VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

Department of Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 471 5328
Fax: +64 4 495 5232
Internet: Tech.Reports@comp.vuw.ac.nz

Arithmetic and Aggregate Operators in Deductive Object-Oriented Databases

Gillian Dobbie and Rodney Topor

Technical Report CS-TR-96/15

Abstract

We extend our previous mathematical foundation for a deductive object-oriented language called Gulog to include arithmetic and aggregate operators. We describe the semantics of the extended language and present a corresponding query evaluation procedure.

Publishing Information

This paper was presented at the International Workshop on Logic in Databases, July 1–2, 1996.

Author Information

Gillian Dobbie
Department of Computer Science
Victoria University of Wellington
PO Box 600
New Zealand
gill@comp.vuw.ac.nz

Rodney Topor
School of Computing and Information Technology
Griffith University
Nathan, Qld 4111
Australia
rwt@cit.gu.edu.au

1 Introduction

1.1 The problem

Arithmetic operators [3] and aggregate operators [14] have been added to realtional and deductive databases and logic programming in order to make the language more expressive. The semantics of these operators in logic programming is now understood [14]. This work has been broadened by Harel in his discussion of recursive languages, which are a generalization of languages with arithmetic [5]. However, it is not well-understood how arithmetic and aggregate operators interact with the features of deductive object-oriented languages. Mathematical foundations have been defined for deductive object-oriented languages [1, 7]. However, none of these include arithmetic and aggregate operators in their definitions. Some practical deductive object-oriented database systems like Coral++ [12], and LDL++ [15] do include arithmetic and aggregates operators, but the authors do not investigate how arithmetic and aggregate operators interact with the object-oriented features.

Previously, we provided a mathematical foundation for a range of deductive and object-oriented features by defining the semantics of a language called Gulog [4]. However, there are queries which are quite natural that cannot be expressed in Gulog. For example, it is not possible to count the number of students in a class, say COMP101, or count the number of students who have passed an exam, say the final exam for class COMP101. These kinds of queries are possible if we include arithmetic and aggregate operators in the language. In this paper we show

- how a deductive object-oriented database language can be extended to include arithmetic and aggregate operators
- how arithmetic and aggregate operators interact with other features in deductive object-oriented database languages
- what the meaning of queries and programs is in this language and
- how queries can be evaluated in deductive object-oriented databases?

1.2 Main results

We describe a mathematical foundation for a deductive object-oriented language with arithmetic and aggregate operators. We do this by extending the syntax and semantics of a language called Gulog to include these operators. A naive extension can lead to ill-defined programs, so we first identify simple syntactic restrictions that ensure programs are simple, and hence have clear, intuitive meanings. We then extend the preferred model semantics of Gulog to allow for arithmetic and aggregate operators, and prove that every simple program has a unique preferred model and hence an unambiguous (and natural) meaning. Finally, we present a query evaluation procedure for simple programs. The procedure consists of a translation of a well-defined Gulog program with arithmetic and aggregate operators to a modularly stratified Datalog program with arithmetic and aggregate operators, followed by the application of a standard query evaluation procedure for modularly stratified Datalog programs. We prove that this procedure is sound with respect to the preferred model semantics for well-defined programs.

These results contribute to the theory of database languages that involve deductive, object-oriented and arithmetic components.

2 Technical details

Researchers have considered how arithmetic and aggregate operators interact with features of deductive databases. The most comprehensive coverage to date is [14]. There has been no work done specifically considering how these operators interact with features of object-oriented databases. This is the area covered we cover in this paper. Our work follows from that in the deductive database area.

In this section, we describe how Gulog can be extended in order to investigate the interaction of arithmetic and aggregate operators with features of object-oriented databases. In Section 2.1, we outline the syntax of a deductive object-oriented language with arithmetic and aggregate operators. Then, in Section 2.2, we describe a syntactic restriction which defines programs that have an intuitive meaning. We call these programs, simple programs. In Section 2.3, we provide a model semantics for simple programs based on preferred models and prove that every simple program has a unique preferred model. In Section 2.4, we discuss a query evaluation procedure for programs in this language, and prove that the procedure is sound with respect to the preferred model semantics. The query evaluation procedure that we describe involves translation. Although, a direct implementation is also possible. We present the translation to Datalog as a convenience, to allow us to build on correctness results for Datalog evaluation. To conclude, in Section 2.5, we describe the kinds of queries that cannot be expressed even when Gulog is extended with arithmetic and aggregate operators.

2.1 Syntax

In this section, we describe how arithmetic and aggregate operators can be included in Gulog [4]. In order to make the results clearer, we do not consider negation or multiple inheritance in this paper. The results can be straightforwardly extended to cover these concepts.

We start by illustrating the syntax of Gulog with an example.

Example 2.1 Clause 1 assigns the action *lower pressure* to a particular system failure (sf) if the pressure is high for that system failure.

$$\{x{:}sf\} \vdash x[action \twoheadrightarrow \text{“}lower\ pressure\text{”}] \leftarrow x[pressure \rightarrow high] \tag{1}$$

The variable typing $\{x{:}sf\}$ is separated from the rest of the clause by the symbol $\vdash$. Atom $\{x{:}sf\} \vdash x[action \twoheadrightarrow \text{“}lower\ pressure\text{”}]$ is the head of the clause. The body of the clause is $\{x{:}sf\} \vdash x[pressure \rightarrow high]$. The variable typing $\{x{:}sf\}$ tells us that in this clause the variable x has type sf. The method atom $\{x{:}sf\} \vdash x[action \twoheadrightarrow \text{“}lower\ pressure\text{”}]$ describes a multi-valued method *action* that has a value *lower pressure* for an object x. The atom $\{x{:}sf\} \vdash x[pressure \rightarrow high]$ describes a functional method *pressure* that has a value *high* for an object x.

Method atoms may take arguments. In multi-valued method 2, we describe a method *failure* that acts on objects of type *reactor* with an argument of type *date* and returns a value of type sf.

$$\{x{:}reactor, y{:}sf, d{:}date\} \vdash x[failure@d \twoheadrightarrow y].\ \square \tag{2}$$

In Gulog, the schema is separated from the program. The following example illustrates the kind of declarations allowed in the schema.

Example 2.2 Consider the following declarations.

$$r1{:}reactor \tag{3}$$

$$psf < sf \tag{4}$$

$$reactor[parts \Rightarrow\!\!\!\!\!\Rightarrow part] \tag{5}$$

$$measurements(reactor, pressure\_sensor) \tag{6}$$

Object declaration 3 declares an object $r1$ is of type *reactor*. Type lattice declaration 4 declares that psf (pressure system failure) is a subtype of sf (system failure). Multi-valued method declaration 5 declares a 0-ary multi-valued method *parts*, which when applied to objects of type *reactor* returns values of type *part*. The signature of 0-ary multi-valued method *parts* is $reactor \Rightarrow\!\!\!\!\!\Rightarrow part$. Finally predicate declaration 6 declares a 2-ary predicate *measurements* with signature $reactor \times pressure\_sensor$. $\Box$

We now describe how Gulog can be extended to include arithmetic and aggregate operators. Arithmetic and aggregate atoms are allowed in the bodies of clauses. We illustrate the use of an aggregate atom in the following example. The syntax of aggregate atoms follows that introduced in [6].

Example 2.3 Clause 7 counts the number of failures that are recorded for each reactor and stores the reactor identifier and count in a relation $failure\_count$.

$$\begin{aligned} &\{x{:}reactor, y{:}date, z{:}sf, c{:}integer\} \vdash failure\_count(x, c) \leftarrow \\ &\quad group\_by(x[failure@y \twoheadrightarrow z], [x], c = count) \end{aligned} \tag{7}$$

First, we illustrate the result of a query and then we describe the syntax of the clause. If the query $\{y{:}reactor, z{:}integer\} \vdash \leftarrow failure\_count(y, z)$ was applied to a database containing

$$r1[failure@12Sept95 \twoheadrightarrow \text{“cooling system failure”}]$$

$$r2[failure@14Sept95 \twoheadrightarrow \text{“system failure”}]$$

$$r1[failure@20Sept95 \twoheadrightarrow \text{“condenser failure”}]$$

then the result would be $failure\_count(r1, 2), failure\_count(r2, 1)$.

In clause 7, the *group_by* atom in the body of the clause groups the $failure$ method on the object x to which it is applied and counts the number of $failure$s for each x. The aggregate operator *count* does not take an argument as it is the count of the number of $failure$s for each x. $\Box$

Arithmetic operators are modeled as predicate atoms. There are built-in predicates for equality, arithmetic comparisons, and arithmetic operators that operate on integers. Aggregate operators are *sum*, *count*, *min*, *max* and *average*. An aggregate atom is defined as follows: $\Gamma \vdash group\_by(A, [x_1, \ldots, x_n], y = f(x_1, \ldots, x_p, z_1, \ldots, z_q))$ is an aggregate atom where Γ is the variable typing for the variables in the atom, A is an atom involving variables $(x_1, \ldots, x_n, z_1, \ldots, z_m)$, $x_1, \ldots, x_n$ are the *group_by* variables, f is an aggregate operator, $(x_1, \ldots, x_p, z_1, \ldots, z_q)$ is a list of variables involving only some subset of the variables $(x_1, \ldots, x_n, z_1, \ldots, z_m)$. This list may be empty. Similarly, there may be no *group_by* variables. The variables $z_1, \ldots, z_m$ may not appear outside the aggregate atom. A ground instance of an aggregate atom is obtained by replacing the variables $x_1, \ldots, x_n, y$ by ground terms $x_{1_0}, \ldots, x_{n_0}, y_0$. The variables $z_1, \ldots, z_m$ are not instantiated in such a ground instance.

Example 2.4 In this example, we describe an aggregate atom and present a ground instance of that atom. Assume a set of declarations including $r1{:}reactor$.
The atom $A = \{y{:}integer, z{:}part\} \vdash group\_by(r1[parts \twoheadrightarrow z], [\,], y = count)$ gives the cardinality of the set of *parts* for reactor $r1$. The atom $group\_by(r1[parts \twoheadrightarrow z], [\,], 100 = count)$ is a ground instance of A. Note that variable z is not instantiated in this ground instance of A. $\Box$

Definition Given a variable typing Γ, atom $\Gamma \vdash A$, and atoms, arithmetic atoms or aggregate atoms $\Gamma \vdash B_1, \ldots, \Gamma \vdash B_n$, $\Gamma \vdash A \leftarrow B_1 \wedge \cdots \wedge B_n$ is an *aggregate clause*. Atom $\Gamma \vdash A$ is the *head atom* of the clause. Atoms $\Gamma \vdash B_1, \ldots, \Gamma \vdash B_n$ are the *body atoms*.

An aggregate program is a set of aggregate clauses and is associated with a set of declarations.

Definition An *aggregate program* with respect to a set of declarations D is a finite set of aggregate clauses with respect to the set of declarations, D.

2.2 Simple programs

To ensure that programs have well-defined semantics, we impose restrictions that ensure they are "stratified", functional methods are single-valued, and the variables in arithmetic atoms will have a finite number of values assigned to them.

The following example illustrates a program that does not have a well-defined semantics.

Example 2.5 Assume appropriate declarations including $r1{:}reactor$, and consider the following program.

$$\{x{:}reactor, z{:}integer\} \vdash x[total \twoheadrightarrow z] \leftarrow group\_by(x[total \twoheadrightarrow z], [x], z = count)$$

This program is attempting to group a set of methods while generating new elements in those sets. We claim that this program does not have an intuitive meaning and so does not have well-defined semantics. $\Box$

A program is inheritance-stratified if there are no cycles in the definition of a method or an aggregate atom. That is, no method depends on an inherited definition of the same method and no atom is defined in terms of itself in an aggregate atom.

Definition An aggregate program P with respect to a set of declarations D is *inheritance-stratified* (or *i-stratified*) if there exists a mapping μ from the set of ground atoms to the set of non-negative integers such that, for every ground instance $C\theta$ of every clause C in P with respect to D, where A is the head of the instance $C\theta$,

- $\mu(A') \leq \mu(A)$, where A' is an atom or arithmetic atom in the body of the clause instance $C\theta$,
- $\mu(B') < \mu(A)$ for every ground instance B' of A', where A' is the atom in a ground aggregation atom in $(\Gamma \vdash group\_by(A', [x_1, \ldots, x_n], y_0 = f(x_1, \ldots, x_p, z_1, \ldots, z_p)))\theta$ in the body of the ground clause $C\theta$,
- $\mu(B') < \mu(A)$ for every ground instance $C'\theta'$ that possibly overrides $C\theta$, where B' is an atom or arithmetic atom in the body of $C'\theta'$, or for every ground instance B' of A' where A' is the atom in a ground aggregation atom in $(\Gamma \vdash group\_by(A', [x_1, \ldots, x_n], y_0 = f(x_1, \ldots, x_p, z_1, \ldots, z_p)))\theta$ in the body of $C'\theta'$,
- $\mu(A') \leq \mu(A)$ for every ground instance $C'\theta'$ that possibly overrides $C\theta$, where A' is the head of $C'\theta'$.

The second point in the above definition is the additional restriction required in the presence of aggregates.

Example 2.6 Consider again the declarations and program in example 2.5. The i-stratification of this program requires:

$$\mu(r1[total \twoheadrightarrow z_0]) < \mu(r1[total \twoheadrightarrow z_0])$$

As expected, this program is not i-stratified.

Again assume appropriate declarations including $r1{:}reactor$, and now consider the following program.

$$\{x{:}reactor, y{:}string, z{:}integer\} \vdash x[total \twoheadrightarrow z] \leftarrow$$
$$group\_by(x[action \twoheadrightarrow y], [x], z = count)$$

This program groups the *action* methods and gives information to method *total*. We claim that this program does have an intuitive meaning and also has a well-defined semantics. The i-stratification of this program requires:

$$\mu(r1[total \twoheadrightarrow z_0]) < \mu(r1[action \twoheadrightarrow y])$$

As expected, this program is i-stratified. □

The class of simple aggregate programs is a generalization of the class of simple Gulog programs. It is necessary to extend the syntactic condition which ensures that functional methods are single-valued to ensure that a finite number of values will be assigned to variables in arithmetic atoms. Such an extension involves the definition of "finiteness dependency". The following definition and example are from [13].

Definition A *finiteness dependency* on a predicate R is a pair $S \to j$, where j and each element of S are argument positions of R. A database B *satisfies* a finiteness dependency $S \to j$ for a predicate R if, for every tuple t in R, the set of tuples $\{s[j] \mid s \in R$ and $s[S] = t[S]\}$ is finite (where $t[S]$ is the projection of t onto the positions in S.)

Example 2.7 The arithmetic relations = and + have the following finiteness dependencies.

$$= \ \{1\} \to 2, \{2\} \to 1$$
$$+ \ \{1,2\} \to 3, \{2,3\} \to 1, \{3,1\} \to 2$$

Consider $= \ \{1\} \to 2, \{2\} \to 1$. One way this can be read is that if the first argument of the = predicate has a finite domain then the second argument has a finite domain. Similarly, if the second argument has a finite domain then the first argument has a finite domain. Comparison relations do not have any finiteness dependencies. Each (finite) k-ary base relation has k dependencies $\emptyset \to j$, for $1 \leq j \leq k$. □

In [2], Abiteboul and Hull showed that compile-time consistency checking to ensure that functional methods are single-valued is undecidable. To ensure that functional methods really are single-valued, we impose the following condition on programs. We extend this condition to ensure that there is a finite number of results for queries involving arithmetic atoms.

Definition A variable z is *restricted* with respect to the terms $\{t, t_1, \ldots, t_n\}$ in the conjunction C if the following conditions hold.

- C contains a method atom $t'[m'@t'_1, \ldots, t'_k \to z]$ and each variable in t', $t'_1, \ldots, t'_k$ either occurs in $t, t_1, \ldots, t_n$ or is itself restricted with respect to $\{t, t_1, \ldots, t_n\}$ in the remainder of C. For example, the variable z is restricted with respect to $\{a, x\}$ in the conjunction $a[m'@x \to y] \wedge a[m''@y \to z]$.

- C contains an arithmetic atom $p(t'_1, \ldots, t'_n)$, some t'_j is z, p has a finiteness dependency $S \to j$, and every variable in $\{t'_i \mid i \in S\}$ is restricted with respect to $\{t, t_1, \ldots, t_n, z\}$ in the remainder of C.

That is, a variable is restricted with respect to a set of terms in a conjunction if that variable will have a finite number of values assuming the set of terms has a finite domain.

We are now able to define the class of programs in which functional methods are single-valued and the variables in arithmetic atoms have a finite number of values. We call programs in this class well-defined programs.

Definition An aggregate program P with respect to a set of declarations D is *well-defined* if the following conditions hold:

- For each clause $\Gamma \vdash t[m@t_1, \ldots, t_n \to t'] \leftarrow C$ in P with respect to D, t' is either an object symbol or is a variable that is restricted with respect to $\{t, t_1, \ldots, t_n\}$ in C.
- For each functional method m with signature $\tau \times \tau_1 \times \cdots \times \tau_n \to \tau'$, and for each type $\sigma \leq \tau$, program P with respect to D does *not* contain two clauses

  $$\Gamma \vdash t[m@t_1, \ldots, t_n \to s] \leftarrow C$$
  $$\Gamma \vdash t'[m@t'_1, \ldots, t'_n \to s'] \leftarrow C'$$

  such that $\Gamma \vdash t{:}\sigma$, $\Gamma \vdash t'{:}\sigma$, and atoms $t[m@t_1, \ldots, t_n \to x]$ and $t'[m@t'_1, \ldots, t'_n \to x']$ (where x and x' are new variables) are unifiable. This together with the first condition ensures that functional methods really are single-valued.
- For each clause $\Gamma \vdash A \leftarrow \cdots \wedge B \wedge \cdots$ where B is an arithmetic atom, every variable t_i in B is restricted with respect to $\emptyset$ in the body of the clause.

The third point is the additional restriction required in the presence of arithmetic. It restricts the class of well-defined programs to those in which arithmetic atoms have a finite domain.

Example 2.8 This example presents two programs that aren't well-defined and one that is well-defined.

First, consider the aggregate clause:

$$\{x{:}integer, y{:}integer\} \vdash p(x, y) \leftarrow x + y = 5$$

We would not expect this clause to be well-defined because there are an infinite number of values that can be assigned to x and y to make the body of the clause true. Following our intuition, it is not suprising that x and y are not restricted with respect to $\emptyset$ in the body of this clause. Thus this program is not well-defined.

Secondly, consider the clause:

$$\{x{:}integer, y{:}integer\} \vdash o[m \to x] \leftarrow x + y = 5$$

Again, we would not expect this clause to be well-defined because there are an infinite number of values that can be assigned to x and y to make the body of the clause true. It is not suprising that x is not restricted with respect to $\{o\}$ in the body of the clause.

Finally, consider the aggregate clause:

$$\{x{:}integer, y{:}integer\} \vdash p(x, y) \leftarrow x + y = 5 \wedge y = 3$$

We now find that x and y are restricted with respect to $\emptyset$ in the body of this clause. Thus this program is well-defined. $\Box$

The following class of programs is particularly important because programs in this class have an intuitive meaning. That is, the interaction of the arithmetic and aggregate operators with other features of deductive object-oriented databases is well understood in this class of programs.

Definition A program P with respect to a set of declarations D is called *simple* if P with respect to D is i-stratified and well-defined.

In the next section we show that every simple program has a unique preferred model that defines a natural semantics, and a simple evaluation procedure that is sound with respect to the resulting preferred model semantics.

2.3 Semantics

In this section, we describe the declarative semantics of programs. As in logic programming, the meaning of a program or database can be given by a preferred model. We first extend the definition of truth value to describe values for aggregate atoms.

Definition Let I be an interpretation, L be an atom, and $\mathcal{T}$ a term assignment (with respect to Γ). Then an atom, arithmetic atom or aggregate atom, A, is given a *truth value* in I (with respect to $\mathcal{T}$ and Γ) as follows:

- If A is $\Gamma \vdash p(t_1, \ldots, t_n)$, and $t'_1, \ldots, t'_n$ are the term assignments of $t_1, \ldots, t_n$ (with respect to $\mathcal{T}$ and Γ), then A is true in I (with respect to $\mathcal{T}$ and Γ) if $p(t'_1, \ldots, t'_n) \in I$.
- If A is $\Gamma \vdash t[m@t_1, \ldots, t_k \rightarrow t_n]$, and $t', t'_1, \ldots, t'_k, t'_n$ are the term assignments of $t, t_1, \ldots, t_k, t_n$ (with respect to $\mathcal{T}$ and Γ), then A is true in I (with respect to $\mathcal{T}$ and Γ) if $t'[m@t'_1, \ldots, t'_k \rightarrow t'_n] \in I$.
- If A is $\Gamma \vdash t[m@t_1, \ldots, t_k \twoheadrightarrow s]$, and $t', t'_1, \ldots, t'_k, s'$ are the term assignments of $t, t_1, \ldots,$ t_k, s (with respect to $\mathcal{T}$ and Γ), then A is true in I (with respect to $\mathcal{T}$ and Γ) if $t'[m@t'_1, \ldots, t'_k \twoheadrightarrow s'] \in I$.
- If A is $\Gamma \vdash group\_by(A'(t, t_1, \ldots, t_n), [t], y = f(t, t_1, \ldots, t_n))$ and $t', t'_1, \ldots, t'_n$ are the term assignments of $t, t_1, \ldots, t_n$ (with respect to $\mathcal{T}$ and Γ), then A is true in I (with respect to $\mathcal{T}$ and Γ) if $A'(t', t'_1, \ldots, t'_n) \in I$ and $f(S)$ is defined for the set $S = \{(t', t'_1, \ldots, t'_n) \mid A(t', t'_1, \ldots, t'_n) \in I\}$ and $y = f(S)$.

Example 2.9 Suppose $I = \{r1[parts \twoheadrightarrow valve], r1[parts \twoheadrightarrow heat\_exchanger]\}$. Then the following ground instance is true in I:

$group\_by(r1[parts \twoheadrightarrow x], [\,], 2 = count)$. $\Box$

A simple program may have more than one minimal model, so we define a priority relation that is used to identify preferred models of a program. As in [8], we can define a priority relationship between ground atoms (using possible overriding and aggregation), and use this to define a preference relationship between models. As in [8], we prefer models in which there are fewer occurrences of higher priority atoms. We say a model of a program P with respect to a set of declarations D is *preferred* if no models of P are preferable to it.

Definition Let P be a program with respect to a set of declarations D. We define an *priority relation* $<_p$ based on P between ground atoms with respect to D. We write $a \leq_p b$ to denote $a <_p b$ or $a = b$. We define the relation $<_p$ by the following rules, for every ground instance $C\theta$ of a clause C in P with respect to D:

- $A \leq_p A'$, where A is the head of $C\theta$, and A' is an atom or arithmetic atom in the body of $C\theta$,
- $A <_p B'$ for every ground instance $C'\theta'$ of C' in P with respect to D, that possibly overrides $C\theta$, where A is the head of $C\theta$ and B' is an atom or arithmetic atom in the body of $C'\theta'$,
- $A \leq_p A'$ for every ground instance $C'\theta'$ of C' in P with respect to D, that possibly overrides $C\theta$, where A is the head of $C\theta$ and A' is the head of $C'\theta'$, and
- $A <_p A'$ where A is the head of C and A' is the ground instance of an atom in the ground aggregation atom in the body of C.

and for all ground atoms A, B, C and F:

- if $A \leq_p B$ and $B \leq_p C$, then $A \leq_p C$,
- if $A \leq_p B$ and $B <_p C$ (respectively, $F <_p A$) then $A <_p C$ (respectively, $F <_p B$).

We use the definition in [4] for preferred models and depict that model M_1 is preferred over M_2 by $M_1 \ll M_2$. The following result holds.

Theorem 2.1 *Let P be a simple program with respect to a set of declarations D. Then P with respect to D has exactly one preferred model, which we denote M_P. For every other model M, we have $M_P \ll M$.*

The proof of this theorem follows from that in [4], which in turn followed from Przymusinski's results for stratified logic programs.

2.4 Query evaluation

We now describe an evaluation procedure for a query in this language by translating the program and query to Datalog with aggregates. Because this language is a generalization of Gulog, we can simply extend the translation described for Gulog in Chapter 5 in [4]. In this translation, method atoms are translated to predicates, and overriding is dealt with by adding extra rules which indicate whether a definition from lower in the type hierarchy applies or not. To deal with inheritance, we also add extra rules that describe a type and its subtypes.

In the extended translation from a program written in the language described in this paper to Datalog, an aggregate atom $\Gamma \vdash group\_by(A, [x_1, \ldots, x_n], y = f(x_1, \ldots, x_p,\ z_1, \ldots, z_q))$ is translated to $\Gamma' \wedge group\_by(A', [x_1, \ldots, x_n], y = f(x_1, \ldots, x_p,\ z_1, \ldots, z_q))$, where Γ' is the translation of Γ and A' is the translation of A. This extended translation treats aggregate atoms in a similar way to the way negative literals are treated in [4].

We now describe a query evaluation procedure that finds the computed answers for a query Q and a program P with respect to a set of declarations D. The computed answer is a set of substitutions. Suppose θ is a computed answer for a query Q and a program P with respect to a set of declarations D, we write θ is a computed answer for $P \cup \{Q\}$. In [4], we proved that if a simple Gulog program P with respect to D is i-stratified then P with respect to D is m-stratified. (m-stratification is like modular stratification.) Again in [4], we proved that if P with respect to D is m-stratified then its translation P' is modularly stratified, and that the well-founded model of P' is equivalent to the associated model of P with respect to D. Ross proves, in [11], that any modularly stratified program can be rewritten as a left-to-right modularly stratified program. Lastly, Ramakrishnan et al. provide a procedure that evaluates programs with left-to-right modularly stratified negation and aggregation in [10] that is sound with respect to well-founded semantics.

The following soundness result then holds.

Theorem 2.2 *Let P with respect to a set of declarations D be a simple program, M_P be the unique preferred model of P with respect to D, Q be a query, and θ be a computed answer for $P \cup \{Q\}$. Then $Q\theta$ is true in the unique preferred model M_P.*

2.5 Expressibility

It is important to realize not only what can be expressed but also what cannot be expressed in a language. We now discuss what can and cannot be expressed in Gulog with arithmetic and aggregate operators. The addition of arithmetic and aggregate operators to Gulog allows queries that involve counting to be expressed for a first time. With counting it is possible to express queries such as the one in the following example that determines if the size of a relation is even or odd.

Example 2.10 Let $p(0)$ represent true and $p(1)$ represent false. Consider a query that returns true if the size of a relation q is even and false otherwise. This is expressed in the following clause:

$$\begin{aligned}&\{x{:}integer, y{:}integer, z{:}integer\} \vdash p(x) \leftarrow \\ &\qquad group\_by(q(z), [\,], y = count) \wedge x = y \ mod\ 2. \ \Box\end{aligned}$$

Some queries cannot be expressed even with arithmetic and aggregate operators. Consider a query whose input is a relation, r, which has n tuples, and whose output is the cross product of r with itself, n times. As the width of the output relation depends on the data, it is not possible to express such queries in Gulog with arithmetic and aggregate operators.

References

[1] S. Abiteboul. Towards a deductive object oriented database language. *Data and Knowledge Engineering*, 5:263–287, 1990.

[2] S. Abiteboul and R. Hull. Data-functions, datalog and negation. In *Proc. of the ACM SIGMOD International Conference on the Management of Data*, pages 143–153, 1988.

[3] A. Chandra and D. Harel. Computable queries for relational data bases. *Journal of Computer and System Sciences*, pages 156–178, 1980.

[4] G. Dobbie. *Foundations of Deductive Object-Oriented Database Systems.* Phd thesis, University of Melbourne, 1995.

[5] D. Harel and T. Hirst. Completeness results for recursive data bases. In *Proceedings of the Twelth ACM PODS Symposium on Principles of Database Systems*, 1993.

[6] D. B. Kemp and P. J. Stuckey. Semantics of logic programs with aggregates. In V. Saraswat and K. Ueda, editors, *Proc. 1991 International Logic Programming Symposium*, pages 387–401, San Diego, USA, 1991.

[7] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. Technical Report 90/14 (revised), Department of Computer Science, State University of New York at Stony Brook, 1990. Further revised as Technical Report 93/06, April 1993.

[8] T. C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, 1988. Further revised as [9].

[9] T. C. Przymusinski. Every logic program has a natural stratification and an iterated fixed point model. In *Proceedings of the Eighth ACM PODS Symposium on Principles of Database Systems*, pages 11–21, 1989.

[10] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Controlling the Search in Bottom-Up Evaluation. In *Proc. of the Joint Int. Conference and Symposium on Logic Programming*, pages 273–287, Washington DC, November 1992.

[11] K. A. Ross. Modular stratification and magic sets for DATALOG programs with negation. In *Proc. of the ACM SIGMOD International Conference on the Management of Data*, pages 161–171, 1990.

[12] D. Srivastava, R. Ramakrishnan, P. Seshadri, and S. Sudarshan. Coral++: Adding object-orientation to a logic database language. In *Proc. of the 19th VLDB Conference*, Dublin, Ireland, 1993.

[13] R. W. Topor. Safe database queries with arithmetic relations. In *Proc. 14th Australian Computer Science Conference*, pages 02–1–02–13, 1991.

[14] A. Van Gelder. Foundations of aggregation in deductive databases. In S. Ceri, K. Tanaka, and S. Tsur, editors, *Proceedings of the Third International Conference on Deductive and Object-Oriented Databases*, 1993.

[15] C. Zaniolo, N. Arni, and K. Ong. Negation and aggregates in recursive rules: the LDL++ approach. In S. Ceri, K. Tanaka, and S. Tsur, editors, *Proceedings of the Third International Conference on Deductive and Object-Oriented Databases*, pages 204–221, Arizona, USA, 1993.