalized quantifiers over finite structures, and pay particular attention to modular logic programs. Moreover, we study the complexity of such programs. It appears that modular logic programming has the expressive power of second-order logic and captures the polynomial hierarchy, and different natural syntactical fragments capture the classes therein. The program complexity parallels the expressive power in the weak exponential hierarchy. Modular logic programming proves to be a rich formalism whose expressiveness and complexity can be controlled by efficiently recognizable syntactic restrictions.

1 Introduction

Generalized quantifiers are a well-known concept for enhancing the expressive capabilities of a logical language [44, 34, 7, 59]. They have been introduced since in many contexts, the standard quantifiers "for all individuals" and "for some individual" are not strong enough for a proper description of the state of affairs. Informally, a generalized quantifier $Q\mathbf{x}$ is a collection C of structures for a language $\mathcal{L}$, and $Q\mathbf{x}\varphi(\mathbf{x},\mathbf{y})$ evaluates on a structure $\mathfrak{A}$ to true for elements $\mathbf{y} \in A$, if the structure $(A, \{\mathbf{x} \in A \mid \mathfrak{A} \models \varphi(\mathbf{x},\mathbf{y})\})$ belongs to C.

In 1957, Mostowski [44] initiated the field, and stimulated a lot of work on generalized quantifiers during the 60's and early 70's. Another approach to generalized quantifiers from 1959 are Henkin quantifiers, which have been introduced

* This work was partially supported by the Austrian Science Fund Project N Z29-INF, and the Max Kade Foundation. Most of this research has been carried out while the first author was with the University of Gießen, Germany, and the third author was with TU Wien.

in [27] and subsequently studied in numerous papers, e.g. [18, 64, 5, 9, 33, 50]. Unifying the two approaches, the most prominent work from this area is the seminal paper of Lindström [34], whose formalization of generalized quantifiers is mostly used to date. After diminished attention for some time, generalized quantifiers gained in the 90's increasing interest in applications, from which new research issues emerge. The applications of generalized quantifiers range over different fields, including linguistics, philosophical logic and computer science.

In linguistics and philosophical logic, it has been recognized that the quantification theory of first-order logic is insufficient as a representation formalism for natural language statements, cf. [28]. Montague [43] made an attempt to properly treat quantifier phrases in English, the idea to which, as pointed out in [29], originally dates back to Russell [48]. Building on the work of Montague [43], Barwise and Cooper have presented an approach which adopts generalized quantifiers in a natural language formalization [6].

In computer science, generalized quantifiers are used in different areas. On the foundational side, they have been considered extensively in descriptive complexity theory as a tool for designing logical languages for capturing computational complexity classes. For example, restricted Henkin quantifiers capture NL [9] over ordered structures, while standard Henkin quantifiers capture Θ_2^p over ordered structures [22]; transitive closure logic captures NL over ordered structures, while deterministic transitive closure logic captures L over ordered structures [30, 25]; and, various extensions of first-order logic by quantifiers such as the Hamiltonian Path Quantifier capture NP or Θ_2^p (i.e., logspace with an NP oracle) over ordered structures, [53, 54, 22]. The relationship of oracles, generalized quantifiers and subprograms was also investigated in [38, 37, 40, 41]. See [31, 13] for a comprehensive overview. Notice that all syntactic complexity classes can be captured by generic generalized quantifiers [36, 39, 61].

In the context of databases, generalized quantifiers have been used as a means for increasing the expressive capability of database query languages. In [1], the transitive closure quantifier was applied ad hoc since connectivity cannot be expressed in Codd's relational calculus. A more systematic treatment is given in [3, 47], where a formalization of subqueries in SQL by generalized quantifiers is presented, and occurrences of generalized quantifiers in existing database query languages are discussed. Further applications of generalized quantifiers in computer science are in software engineering, where generalized quantifiers are used in the specifications of the Z and B notation [52, 65]. Moreover, generalized quantifiers have also been investigated in the context of fuzzy logic [26], which has become popular in computer science more recently.

In this paper, we address another application of generalized quantifiers in computer science; namely, in the field of logic programming. As pointed out in [16], enhancing logic programming by generalized quantifiers provides an elegant and appealing method for handling the following two problems. Firstly, the problem of interfacing external (non-logical) functions in a logic program, of which no other information than a semantical description is available. Such a situation is present e.g. with library routines, for which usually only a functional specification is known. Secondly, the problem of modular programming, where a formalism for the synthesis of program modules into logic programs is sought.

The two problems are not unrelated; the latter problem can be seen as an instance of the former, in which modules are functions for which information about the implementation in terms of proof theory etc. is available. For both problems, so far mainly ad hoc solutions have been proposed. Modular logic programming has been approached from two main directions [10]. The one direction is to build an algebra of programs, in which composition operators are available for combining two programs into a single one (cf. [19, 45]). The other direction resorts to using concepts of modal logic and embedded implications for defining the meaning of subgoals which access external functions (cf. [42]). Both approaches, however, have problems in handling negation, and typically, negation is disregarded for combined programs. This is a severe shortcoming, since negation is an indispensable element of a logic-based programming language to be sufficiently expressive for solving problems in practice.

Our extension of logic programs by generalized quantifiers is a natural enrichment of the formalism which allows for a natural and appealing solution to the problem of incorporating external functions into logic programs. As an application, we obtain a semantics for modular logic programming which is conceptually clean and, as a major advantage, harbors a full treatment of negation in the style of ordinary logic programs with negation.

Roughly, the idea of the approach is to allow in the bodies of rules besides standard logical literals also formulas $Q\mathbf{x}[R(\mathbf{x})]$, which describe the application of a generalized quantifier $Q\mathbf{x}$ to a structure defined by the predicate R. Informally, $Q\mathbf{x}$ can been seen as an interface to an external function whose argument is the predicate R. For any extension of R, the function resp. quantifier $Q\mathbf{x}$ evaluates to either true or false, and thus succeeds or fails as a subgoal in a rule body. An extension to functions with multiple arguments is straightforward.

In the present paper, we survey the expressive capability of logic programs with different classes of generalized quantifiers, where we focus mainly on finite structures. We pay particular attention to various natural syntactic fragments of the formalism. The programs may use generalized quantifiers of a particular complexity, and in particular generalized quantifiers which are definable by logic programs.

It appears that normal logic programs with quantifiers in NP capture the class Π_2^p of the polynomial hierarchy; thus, by adding such quantifiers, the expressive capability of normal programs increases from co-NP [51] by one level in the hierarchy. As a consequence, modular logic programs capture Π_2^p as well. The expressive capability further increases by allowing nested program modules. The natural extension of the formalism to nested modular logic programs captures the polynomial hierarchy, and is therefore as expressive as second-order logic over finite structures. In fact, the equivalence of nested modular logic programs to second order logic holds even for arbitrary structures. Moreover, natural syntactic fragments of the formalism capture the classes Π_k^p, $k \geq 1$, of the polynomial hierarchy, and over ordered finite structures, also the classes Δ_k^p and Θ_k^p, for

$k \geq 1$.

The rest of this paper is organized as follows. In the next section, we recall some basic concepts and fix notation. In Section 3, we consider logic programs with generalized quantifiers, and in Section 4, we describe a formalism for modular logic programming based on logic programs with generalized quantifiers. Thereafter, we address the expressive capability of the formalism in Section 5 and the computational complexity in Section 6. Programs with nested modules are considered in Section 7. Finally, Section 8 concludes the paper.

2 Preliminaries and Notation

Notation. 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. Fraktur letters $\mathfrak{A}, \mathfrak{B}, \ldots$ denote logical structures. Sets of structures are denoted by capital Greek letters Π, Σ or C, and classes or mappings thereof by $\mathcal{M}, \mathcal{Q}, \ldots$; lower case Greek letters $\tau, \sigma, \ldots$ denote signatures.

Logic. A signature[4] is a sequence $\tau = (P_1^{a_1}, \ldots, P_k^{a_k}, f_1^{b_1}, \ldots, f_l^{b_l}, c_1, \ldots, c_m)$ where the P_i are relational symbols of arity $a_i \geq 0$, the f_i are functions with $b_i \geq 1$ arguments, and the c_i are constants. τ is *relational*, if it contains only relational symbols.

A *structure* $\mathfrak{A}$ over τ is denoted by $(A, P_1^{\mathfrak{A}}, \ldots, P_k^{\mathfrak{A}}, f_1^{\mathfrak{A}}, \ldots, f_l^{\mathfrak{A}}, c_1^{\mathfrak{A}}, \ldots, c_m^{\mathfrak{A}})$. A is called the *universe* or *domain* of $\mathfrak{A}$, and denoted $|\mathfrak{A}|$. $\mathfrak{A}$ is finite if $|\mathfrak{A}|$ is finite. The set of all structures over τ is denoted by $\mathrm{Struct}(\tau)$.

Let $\mathfrak{A}, \mathfrak{B} \in \mathrm{Struct}(\tau)$, s.t. $|\mathfrak{A}| = |\mathfrak{B}|$. Then $\mathfrak{A} \subseteq \mathfrak{B}$, if $P_i^{\mathfrak{A}} \subseteq P_i^{\mathfrak{B}}$ for $1 \leq i \leq k$, $f_i^{\mathfrak{A}} = f_i^{\mathfrak{B}}$ for $1 \leq i \leq l$, and $c_i^{\mathfrak{A}} = c_i^{\mathfrak{B}}$ for $1 \leq i \leq m$. A *computational problem* over signature τ is a set $\Pi \subseteq \mathrm{Struct}(\tau)$, s.t. Π is closed under isomorphisms. Π is *monotone*, if $\mathfrak{A} \subseteq \mathfrak{B}$ and $\mathfrak{A} \in \Pi$ implies $\mathfrak{B} \in \Pi$.

For a relational signature τ and integer l, $\tau^{(l)} = (P_1^{la_1}, \ldots, P_k^{la_k})$ is called the *$l-$ary vectorization* of τ. For a computational problem Π, let $\Pi^{(l)}$ denote the problem Π over $l-$tuples, i.e. over universe $\underbrace{n \times \cdots \times n}_{l \text{ times}}$.[5] Let $\mathfrak{A}$ be a relational structure, and $U \subseteq |\mathfrak{A}|$. Then the restriction of $\mathfrak{A}$ to U, in symbols $\mathfrak{A}|U$, is the structure $(U, P_1^{\mathfrak{A}} \cap U^{a_1}, \ldots, P_k^{\mathfrak{A}} \cap U^{a_k})$. For a τ-structure $\mathfrak{A}$ and a signature τ_0 contained in τ, $\mathfrak{A}|\tau_0$ is the τ_0-structure obtained from $\mathfrak{A}$ by removing all relations, functions, and constants not contained in τ_0.

The set of all finite models of a formula Ψ is denoted by $\mathrm{Mod}(\Psi)$.

[4] In contrast to the usual convention in finite model theory, we do not disregard function symbols in our language at this stage, because they are commonly available in logic programming.

[5] This is a coherent definition, since we deal only with problems which are closed under isomorphisms.

Let $\varphi(x_1,\ldots,x_n)$ be a formula with free variables $x_1,\ldots,x_n$, and let $\mathfrak{A}$ be a structure. Then $\varphi^{\mathfrak{A}}$ denotes the $n-$ary relation $\{(d_1,\ldots,d_n) \mid \mathfrak{A} \models \varphi(d_1,\ldots,d_n)\}$.

Let $\mathcal{L}$ be a syntactic fragment of first-order logic. Given signatures τ, σ and a natural number k, a k-ary interpretation I of τ into σ is a definition of the $\sigma^{(k)}$ relations in terms of τ, i.e., a tuple of $\mathcal{L}$ formulas, s.t. for each predicate symbol R in σ with arity r, I contains a formula φ_R over τ with rk free variables which defines R^{rk}. For a structure $\mathfrak{A} \in \mathrm{Struct}(\tau)$, $I(\mathfrak{A})$ denotes the structure over $\sigma^{(k)}$ which is defined by I. Let $\Pi \subseteq \mathrm{Struct}(\tau)$, $\Sigma \subseteq \mathrm{Struct}(\sigma)$ be problems. Π is $\mathcal{L}$–reducible to Σ if there exist an interpretation I of τ into σ, s.t. for all $\mathfrak{A} \in \mathrm{Struct}(\tau)$, $\mathfrak{A} \in \Pi$ iff $I(\mathfrak{A}) \in \Sigma^{(k)}$ where k is the arity of I. By restricting the logic $\mathcal{L}$ for the interpretations, we obtain low-level reductions: A *quantifier-free reduction* [30] is a reduction whose defining formulas are quantifier-free.

Logic Programs. A logic program is a finite collection of clauses (also called *rules*) of the form

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

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 logic programming, a number of different semantics have been proposed, since the definition of negation is nontrivial and allows for alternative approaches, cf. [2, 8]. For background, consult [35].

Example 1. In the following program, the extension of TC in the least fixpoint is the transitive closure of the binary relation E. We say that the program *computes* TC.

$$\begin{aligned} TC(x,y) &\leftarrow E(x,y)\\ TC(x,y) &\leftarrow TC(x,z), E(z,y) \end{aligned}$$

Generalized Quantifiers (GQs). 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 $\mathfrak{A} \cong \mathfrak{B}$ and $\mathfrak{A} \in C$, then $\mathfrak{B} \in C$. 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.[6]

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

[6] In logic programming, 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.

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 $Q_C \mathbf{x}_1 \cdots \mathbf{x}_n[\varphi_1, \ldots, \varphi_n]$ 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 remaining free variables $\mathbf{y}$ after the formula.

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 $\mathfrak{A}$ be a structure and $\mathbf{b}$ a tuple in $\mathfrak{A}$ matching the arity of $\mathbf{y}$. Then $(\mathfrak{A}, \mathbf{b}) \models \Theta$, if and only if the structure $(A, \varphi_1^{\mathfrak{A},\mathbf{b}}, \ldots, \varphi_n^{\mathfrak{A},\mathbf{b}})$ belongs to C, where $\varphi_i^{\mathfrak{A},\mathbf{b}} = \{\mathbf{a} \mid \mathfrak{A} \models \varphi_i[\mathbf{a}, \mathbf{b}]\}$. The *family* $\mathbf{C}$ *of* C*-quantifiers* is the set $\{Q_C, Q_{C^{(2)}}, Q_{C^{(3)}}, \ldots\}$.

Example 2. The following GQs will be used in further examples.

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

Complexity Theory. We assume some familiarity with the basic concepts of complexity theory, such as the classes P, NP, L (logspace), oracle computations, and the polynomial hierarchy (PH) whose classes are Σ_k^p, Π_k^p, and Δ_k^p, $k \geq 1$; consult [20, 46] for background.

Intertwined with the levels Σ_k^p, Π_k^p of PH are the classes Θ_k^p [63] ($\Sigma_k^p \subseteq \Theta_{k+1}^p \subseteq \Sigma_{k+1}^p$), which are now considered intermediate constituents of PH. The Θ_{k+1}^p classes have natural complete problems in terms of optimization variants of Σ_k^p-complete problems.

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 2.1 [63] $\Theta_{k+1}^p = \mathrm{P}_{\|}^{\Sigma_k^p} = \mathrm{L}^{\Sigma_k^p}$.

3 Logic Programs with Generalized Quantifiers

Let τ_0 be a signature for describing the program input, and let τ be an extension of τ_0 by new relational symbols. A *logic program with GQs (GQLP)* on τ is a finite collection $\mathcal{P}$ of rules

$$A \leftarrow B_1, \ldots, B_m \tag{2}$$

where A is a τ-atom whose predicate does not occur in τ_0, and each B_i is either a τ-literal or a generalized quantifier literal (GQ-literal) over τ, which is defined next.

Suppose that Q_C 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 τ such that the arity of S_i equals a_i. Then, the formula

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

is a GQ-atom. Notice that the free variables of this GQ-atom are $\mathbf{v} = v_1, \ldots, v_l$. 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* is a possibly negated GQ-atom. For brevity, we denote a GQ atom (3) by

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

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

For any collection $\mathcal{Q}$ of generalized quantifiers, we denote by $\tau^*_{\mathcal{Q}}$ the extension of τ by all predicate letters $Q_C[\mathbf{S}]$, where $Q_C \in \mathcal{Q}$ and $\mathbf{S} = S_1, \ldots, S_n$ is a list of predicate letters S_i from τ which matches the signature of Q_C; every such $Q_C[\mathbf{S}]$ is a *GQ-predicate*. Notice that $\tau^*_{\mathcal{Q}}$ is finite if τ and $\mathcal{Q}$ are finite. Then, a GQLP over τ is syntactically an ordinary logic program over the signature $\tau^*_{\mathcal{Q}}$.

Example 3. Consider the following program $\mathcal{P}$, which uses two GQs: The isomorphism GQ $Q_{\cong}[G_1, G_2]$, which tells whether G_1 and G_2 are isomorphic graphs, and the transitive closure GQ TC:

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

Suppose that τ_0 contains the relation E and the constant symbols a and b. Given a graph $G = (V, E)$, and vertices $a, b \in V$, this program assigns the propositional letter Iso true on 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 $Q_C \mathbf{x}_1 \cdots \mathbf{x}_n[\varphi_1, \ldots, \varphi_n](\mathbf{v})$ may contain arbitrary first-order formulas φ_i and also GQs. In the context of logic programming, there is no need for allowing complex argument formulas φ_i, since such a φ_i can be associated with a defining equivalent auxiliary relation A_{φ_i} which is computed within the logic program. Moreover, 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 $\varphi^{\mathfrak{A}}$ for the semantical definition of the GQ.

In the context of first-order logic, the evaluation of formulas with GQs 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, even if no GQs are present, but just negation.

Since negation is provided by the family of complement GQs $Q_{\sim}$, 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.

We define the semantics of a GQLP $\mathcal{P}$ in the spirit of the stable model semantics for logic programs with negation [21], which is widely accepted. Informally, a stable model of a program $\mathcal{P}$ amounts to a solution of a fixpoint equation

$$\mathfrak{M} = f(\mathcal{P}, \mathfrak{M}),$$

where $f(\mathcal{P}, \mathfrak{M})$ is the result of $\mathcal{P}$ if negation is evaluated with respect to $\mathfrak{M}$.

Our approach is to treat GQ-literals like negative literals; since GQs subsume negation, negation is implicitly treated with (unnegated) GQ-atoms. Under an assumption, based on $\mathfrak{M}$, about the truth of GQ-literals and negative standard literals, a GQLP $\mathcal{P}$ is reduced similar as in [21] to an ordinary positive logic program which is evaluated. If the least minimal model of the latter program justifies the assumptions made by $\mathfrak{M}$, it is considered an acceptable model of $\mathcal{P}$; we call such a model *GQ-stable*.

We next provide a formalization of this concept. Suppose that we have signatures τ_0 and τ as above, a GQLP $\mathcal{P}$ and a structure $\mathfrak{A} \in \mathrm{Struct}(\tau)$.

Then, the *ground instantiation* of $\mathcal{P}$ on $\mathfrak{A}$, denoted $ground(\mathcal{P}, \mathfrak{A})$, is the collection of all interpreted rules $C\vartheta$, where C is from $\mathcal{P}$ and ϑ is any ground substitution over $\mathfrak{A}$.

Definition 3.1 *Let $\mathcal{P}$ be a GQLP and $\mathfrak{A} \in \mathrm{Struct}(\tau)$. The* reduct *of $\mathcal{P}$* w.r.t. *$\mathfrak{A}$, denoted $red(\mathcal{P}, \mathfrak{A})$, is the set of rules obtained from $ground(\mathcal{P}, \mathfrak{A})$ as follows.*

1. *Remove every rule r with a literal L in the body such that $\mathfrak{A} \not\models L$, where L is either negative or a GQ-literal.*
2. *Remove all negative literals and GQ-literals from the remaining rules.*

Notice that $red(\mathcal{P}, \mathfrak{A})$ is a collection of interpreted Horn clauses; thus, there is a least structure $\mathfrak{B}$, denoted $\mathfrak{A}_\infty(\mathcal{P})$, such that $\mathfrak{B}|\tau_0 = \mathfrak{A}|\tau_0$ (i.e., $\mathfrak{B}$ provides the same input to $\mathcal{P}$ as $\mathfrak{A}$) and $B \models r$, for every rule $r \in red(\mathcal{P}, \mathfrak{A})$. Since $red(\mathcal{P}, \mathfrak{A})$ is an ordinary logic program, the structure $\mathfrak{A}_\infty(\mathcal{P})$, also called the least model of $\mathcal{P}$ w.r.t. $\mathfrak{A}$, can be obtained as the least fixpoint of a monotonic operator, cf. [35].

Definition 3.2 *Let $\mathcal{P}$ be a GQLP and let $\mathfrak{A}_0 \in \mathrm{Struct}(\tau_0)$. An expansion $\mathfrak{A} \in \mathrm{Struct}(\tau)$ of $\mathfrak{A}_0$ is a* GQ-stable model of $\mathcal{P}$ on $\mathfrak{A}_0$*, iff it satisfies the fixpoint equation*

$$\mathfrak{A} = \mathfrak{A}_\infty(\mathcal{P}).$$

The collection of all stable models of $\mathcal{P}$ is denoted by $SM(\mathcal{P}, \mathfrak{A}_0)$.

The meaning *of $\mathcal{P}$ on $\mathfrak{A}_0$, denoted $\mathcal{M}^{st}_{\mathcal{P}}(\mathfrak{A}_0)$, is the structure which is the intersection of all GQ-stable models of $\mathcal{P}$ on $\mathfrak{A}_0$, i.e.,*

$$\mathcal{M}^{st}_{\mathcal{P}}(\mathfrak{A}_0) = \bigcap_{\mathfrak{A} \in SM(\mathcal{P}, \mathfrak{A}_0)} \mathfrak{A}.$$

(If $SM(\mathcal{P}, \mathfrak{A}_0) = \emptyset$, then $\mathcal{M}^{st}_{\mathcal{P}}(\mathfrak{A}_0)$ is the unique maximal structure $\mathfrak{B}$ such that $\mathfrak{B}|\tau_0 = \mathfrak{A}_0$.)

Example 4. Consider the following program $\mathcal{P}$, which uses the majority quantifier Q_M and the modularity quantifier Q_2 (i.e., the even quantifier).

$$\begin{aligned}
Q(x) &\leftarrow \neg S(x) \\
S(x) &\leftarrow \neg Q(x) \\
A(x) &\leftarrow Q_M[Q, S], S(x) \\
Q(x) &\leftarrow Q_2[A], S(x) \\
W(a, b) &\leftarrow
\end{aligned}$$

Suppose that τ_0 contains merely the constant symbols a and b, and τ contains in addition the relation symbols W, A, Q, and S.

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.

Consider Herbrand models on τ and let $\mathfrak{M}_1 = \{W(a, b), Q(a), Q(b)\}$. (We use the familiar notation for Herbrand models.) This interpretation is a GQ-stable model of $\mathcal{P}$ (w.r.t. the unique Herbrand model $\mathfrak{M}_0 \in \text{Struct}(\tau_0)$). Indeed, the reduct $red(\mathcal{P}, \mathfrak{M}_1)$ consists of the clauses

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

Clearly, $\mathfrak{M}_1$ is the least model of this program (w.r.t. $\mathfrak{M}_0$). Another GQ-stable model of $\mathcal{P}$ is $\mathfrak{M}_2 = \{W(a, b), B(a, b), S(a), S(b)\}$. The Herbrand model $\mathfrak{M}_3 = \{W(a, b), B(a, b), Q(a), S(b)\}$ is not a GQ-stable model of $\mathcal{P}$: $red(\mathcal{P}, \mathfrak{M}_3)$ contains the clauses $S(b) \leftarrow$ and $Q(b) \leftarrow S(b)$, which means that the least model of $red(\mathcal{P}, \mathfrak{M}_3)$ contains $Q(b)$. In fact, $\mathfrak{M}_1$ and $\mathfrak{M}_2$ are all GQ-stable models of $\mathcal{P}$ w.r.t. $\mathfrak{M}_0$, i.e., $SM(\mathcal{P}, \mathfrak{M}_0) = \{\mathfrak{M}_1, \mathfrak{M}_2\}$. Therefore,

Let us note some interesting properties of GQ-stable models. The following proposition tells that the term "model" is justified.

Proposition 3.1 *Let $\mathfrak{M}$ be a stable model of a GQLP $\mathcal{P}$. Then, $\mathfrak{M}$ is a model of $\mathcal{P}$, i.e., $\mathfrak{M} \models \forall^*(\mathcal{P})$, where $\forall^*$ is the universal closure of $\mathcal{P}$.*

Another important property is that GQ-stable models conservatively extend the concept of stable models for ordinary logic programs.

Proposition 3.2 *Let $\mathcal{P}$ be a GQLP. If $\mathcal{P}$ has no GQ-literals, then $\mathfrak{M}$ is a GQ-stable model of $\mathcal{P}$ iff $\mathfrak{M}$ is a stable model of $\mathcal{P}$.*

Therefore, we refer in the sequel to GQ-stable models simply as stable models.

As Example 4 has shown, a program $\mathcal{P}$ may have multiple stable models on a structure. Inherent nondeterminism in the choice of the rules enables the generation of an enumeration of the universe of a finite structure; we illustrate this in the next example.

Example 5. Consider the following program LP_{succ}, which does not contain generalized quantifiers. Therefore, it behaves according to stable semantics for ordinary logic programs [21]. We shall refer to this program in Section 5.

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

In this program, L and S are binary relations which intuitively describe a linear order $<$ on the domain and the associated successor relation, respectively; the relations L^- and S^- describe the complements of L and S, respectively.

The propositional atom $\perp$ in the heads of the rules is meant to be a logical constant for falsity. Such a constant can be easily defined in stable semantics, by adding a rule

$$r: \quad P_\perp \leftarrow \perp, \neg P_\perp$$

where $P_\perp$ is a fresh propositional letter which does not occur elsewhere. Then, in every stable model $\mathfrak{M}$, proposition $\perp$ must be false, since otherwise, the rule r cannot be satisfied under stable negation.

It is easy to see that on any finite universe U, the relation L is in every stable model a linear order on U, and S is the corresponding successor relation. *Therefore, we may assume that in any stable model of a program on a finite structure, the universe is enumerated.* Of course, this does not mean that stable model semantics actually defines an order on the domain, because both S and L are empty in the intersection of all stable models.

4 Application: Modular Logic Programming

In this section, we describe how the formalism of logic programs with generalized quantifiers can be fruitfully applied for defining the semantics of logic programs which refer to program modules. This approach can be found in [16, 17], and we recall the definitions from there.

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 and restrict the number of output relations to $n = 1$, it is clear that such a module is a generalized quantifier. The restriction to a single output relation is not essential, however.

Thus, 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 modular logic programs is then inherited from the semantics of GQLPs. Notice that this approach is very flexible, and allows for easy incorporation of external predicates and functions, 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 a logic program, which is a *logic program module*. The similarity type of $LP[\mathbf{Q}]$ is the list of arities of predicates in $\mathbf{Q}$.

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*. An atom is a *call atom*, 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* predicates (atoms, literals respectively.)

Definition 4.1 ([16]) *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 $\mathbf{Q}$ can be seen as variables for relations, which are substituted at a call by relations over the domain. All relations in a module $LP[\mathbf{Q}]$ are local; no relations from outside are visible inside a module. The similarity type of a module $LP[\mathbf{Q}]$ is the input signature τ_0 of its implementation program P. Here, τ_0 is relational, and we do for simplicity not consider possible constants and functions in the body P; they could be provided in an extended signature τ_0.

Note that the program P does not contain call literals. Hence, the stable semantics (as well as other 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 7.

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 ([16]) *A* modular logic program *(MLP) is a finite collection LP 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 $Q'_i \in \mathbf{Q}'$.

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 the semantics $\mathcal{M}^{st}$ as a GQ $Q_{C(\mu)}$ which is associated with the collection $C(\mu)$ of all structures $\mathfrak{A} = (A, \mathbf{Q}', R')$, 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 atom $LP(\mathbf{a})$ on $\mathfrak{A}_0 = (A, \mathbf{Q}')$; i.e.,

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

We call $Q_{C(\mu)}$ the *module quantifier of μ under $\mathcal{M}^{st}$.*

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

Example 6. Let $\mathfrak{M} = \{E(1,2), E(2,3), E(3,1), E(2,4), U(1)\}$ be a Herbrand model of the signature $\tau = \{E, U\}$ on the domain $\{1,2,3,4\}$. $E^{\mathfrak{M}}$ is the digraph depicted below.

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

Consider the module $\mu = (TC[G], P_{TC})$, where P_{TC} is the program from Section 2. The program P_{TC} has on $\mathfrak{A}_0 = \{G(1,2), G(2,3), G(3,1), G(2,4)\}$ the least model $\mathfrak{M}' = \{G(1,2), G(2,3), G(3,1), G(2,4)\} \cup \{TC(i,j), TC(i,4) \mid 1 \leq i,j \leq 3\}$. Thus, a call $TC[E]$ of μ where E is defined in $\mathfrak{M}$, yields that e.g. $TC[E](1,1)$ is true, while $TC[E](4,1)$ is false.

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.

5 Expressiveness

Dependency Graph. To obtain a fine-grained analysis of the expressiveness of GQLPs, we generalize the familiar concept of dependency graph from ordinary LPs to GQLPs. 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.

Definition 5.1 ([16]) *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 $P_1(\mathbf{t}_1) \leftarrow ..., \neg P_2(\mathbf{t}_2), ...$ 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 7. Consider a GQLP and its dependency graph. (red arcs are symbolized by label "¬", yellow arcs by "?"; green arcs are unlabeled).

$$\begin{array}{rl} E(x,y) & \leftarrow G(x,y,y) \\ F(x,y) & \leftarrow \neg G(x,x,y) \\ Y & \leftarrow Q_{\cong}[E,F] \\ N & \leftarrow \neg Y \end{array}$$

$$\begin{tikzcd}
 & G & & & \\
E \arrow[ur] & & F \arrow[ul, "\neg"'] & & \\
 & Q_{\cong}[E,F] \arrow[ul, "?"] \arrow[ur, "?"'] & & Y \arrow[ll, "?"'] & N \arrow[l, "\neg"']
\end{tikzcd}$$

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

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

Modular Programs. 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 $\mathcal{C}_S$. (Since subprograms do not contain call predicates, their dependency graphs do not contain yellow edges.)

Expressive Power. Let $\mathcal{L}$ be a class of logic programs, and let C be a complexity class. We say that $\mathcal{L}$ *has expressivity* C (or, equivalently, $\mathcal{L}$ *captures* C) *over a class of structures* $\mathcal{K}$, if

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

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

The expressive power of ordinary logic programming over finite structures has been investigated e.g. in [51, 15, 49]; see [11] for a survey. Stable model semantics was shown to be quite expressive.

Proposition 5.1 ([51]) **LP** *captures* co-NP*, or equivalently, all* Π^1_1 *definable queries over finite structures.*

Thus, stable model semantics has the full power of Π^1_1. In fact, it has been shown that even monotone programs can be syntactically captured:

Theorem 5.2 ([57]) *Monotone NP is captured by the fragment of second order logic where all relational symbols (but not the second order relational variables) occur only positively.*

5.1 Finite structures

In this subsection, we survey the expressiveness of modular logic programs over finite structures. The results are compactly summarized in Table 2 ($\mathbf{LP^k}$ and $\mathbf{LP}^*$ are defined in Section 7).

Language	*Expressiveness*	
LP(X), CI(X), CRF(X) $\mathbf{X} \in \{\mathbf{LP}, \mathbf{Mon}\}$ **2Pos(LP), SStrat(LP)**	Π_2^1	
2Pos(Mon)	Δ_2^1	*
SCI(X), Strat(X), RF(X) $\mathbf{X} \in \{\mathbf{LP}, \mathbf{Mon}\}$ **SStrat(Mon)**	Θ_2^1	*
LP, Pos(LP)	Π_1^1	
Pos(Mon)	monotone Π_1^1	
$\mathbf{LP}^k$	Π_k^P	
$\mathbf{LP}^*$	Second-Order Logic	

Table 2. Expressiveness of MLP over finite structures

In this table, an asterisk "*" indicates that the proof requires an enumeration of the universe, and thus establishes capturing over ordered finite structures. Note, however, that capturing of Θ_2^p over all finite structures is not possible unless the exponential Boolean hierarchies EBH and ExpBH collapse to some intermediate level; similar results hold for Δ_2^p, cf. [12].

Arbitrary Program Modules. The following theorem states the full expressive power of the formalism.

Theorem 5.3 **LP(LP)** *captures* Π_2^p *over finite structures.*

Proof. (Sketch) A straightforward algorithm shows that the expressivity of **LP(LP)** is in Π_2^p.

For the other direction, consider a Π_2^1 formula $\varphi(x_1, \ldots, x_n)$ over signature $\tau = (\mathbf{Q})$. In the language extended by a new relation symbol B of arity n, it follows by the extension of Fagin's Theorem [58] that there exists a Π^1_2 formula $\forall R \exists S.\varphi$ which is true on a structure $\mathfrak{A}$ iff $\varphi^{\mathfrak{A}} \subseteq B^{\mathfrak{A}}$. Since stable semantics captures co-NP, there exists a program quantifier P_S for the relation $\neg \exists S.\varphi$. Then the program

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

computes precisely the relation A as the *intersection* of all stable models. ∎

Remark. Note that in **LP(LP)**, the subprograms are arbitrary, so that any quantifiers in co-NPcan be computed, and in particular, arbitrary vectorizations of a given quantifier.

The full expressive power of modular logic programs is available in some of the syntactical fragments for main programs from above.

Corollary 5.4 CRF(LP) *and* **CI(LP)** *capture Π^p_2 over finite structures.*

Proof. Follows from the fact that the program in the proof of Theorem 5.3 is call recursion free and call independent. ∎

Corollary 5.5 SStrat(LP) *captures Π^p_2 over finite structures.*

Proof. It suffices to show that the complement quantifier $C^{(k)}_{\sim}$ (the vectorization of $C_{\sim}$ to arbitrary arity k) can be expressed in **SStrat(LP)**: To this end, we use additional modules N^k, $k \geq 1$:

$$\begin{aligned} &N^k[S] : \\ &\;\; N^k(\mathbf{x}) \leftarrow \neg S(\mathbf{x}) \end{aligned}$$

It appears that the expressive power of the remaining syntactic fragments for main programs have expressive power limited to a subclass of Δ^p_2; on ordered structures, this subclass is Θ^p_2. For the next results, we make use of the following normal form, which has been introduced in [53, 54] and generalized in [22].

Theorem 5.6 (Stewart Normal Form [53, 54, 22]**)** *Let Ω be NP-complete under quantifier-free reductions over ordered finite structures. Then Θ^p_2 is captured over ordered structures by the class of formulas*

$$\exists \mathbf{x}.\ \Omega\mathbf{x}.[\varphi_1](\mathbf{v}) \wedge \neg \Omega\mathbf{x}.[\varphi_2](\mathbf{v})$$

where φ_1 and φ_2 are quantifier-free formulas.

Theorem 5.7 SCI(LP) *captures Θ^p_2 over all ordered finite structures.*

Proof. (Sketch) Call-Independence means that the oracle calls can be made in parallel, and hence Theorem 2.1 implies that the expressivity is in $P_{\|}^{NP}$. On the other hand, Theorem 5.6 leads to a program of the form

$$A(\mathbf{y}) \leftarrow P_{P1}[\mathbf{Q}](\mathbf{x},\mathbf{y}), \neg P_{P2}[\mathbf{Q}](\mathbf{x},\mathbf{y}) \tag{5}$$

■

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 (5). Surprisingly, the expressive power does not increase if we move from **SCI(LP)** to **Strat(LP)**.

Theorem 5.8 *Over ordered finite structures,* **SCI(LP) = Strat(LP) = RF(LP)**.

Proof. (Sketch) 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. From the characterization of Θ_2^p in terms of parallel queries (Proposition 2.1) it follows that 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 5.7.

The result for **RF(LP)** follows immediately from the result about **Strat(LP)** and the fact that the program in the proof of Theorem 5.7 is actually also from this class. ■

Corollary 5.9 **SCI(LP), Strat(LP)**, *and* **RF(LP)** *obey the Stewart Normal Form.*

Monotone Program Modules. Let us now turn to monotone modules, i.e., programs which define a monotone generalized quantifier. Here, we have the following.

Theorem 5.10 ([17]) **Pos(Mon) = Mon**, *i.e.* **Pos(Mon)** *captures the class of all monotone co*NP *queries.*

The intuitive reason for this result is that any program in **Pos(Mon)** is monotone, and its evaluation can be defined in terms of a monotone operator [17].

Let us now proceed to richer fragments of main programs than **Pos**. For stratified program, we obtain the following result.

Theorem 5.11 *Over ordered finite structures,* **Strat(Mon)**, **SStrat(Mon)**, *and* **RF(Mon)** *capture* Θ_2^p.

Proof. (Sketch) With monotone modules, all yellow edges become "monotone", and thus all cycles in the given program classes become monotone, and therefore by Theorem 5.10 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. By the following lemma, we can overcome this problem by including non-recursive rules which simulate the quantifier-free reductions.

Lemma 5.12 *Over ordered finite structures, there exists a subclass* **R** *of* (**CI** $\cap$ **RF**)*, such that* **R(Mon)** = **LP**.

To see the lemma, from [56, 57] we know that there exist monotone problems which are complete for NP under quantifier-free reductions with successor; such reductions can be easily computed by recursion-free programs.

Thus, by the lemma, it suffices to add a rule like in the proof of Theorem 5.7, and we obtain an equivalent program in **RF(Mon)**. ∎

For the remaining fragments of main programs, the expressiveness is higher.

Corollary 5.13 **LP(Mon)**, **CRF(Mon)**, *and* **CI(Mon)** *capture* Π_2^p.

Proof. (Sketch) Membership follows from Theorem 5.3. For expressiveness, we extend the idea of the proof of Theorem 5.11. Thus, we employ the program used in the proof of Theorem 5.3, and provide the **LP** modules by Lemma 5.12.

Since the proof of Lemma 5.12 requires ordered domains, we add the rules of program LP_{succ} (cf. Section 3) which in every stable model defines a successor relation on the domain. Since the quantifier-free reduction can be simulated recursion-free, the result follows. ∎

5.2 Modules under brave and skeptical stable semantics

The program classes **2Pos(*)** are natural generalizations of the respective **Pos(*)** classes. In the class **2Pos(*)**, it is possible to call a program module in one of two modes, which correspond to the brave and the skeptical variant of the stable model semantics. The mode corresponding to the skeptical variant is just the one that we used for evaluating modules under the stable semantics so far; the mode corresponding to the brave variant follows a dual definition, which takes the union of all stable models rather than their intersection. The mode of a module call $LP[\mathbf{Q}](\mathbf{t})$ is indicated by prefixing the call predicate $LP[\mathbf{Q}]$ with "$\Box$" for the skeptical mode, and with "$\Diamond$" for the brave mode. Similar reasoning modes are employed in epistemic logic programming [4].

This approach supports programming with modules under different calling (or reasoning) modes.

Clearly, **2Pos(LP)** and **LP(LP)** have the same expressive power since negation can be provided by a module (cf. proof of Corollary 5.5). 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; for such programs, a least fixpoint computation can be applied for evaluating the semantics. The expressiveness must be higher than co-NP, since both co-NP-hard and NP-hard modules can be employed.

To investigate the exact expressibility, we utilize the following strong result.

Theorem 5.14 ([55]) *Let $\Omega \in$ NP be monotone such that the Hamiltonian Path Problem HP is monotone projection-reducible to Ω. Then Δ_2^p is captured over ordered structures by least fixed point logic, extended by the family of generalized quantifiers for Ω.*

Moreover, every Δ_2^p property can be defined by a formula

$$\mathrm{LFP}_{\mathbf{x},R}\ \Omega\mathbf{y}[\varphi_1](\mathbf{x},\mathbf{max}) \vee \neg\Omega\mathbf{z}[\varphi_2](\mathbf{x},\mathbf{max})$$

where R occurs only positively in the formula, and **max** *denotes a tuple of the maximum elements of the order.*

Theorem 5.15 **2Pos(Mon)** *captures Δ_2^p over ordered finite structures.*

Proof. (Sketch) 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 thus obviously exists.

For expressiveness, we apply Theorem 5.14 to 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:

$$\begin{aligned} R(\mathbf{x},\mathbf{y}) &\leftarrow \Box P_{H1}[\mathbf{Q},R](\mathbf{x},\mathbf{y}) \\ R(\mathbf{x},\mathbf{y}) &\leftarrow \Diamond P_{H2}[\mathbf{Q},R](\mathbf{x},\mathbf{y}) \\ A(\mathbf{y}) &\leftarrow R(\mathbf{max},\mathbf{y}) \end{aligned}$$

■

Analogous generalizations **2X(*)** can be defined for the other syntactic fragments **X** of main programs; however, since they have negation, the brave calling mode can be easily emulated by the skeptical mode using auxiliary predicates; the expressive power by allowing calls in brave mode thus remains unchanged. Therefore, we do not consider such generalizations here.

5.3 Beyond finite structures

Theorem 5.3 shows that modular logic programming has the same expressive power as disjunctive logic programming over finite structures [15]. Thus, it appears that modular logic programming can be translated into disjunctive logic programming and vice versa. However, such translations are in general only available for finite structures. As shown recently [14], over infinite Herbrand models (resp. the integers) DLP under the stable semantics can express precisely all Π^1_1 definable relations, and thus has the same expressive power as normal logic programming [51] on these models.

It is easy to see that the proof of Theorem 5.3 works over infinite Herbrand models resp. the integers as well, and thus establishes that MLP can express all Π^1_2 definable relations over these models. Since, as well-known, there are Π^1_2 definable relations over the integers which are not Π^1_1 definable, it follows that MLP is strictly more expressive than DLP on the integers.

In fact, since the stable semantics expresses all Π^1_1 relations over arbitrary structures [51], the proof of Theorem 5.3 establishes that modular logic programs express Π^1_2 over arbitrary structures. Thus, modular logic programming is a highly expressive (and complex) formalism in general.

6 Complexity

In this section, we survey the complexity of modular logic programs, and address the data and the program complexity [60] of the various classes of modular logic programs which we have introduced above.

Data Complexity. The *data complexity* of an LP class $\mathbf{C}$ is the complexity of the mappings $\mathcal{M}^{st}_{LP}$ for fixed programs $LP \in \mathbf{C}$. All entries in Table 2 imply completeness results for the respective complexity classes, because capturing of C by a language $\mathcal{L}$ and the existence of a complete problem in C clearly imply that $\mathcal{L}$ contains some C-complete problem. Note, however, that the converse does not necessarily hold; a language $\mathcal{L}$ can be C-complete while it does not capture C. For example, **Pos** is known to be P-complete with respect to data complexity, while it does not capture P (see [11]). Therefore, capturing of C is a stronger concept than C-completeness.

Program Complexity. Besides data complexity, Vardi [60] introduced the concept of *program (resp. combined) complexity*: program complexity[7] is the complexity of evaluating varying logic programs over a fixed input structure $\mathfrak{A}_0$; in combined complexity, both LP and $\mathfrak{A}_0$ are varying. Typically, program complexity and combined complexity are the same (wrt logspace reducibility), and they are exponentially higher than the data complexity.

In [23, 24], a method for deriving program complexity results from expressivity results has been developed. It is based on complexity theoretic investigations of succinct problems, i.e. computational problems, whose instances are not given in the usual way as a string, but by a Boolean circuit or by a formula representing this string. The representation by a formula is of particular interest, since it can be used for simulating succinct input representation within most logics used in computer science.

[7] In [60], the term *expression complexity* was used instead of *program complexity*. Both terminologies are widely used.

Language	*Data Complexity*	*Program Complexity*
LP(X), CI(X), CRF(X) $\mathbf{X} \in \{\mathbf{LP}, \mathbf{Mon}\}$ **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(X), Strat(X), RF(X) $\mathbf{X} \in \{\mathbf{LP}, \mathbf{Mon}\}$ **SStrat(Mon)**	Θ_2^p	$E\Theta_2^P = \mathrm{PSPACE}^{\mathrm{NP}}$
LP, Pos(LP)	coNP	coNEXP
Pos(Mon)	coNP	coNEXP
$\mathbf{LP}^k$	Π_k^1	$E\Pi_k^P$
$\mathbf{LP}^*$	PH	EXPSPACE

Table 3. Complexity of MLP over finite structures

The following two syntactic closure properties are easily satisfied by most logics:

Definition 6.1 *A logic $\mathcal{L}$ is* uniformly interpretation closed*, if there exists a mapping*

$$(\rho, \varphi_1, \ldots, \varphi_r) \rightarrow \rho[R_1/\varphi_1, \ldots, R_r/\varphi_r]$$

which is computable in logspace and such that for all structures $\mathfrak{A}$, $\mathcal{L}$-expressions ρ, and quantifier-free first order formulas $\varphi_1, \ldots, \varphi_r$ it holds that

$$(\mathfrak{A}, \varphi_1^{\mathfrak{A}}, \ldots, \varphi_r^{\mathfrak{A}}) \models \rho \iff \mathfrak{A} \models \rho[R_1/\varphi_1, \ldots, R_r/\varphi_r]$$

Definition 6.2 *A logic $\mathcal{L}$ is* uniformly vector closed*, if there exists a mapping*

$$(\varphi, w) \mapsto \varphi^{(|w|)}$$

which is computable in logspace and such that

$$\mathrm{Mod}(\varphi)^{(|w|)} = \mathrm{Mod}(\varphi^{(|w|)})$$

for all binary strings w and $\mathcal{L}$-expressions φ.

For brevity we say that a logic is *uniformly closed* if it is both uniformly interpretation closed and uniformly vector closed.

Lemma 6.1 ([24]) *If $\mathcal{L}$ captures C and is uniformly closed, then $\mathcal{L}$ has program complexity at least $E(C)$ and $Exp(C)$ under logspace-reductions.*

Here, $E(C)$ and $Exp(C)$ refer to the exponential analogues of the class C; thus $E(\mathrm{NP}) = \mathrm{NE}$, and $Exp(\mathrm{NP}) = \mathrm{NEXPTIME}$; see [24] for details.

We apply this lemma for precisely capturing the program complexity of all syntactic fragments of modular logic programs that we have considered. The results are summarized in Table 3.

Theorem 6.2 *All program complexity results of Table 3 hold.*

Proof. All languages in this paper are uniformly interpretation closed: For a quantifier-free formula φ_i, it is easy to write a program P_φ simulating φ_i, whose output predicate is R_i. Uniform vector closure is obtained by replacing all occurrences of variables x by tuples $x_1, \ldots, x_n$ of variables, for $n \geq 1$. ■

In fact, the languages in this paper are even able to simulate succinct circuit problems, cf. [24]. Therefore, program complexity can be related to the concept of leaf languages [61, 62]; it was shown that leaf languages always admit complete problems under quantifier-free reductions. A straightforward encoding of programs as finite structures shows that the above notions of uniformity can be restricted to quantifier-free uniformity (see [61]), and therefore we obtain the following corollary:

Corollary 6.3 *Let $\mathcal{L}$ be any LP language considered in this paper. If $\mathcal{L}$ captures C, then $\mathcal{L}$ has program complexity at least* $\mathrm{Leaf}^P(C)$. *This holds even with respect to quantifier-free reductions.*

Remark. (1) In [41], a diagonalization argument is given to prove that there are arity hierarchies within certain logics. A similar argument shows that the combined complexity of languages closed under complementation is always higher than the expressiveness.

Indeed, consider programs over strings (i.e., unary alphabets with linear order). For a program P, let $\lceil P \rceil$ denote its encoding as a string. For a language $\mathcal{L}$ which is closed under complementation, we define the set $AUTSAT(\mathcal{L}) = \{\lceil P \rceil \mid \lceil P \rceil \models P\}$; i.e., $AUTSAT(\mathcal{L})$ contains those programs P which evaluate to true on their own representation. The complexity of $AUTSAT(\mathcal{L})$ is clearly a lower bound for the combined complexity of $\mathcal{L}$. Moreover, $AUTSAT(\mathcal{L})$ is not definable in $\mathcal{L}$: Suppose its complement $\Delta(\mathcal{L})$ is definable in $\mathcal{L}$: Then there exists a program $P' \in \mathcal{L}$, such that for all programs $P \in \mathcal{L}$, $\lceil P \rceil \models P'$ iff $\lceil P \rceil \notin AUTSAT(\mathcal{L})$. Setting $P = P'$, we arrive at a contradiction.

As a matter of fact, when we say that the data complexity of a language (e.g. NP) is strictly weaker than its program complexity (e.g. NEXPTIME), this is just an application of the time hierarchy theorems [32], which are based on a more refined diagonalization technique.
(2) The same technique can be used to show in an easy way that restricting the arity of relation symbols in those classes of programs which are closed under complement yield infinite hierarchies wrt to expressiveness: Consider for example $\mathbf{RF}(\mathbf{LP})$. For $k \geq 1$, $\mathbf{RF}_k(\mathbf{LP}_k)$ is the class of $\mathbf{RF}(\mathbf{LP})$ programs, where all relations have maximum arity k. It is easy to see that the combined complexity of

$\mathbf{RF}(\mathbf{LP})$ remains in Θ_2^p, and hence there is a $\mathbf{RF}_k(\mathbf{LP}_k)$ program LP computing $\Delta(\mathbf{RF}_k(\mathbf{LP}_k))$. Like in (1), we conclude that LP must have arity at least $k+1$, and thus, the result follows.

Theorem 6.4 *Let $\mathcal{L}$ be any logic programming language considered in this paper. If $\mathcal{L}$ captures a complexity class C, which is closed under complement, then $\mathcal{L}$ has infinite arity hierarchies wrt to expressiveness.*

7 Nested Program Modules

It is evident that a modular logic program can again be used as a subprogram to a new logic program; proceeding this way, we obtain nested modular programs of arbitrary depth. A program which is built up this way and which does not contain cyclic dependencies of subprograms (i.e., no recursive calls of modules), is called *hierarchical.*

For finitely nested, hierarchical programs, a semantics can be inductively defined, based on the semantics of programs of nesting depth 2 which we presented. In particular, stable models of such programs can be naturally defined, and stratified, positive, etc. fragments of programs can be considered. We do not carry out this here; the formalization should be clear.

We extend out notation for nested classes in the following way. For any $k \geq 1$, $\mathbf{LP}^k$ 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, $\mathbf{LP} = \mathbf{LP}^1$ and $\mathbf{LP}^{k+1} = \mathbf{LP}(\mathbf{LP}^k)$.

Theorem 7.1 $\mathbf{LP^k}$ *captures the class Π_k^p of PH, and* $\mathbf{LP}^*$ *captures PH.*

Proof. Analogous to Theorem 5.3. ■

Thus, modular logic programming with nestings is as expressive as full second-order logic over finite structures (cf. [58]). In fact, this is an interesting difference to extensions of first order logic by generalized quantifiers: Although logic programs can be considered as a nested quantifier, there is no collapse to the Θ_2^p level, as there is in extensions of first order logic by NP quantifiers [22]; the reason is that in stable model semantics, there is a direct access to relational variables which is not the case for generalized quantifiers.

Moreover, from the considerations in Section 5.3, we can conclude that nested modular logic programs allow to express full second order logic over arbitrary structures. Thus, $\mathbf{LP}^*$ programming with nestings is a tremendously expressive language.

An inductive definition of different fragments by nesting programs from different fragments of $\mathbf{LP}$ is straightforward. For example, we can define classes such as $\mathbf{Strat}(\mathbf{LP}(\mathbf{Strat}))$, $\mathbf{Strat}(\mathbf{CRF}(\mathbf{LP}))$ etc.

The capturing results from Table 2 can be lifted to all constituent classes of PH by appropriate classes of nested MLPs. For example, $\mathbf{Strat(SStrat(LP))}$ captures Θ_3^p, and it obeys the Stewart Normal Form in the main programs. Note that $\mathbf{LP(Strat(LP))}$ collapses to $\mathbf{LP(LP)}$ since the logspace computations in the medium module can be transferred to the main program.

8 Conclusion

The extension of logic programs by generalized quantifiers is a useful tool for a natural formalization of modular logic programs.

As shown above, the formalism for modular logic programming described in [17, 16] has high expressive capability and equals the power of second order logic. Over finite structures, it captures the polynomial hierarchy, and natural fragments of the language capture the constituent classes in this hierarchy.

Several issues remain for further research. On the semantical side, an intriguing issue is a semantics for nested modular logic programs which are not hierarchical, i.e., modular logic programs which use recursive calls of modules. Furthermore, it would be interesting to experience semantics for modular logic programs which are based on semantics different from the stable model semantics; in particular, for semantics which use multi-valued logics (see [11] for several alternative semantics of logic programs).

Another direction of research is the study of the effect of restricting arities of generalized quantifiers on expressiveness. In particular, it would be interesting to see arity hierarchies for program classes capturing complexity classes which are not closed under complementation (e.g., $\mathbf{LP}(\mathbf{LP})$). Moreover, the picture of the arity hierarchies in the other classes of programs remains to be refined.

Acknowledgment

We are grateful to Janos Makowsky for valuable comments and remarks.

References

1. A. Aho and J. Ullman. Universality of Data Retrieval Languages. In *Proceedings ACM Symposium on Principles of Programming Languages*, pages 110–117, 1979.
2. K. Apt and N. Bol. Logic Programming and Negation: A Survey. *Journal of Logic Programming*, 19/20:9–71, 1994.
3. 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.
4. C. Baral and M. Gelfond. Logic Programming and Knowledge Representation. *Journal of Logic Programming*, 19/20:73–148, 1994.
5. J. Barwise. Some Applications of Henkin Quantifiers. *Israel Journal of Mathematics*, 25:47–63, 1976.
6. J. Barwise and R. Cooper. Generalized Quantifiers and Natural Language. *Linguistic and Philosophy*, 4:159–219, 1981.
7. J. Barwise and S. Feferman. *Model-Theoretic Logics*. Springer, 1985.
8. N. Bidoit. Negation in rule-based database systems: a survey. *Theoretical Computer Science*, 78:3–83, 1991.
9. A. Blass and Y. Gurevich. Henkin Quantifiers and Complete Problems. *Annals of Pure and Applied Logic*, 32:1–16, 1986.

10. M. Bugliesi, E. Lamma, and P. Mello. Modularity in Logic Programming. *Journal of Logic Programming*, 19/20:443–502, 1994.
11. E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and Expressive Power of Logic Programming. In *Proceedings of the Twelth IEEE International Conference on Computational Complexity (CCC '97)*, pages 82–101, 1997.
12. A. Dawar, G. Gottlob, and L. Hella. Capturing Relativized Complexity Classes without Order. *Mathematical Logic Quarterly*, 15:109–122, 1998.
13. H.-D. Ebbinghaus and J. Flum. *Finite Model Theory*. Perspectives in Mathematical Logic. Springer, 1995.
14. T. Eiter and G. Gottlob. Expressiveness of Stable Model Semantics for Disjunctive Logic Programs with Functions. *Journal of Logic Programming*, 33(2):167–178, 1997.
15. T. Eiter, G. Gottlob, and H. Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–417, September 1997.
16. T. Eiter, G. Gottlob, and H. Veith. Modular Logic Programming and Generalized Quantifiers. In J. Dix, U. Furbach, and A. Nerode, editors, *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-97)*, number 1265 in LNCS, pages 290–309. Springer, 1997.
17. T. Eiter, G. Gottlob, and H. Veith. Modular Logic Programming and Generalized Quantifiers. Technical Report CD-TR 97/111, Christian Doppler Laboratory for Expert Systems, TU Vienna, January 1997. Preliminary Report.
18. H. Enderton. Finite Partially-Ordered Quantifiers. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 16:393–397, 1970.
19. M. Fitting. Enumeration Operators and Modular Logic Programming. *Journal of Logic Programming*, 4:11–21, 1987.
20. M. Garey and D. S. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979.
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. G. Gottlob. Relativized Logspace and Generalized Quantifiers over Ordered Finite Structures. *Journal of Symbolic Logic*, 62(2):545–574, 1997.
23. G. Gottlob, N. Leone, and H. Veith. Second-Order Logic and the Weak Exponential Hierarchies. In J. Wiedermann and P. Hajek, editors, *Proceedings of the 20th Conference on Mathematical Foundations of Computer Science (MFCS '95)*, number 969 in Lecture Notes in Computer Science, pages 66–81, Prague, 1995.
24. G. Gottlob, N. Leone, and H. Veith. Succinctness as a Source of Complexity in Logical Formalisms. *Annals of Pure and Applied Logic*, 97:231–260, 1999.
25. E. Grädel. Capturing Complexity Classes with Fragments of Second Order Logic. *Theoretical Computer Science*, 101:35–57, 1992.
26. P. Hájek. *Metamathematics of Fuzzy Logic*. Kluwer Academic Publishers, 1998.
27. 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.
28. J. Hintikka. Quantifiers vs. Quantification Theory. *Linguistic Inquiry*, 5:153–177, 1974.
29. J. Hintikka. Quantifiers in Logic and Quantifiers in Natural Languages. In S. Körner, editor, *Philosophy of Logic*, chapter V. Univ. of California Press, Berkeley et al, 1976.

30. N. Immerman. Languages that Capture Complexity Classes. *SIAM Journal of Computing*, 16:760–778, 1987.
31. N. Immerman. *Descriptive Complexity*. Springer, 1997.
32. D. S. Johnson. A Catalog of Complexity Classes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 2. Elsevier Science Publishers B.V. (North-Holland), 1990.
33. M. Krynicky and J. Väänänen. Henkin and Function Quantifiers. *Annals of Pure and Applied Logic*, 43:273–292, 1989.
34. P. Lindström. First Order Predicate Logic with Generalized Quantifiers. *Theoria*, 32:186–195, 1966.
35. J. Lloyd. *Foundations of Logic Programming*. Springer, Berlin, 1984, 1987.
36. J. Makowsky. Model Theory and Computer Science: An Appetizer. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume I, chapter 6. Oxford University Press, 1992.
37. J. Makowsky. Capturing Relativized Complexity Classes with Lindström Quantifiers. In W. D. Pauli-Schimanovich, editor, *The Foundational Debate*, pages 133–140. Kluwer Academic Publishers, 1995.
38. J. Makowsky and Y. Pnueli. Oracles and Quantifiers. In *Computer Science Logic 1993 (CSL'93)*, number 832 in LNCS, pages 189–222. Springer, 1994.
39. J. Makowsky and Y. Pnueli. Computable Quantifiers and Logics over Finite Structures. In M. Krynicki, M. Mostowski, and L. Szczerba, editors, *Quantifiers: Logics, Models and Computation, Volume I*, pages 313–357. Kluwer Academic Publishers, 1995.
40. J. Makowsky and Y. Pnueli. Logics Capturing Relativized Complexity Classes Uniformly. In D. Leivant, editor, *Proceedings LCC'94, Logic and Computational Complexity*, number 960 in LNCS, pages 463–479. Springer, 1995.
41. J. Makowsky and Y. Pnueli. Arity and Alternation in Second-Order Logic. *Annals of Pure and Applied Logic*, 78:189–202, 1996.
42. 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.
43. R. Montague. In R. Thomasson, editor, *Formal Philosophy: Selected Papers*. Yale University Press, New Haven, Conn, 1974.
44. A. Mostowski. On a Generalization of Quantifiers. *Fundamenta Mathematicae*, 44:12–36, 1957.
45. 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.
46. C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
47. S. Rao, A. Badia, and D. Van Gucht. Processing Queries Containing Generalized Quantifiers. Technical Report TR 428, Computer Science Department, Indiana University, 1996.
48. B. Russell. *The Principles of Mathematics*. George Allen and Unwin, London, 1903.
49. 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.
50. G. Sandu and J. Väänänen. Partially Ordered Connectives. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 38:361–372, 1992.

51. 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.
52. J. Spivey. *The Z Notation: A Reference Manual*. Addison Wesley, Harlow, 1992.
53. I. Stewart. Logical Characterizations of Bounded Query Classes I: Logspace Oracle Machines. *Fundamenta Informaticae*, 18:65–92, 1993.
54. I. Stewart. Logical Characterizations of Bounded Query Classes II: Polynomial-Time Oracle Machines. *Fundamenta Informaticae*, 18:93–105, 1993.
55. I. Stewart. Incorporating generalized quantifiers and the least fixed point operator. In *Proc. Computer Science Logic*, number 832 in LNCS, pages 318–333, 1994.
56. I. Stewart. Complete problems for monotone NP. *Theoretical Computer Science*, 145:147–157, 1995.
57. I. Stewart. Logical Descriptions of monotone NP problems. *Journal of Logic and Computation*, 4:337–357, 1995.
58. L. J. Stockmeyer. The Polynomial-Time Hierarchy. *Theoretical Computer Science*, 3:1–22, 1977.
59. J. Väänänen. Generalized Quantifiers. *Bulletin of the EATCS*, 54, June 1997.
60. M. Vardi. Complexity of Relational Query Languages. In *Proceedings 14th STOC*, pages 137–146, San Francisco, 1982.
61. H. Veith. Succinct Representation, Leaf Languages, and Projection Reductions. *Information and Computation*, 142(2):207–236, 1998.
62. H. Vollmer. A General Quantifier Concept in Computational Complexity Theory. In J. Väänänen, editor, *Generalized Quantifiers and Computation: Proceedings of the ESSLLI Workshop 1997*, Lecture Notes in Logic. Springer, 1998. This volume.
63. K. Wagner. Bounded Query Classes. *SIAM Journal of Computing*, 19(5):833–846, 1990.
64. W. Walkoe. Finite Partially-Ordered Quantification. *Journal of Symbolic Logic*, 35:535–555, 1970.
65. J. Wordsworth. *Software Engineering with B*. Addison Wesley, 1996.