VICTORIA UNIVERSITY OF WELLINGTON

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

Representing inheritance and overriding in Datalog

Gillian Dobbie and Rodney Topor

Technical Report CS-TR-94/11
24 May, 1994.

Abstract

We describe the relationship between inheritance with overriding in an object-oriented database language, Gulog, and negation. We present a translation from Gulog programs to Datalog programs with negation and show that evaluating queries in the translation yields correct answers with respect to the original program.

Publishing Information

This paper will appear in Computers and AI, number 2, 1994. The ideas were first presented at the Deductive Database Workshop in conjunction with ICLP'93

Author Information

Rodney Topor's address is School of Computing and Information Technology, Griffith University, Nathan, Qld 4111, Australia. Email: rwt@cit.gu.edu.au

1 Introduction

It is now widely believed that object-oriented database systems are required for many non-traditional database applications, and several commercial object-oriented database systems have been implemented and successfully applied. However, theoreticians remain concerned that such object-oriented database systems lack the kind of simple mathematical foundation possessed by relational or deductive database systems. Such a foundation is required for defining the semantics of databases and queries, for database design, and for query optimization. In addition, the commercial object-oriented database systems mentioned above lack the kind of deductive component that has proved so useful for views in relational database systems and for more general derived relations in deductive database systems.

There have been several attempts to present a simple mathematical foundation for object-oriented database systems with deduction [1, 2, 3, 4, 6, 9, 13, 14, 15]. The best developed of these is the frame logic (or F-logic) proposed by Kifer et al. [13]. This theory consists of a logic with a "sound and complete proof theory" that captures many aspects of common object-oriented database systems. However, the proof theory given in [13] does not address the nonmonotonic aspects of object-oriented database systems introduced by overriding, i.e., the combination of method redefinition in subclasses and dynamic binding. McCabe's *L&O* [15] is also a deductive object-oriented language with overriding. The semantics of *L&O* programs is given by translating *L&O* programs to logic programs and thus provides little direct insight into the relationship between inheritance, overriding and deduction. Query processing is also described using a translation to Prolog. Another significant theoretical language is IQL, described by Abiteboul and Kanellakis in [3]. IQL treats inheritance as a specialisation of union types, but, as in many logics of object-oriented languages, has no concept of overriding. In [4], Abiteboul et al. describe another language, $datalog^{meth}$, and three different translations to Datalog which capture the semantics of the langauge with monotonic, static and dynamic inheritance respectively. With monotonic inheritance there is no overriding, inherited values are accumulated. Static inheritance is like inheritance in procedural languages, overriding occurs independent of the value of the body of the clause. With dynamic overriding, a clause overrides only if the body of the clause is true. They introduce views into the language using virtual classes and show how dynamic inheritance can be simulated using static inheritance and views. They extend $datalog^{meth}$ to $datalog^{meth,flex}$ which allows them to reason about the class hierarchy.

In [11], we defined a simple language suitable for representing typical deductive object-oriented databases. Our aim was to study the interaction of inheritance, overriding and deduction in the simplest possible context. Distinguishing features of our language are that it separates schema declarations from data definitions, and that it allows methods to be overridden on specific instances of a subclass, thus generalizing other deductive object-oriented languages. In [11], we defined the language, presented a declarative semantics based on Przymusinski's perfect model semantics for logic programs, gave bottom-up and top-down query evaluation procedures, and proved their soundness. We consider the introduction of set-valued methods, multiple inheritance, and negation in [10].

In this paper we discuss the relationship between inheritance with overriding in Gulog programs and negation in Datalog programs[1]. We have extended the paper presented at the Deductive Database workshop [12], to include set-valued methods, negation and multiple inheritance. We present a translation from Gulog programs into Datalog programs, prove that every stratified Gulog program P is translated into a stratified Datalog program P', and prove that the standard model of P' includes the standard model of P. We then generalize this result to a larger class of programs and show that evaluating (translated) queries in

[1]Throughout, when we say Datalog, we mean Datalog with negation

P' yields correct answers (with respect to P). This thus provides an alternative evaluation procedure for more general Gulog queries.

Inheritance in Gulog is the same as dynamic inheritance in [4] and our translation to Datalog is like their dynamic rewriting. Our work differs from theirs in the following ways. We distinguish between set-valued and functional methods, and allow a type to have more than one parent in the type hierarchy. We also identify two interesting classes of programs which correspond to locally stratified and modularly stratified Datalog programs.

This paper is organised as follows. In Section 2 we introduce Gulog, informally describing the syntax and semantics. In Section 3 we provide a formal description of the semantics of Gulog. Section 4 describes the translation from a Gulog program to a Datalog program. In Section 5 we investigate the relationships between the original Gulog program and its translation. In Section 6 we identify a more general class of programs and study their properties. Section 7 reviews the results of the paper, and investigates how extensions to the language may affect these results.

2 Summary of Gulog

In this section we informally describe the syntax and semantics of Gulog. Gulog can be considered as either a deductive object-oriented logic (cf. F-logic) or as a deductive object-oriented database programming language. In the latter interpretation, each program in the language, which includes both facts and rules, corresponds to a database. Similarly, goals in the language correspond to queries. See [11] for a more formal description of the main features of Gulog and [10] for the set, multiple inheritance and negation extensions.

The syntax of Gulog is based on that of Login [5] and is similar to the syntax of F-logic [13]. However, in Gulog, schema declarations are separate from data definitions, we distinguish between objects and types (or classes), and method overriding can be defined on specific instances of a subtype.

To introduce Gulog, we use an example which models part of a system used in the diagnosis of failures in a system of reactors [7]. In this system sensors measure flow, temperature and pressure. If a sensor reading is "abnormal", a failure has occurred. The part of the system which we describe records failures in a group of reactors. Each reactor comprises a set of parts, and we want to record what kind of failure has occurred on a particular day for each reactor. In Gulog we use the following declaration to specify the signature of type *reactor*:

$$reactor[parts \Rrightarrow \{part\}; failure@date \Rrightarrow \{sf\}]. \tag{1}$$

In this declaration, symbols *reactor*, *part*, *date*, and sf (system failure) denote types; symbols *parts* and *failure* denote set-valued methods. The signature of method *parts* is $reactor \twoheadrightarrow part$. The signature of method *failure*, which has one argument is $reactor \times date \twoheadrightarrow sf$. A method can be applied to any object in the type on which it is declared, or in any subtype of that type. For example the method *failure* can be applied to any object of type *reactor*, or in a subtype of *reactor*. Its argument must be of type *date*, or a subtype of *date*. The value of this method is a set of type sf, or a subtype of sf. The classification hierarchy of the failures is given in Figure 1.

In Gulog, this hierarchy is written:

$$psf < sf \tag{2}$$

$$csf < sf \tag{3}$$

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

$$vcf < psf \tag{5}$$

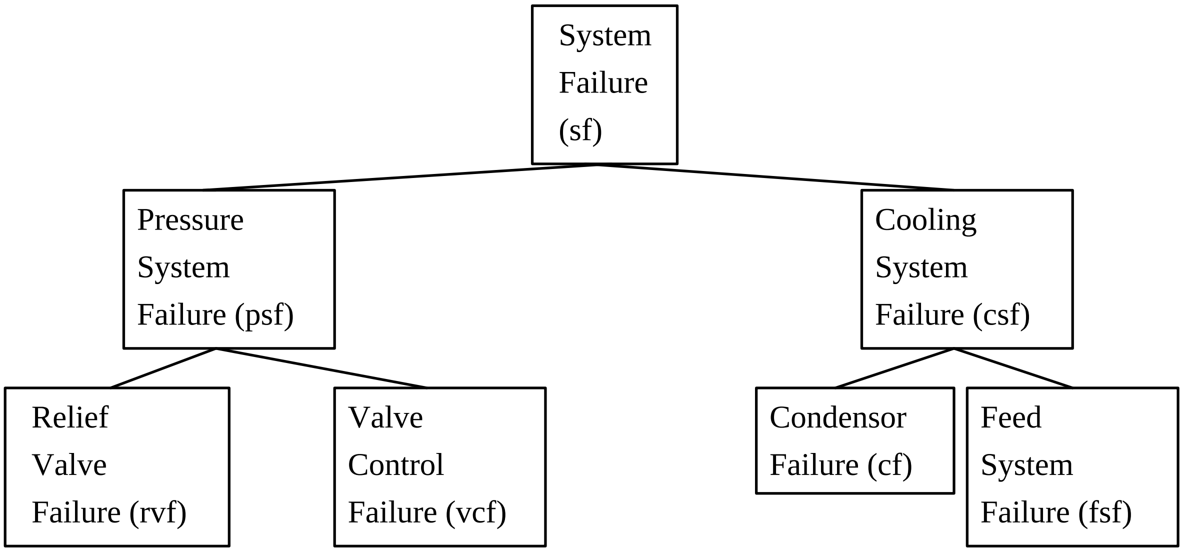

Figure 1: Failure Classification Hierarchy

$$cf < csf \tag{6}$$
$$fsf < csf. \tag{7}$$

Declaration 2 states that type psf is a subtype of type sf (and sf is a supertype of psf). The type psf thus inherits methods from type sf.

In Gulog we can also represent multiple inheritance, where a type can have more than one parent in the type hierarchy. To illustrate this, we include a type $total\_failure$ (tf), to which an object belongs if all measurements are abnormal. In Gulog, this inheritance hierarchy is represented by declarations 2 to 7 and

$$tf < rvf \tag{8}$$
$$tf < vcf \tag{9}$$
$$tf < cf \tag{10}$$
$$tf < fsf. \tag{11}$$

The relation $<$ is a partial order on the type symbols in the declarations.

To indicate that an object is an instance of a type in Gulog, we write:

$$r1 : reactor. \tag{12}$$

In this declaration symbol $reactor$ denotes a type; and symbol $r1$ denotes an object. Object $r1$ is an instance of type $reactor$. We say $r1$ belongs to type $reactor$ or is of type $reactor$.

The measurements from the sensors are presented to the system in a predicate $measurements$. The arguments of $measurements$ are of type (or a subtype of) *reactor, pressure_sensor, temperature_sensor, relief_valve_sensor, valve_control_sensor, condensor_sensor, feed_system_sensor, sf.* The Gulog declaration is:

$$\begin{aligned} &measurements(reactor, pressure\_sensor, temperature\_sensor, relief\_valve\_sensor,\\ &\quad valve\_control\_sensor, condensor\_sensor, feed\_system\_sensor, sf). \end{aligned} \tag{13}$$

The signature of predicate $measurements$ is $reactor \times pressure\_sensor \times temperature\_sensor \times relief\_valve\_sensor \times valve\_control\_sensor \times condensor\_sensor \times feed\_system\_sensor \times sf$.

We have described the schema declarations. The data is defined by a set of clauses. A program or database (with respect to a set of declarations) is a finite set of clauses. Each object in a program has a single type throughout the program, whereas each variable may be declared to have a different type in each clause in which it occurs. In the following clause:

$$\{x:reactor, y:sf, d:date\} \vdash failure(x,d,y) \leftarrow x[failure@d \twoheadrightarrow y] \tag{14}$$

$\{x:reactor, y:sf, d:date\}$ is a variable typing. The symbol $\vdash$ separates the variable typing from the rest of the clause. The typed atom $\{x:reactor, y:sf, d:date\} \vdash x[failure@d \twoheadrightarrow y]$ is a typed method atom. The typed atom $\{x:reactor, y:sf, d:date\} \vdash failure(x,d,y)$ is a typed predicate atom. Every typed functional (resp. set-valued) method atom has the form $\Gamma \vdash t[m@t_1,\ldots,t_n \rightarrow t']$ (resp. $\Gamma \vdash t[m@t_1,\ldots,t_n \twoheadrightarrow t']$), where Γ is the variable typing of the variables in the atom $t[m@t_1,\ldots,t_n \rightarrow t']$ (resp. $t[m@t_1,\ldots,t_n \twoheadrightarrow t']$), $\Gamma \vdash t{:}c$, $\Gamma \vdash t_i{:}c_i$ (for $1 \leq i \leq n$), and $\Gamma \vdash t':c'$ are terms (variables or object symbols) where t has type c, t_i has type c_i, and t' has type c', m is a method symbol, and the terms are consistent with the signature of m. Every typed predicate atom has the form $\Gamma \vdash p(t_1,\ldots,t_n)$, where p is a predicate symbol and each $\Gamma \vdash t_i : c_i$ is a term consistent with the signature of p. Predicate atoms allow us to represent both base and derived relations in our language, and provide a convenient method of representing the answers to queries. Clause 14 defines predicate $failure$ of arity 3.

Recall declarations 2 to 7 and consider clause 15 which defines functional method *problem* when applied to any object of type sf:

$$\{y:sf\} \vdash y[problem \rightarrow \text{“system failure”}]. \tag{15}$$

By inheritance the method *problem* also returns the value “system failure” when applied to any object belonging to a subtype of sf unless the method *problem* has been overridden for that object. Clause 16 defines functional method *problem* to return a value “temperature high” for all objects of type csf provided the method *temp* of that object has value $high$:

$$\{y:csf\} \vdash y[problem \rightarrow \text{“temperature high”}] \leftarrow y[temp \rightarrow high]. \tag{16}$$

Clause 16 thus overrides clause 15 for objects of type csf whose method *temp* has value $high$. However if method *temp* of some object of type csf were *low* (not $high$), then the object would inherit the value for the method *problem*, with value “system failure” from type sf.

Negated atoms are allowed in the body of a clause. Clause 17 is used to validate the data produced. A temperature sensor is recognised as malfunctioning if there is a condensor failure and the temperature is normal.

$$\{y:cf\} \vdash y[malfunction \rightarrow temp\_sensor] \leftarrow \neg y[temp \rightarrow high] \wedge \neg y[temp \rightarrow low] \tag{17}$$

The method $malfunction$ has the value *temp_sensor* when applied to an object of type cf, if the method *temp* when applied to the same object does not return the value $high$ and does not return the value *low*. A literal is an atom or a negated atom. We call a negated atom a negative literal.

With multiple inheritance, the value of a method may be inherited from more than one parent. This can cause an ambiguity. There must be a least upper bound of the parents of the type which has more than one parent. For example, let tf be a subtype of types rvf, vcf, cf, and fsf, as described in declarations 8 to 11. In this case the least upper bound is sf. Currently, to simplify our presentation, we prohibit programs where a method is defined on any of the paths from the common ancestor to the common child in the hierarchy graph.

A goal or query is a clause with an empty head:

$$\{x:reactor, y:sf\} \vdash \leftarrow x[failure@17May \twoheadrightarrow y]. \tag{18}$$

Gulog contains the essential features of objects, types, functional methods, set-valued methods, inheritance, overriding, multiple inheritance, negation and deduction. When atoms are restricted to predicate atoms, the language becomes a form of typed Datalog.

We are interested in a class of programs called "simple" programs because each simple program has a natural semantics defined by a unique preferred model, and because there are bottom-up and top-down procedures which compute this model [10]. These programs have an intuitive semantics. Before defining simple programs we define a form of stratification which restricts the combination of deduction and overriding. We also define "well-defined" programs in which we require functional methods to be single-valued, and methods to be unambiguously defined with respect to multiple inheritance.

Thus, we start by defining "possible overriding".

Definition Let P be a program, C be $\Gamma \vdash t[m@t_1, \ldots, t_k \rightarrow t'] \leftarrow B$, (resp. $\Gamma \vdash t[m@t_1, \ldots, t_k \twoheadrightarrow t'] \leftarrow B$) and C' be $\Gamma' \vdash s[m@s_1, \ldots, s_k \rightarrow s'] \leftarrow B'$ (resp. $\Gamma' \vdash s[m@s_1, \ldots, s_k \twoheadrightarrow s'] \leftarrow B'$), where $C, C' \in P$, $\Gamma \vdash t : \sigma$ is a term, $\Gamma' \vdash s : \tau$ is a term and $\sigma < \tau$. If there are ground instances $C\theta$ of C and $C'\theta'$ of C' such that $t\theta = s\theta'$, $t_i\theta = s_i\theta'$ for all i, $1 \leq i \leq k$, and $t'\theta \neq s'\theta'$ then clause $C\theta$ *possibly overrides* clause $C'\theta'$ with respect to P.

Example 2.1 Assume appropriate declarations including declarations 2 to 7, and $f1 : cf$. Consider the program P:

$$\{x : sf\} \vdash x[severity \twoheadrightarrow high] \leftarrow x[treatment \rightarrow \text{"don't know"}]$$
$$\{x : sf\} \vdash x[severity \twoheadrightarrow low] \leftarrow \neg x[treatment \rightarrow \text{"don't know"}]$$
$$\{x : csf\} \vdash x[severity \twoheadrightarrow high] \leftarrow x[treatment \rightarrow \text{"turn off condensor"}]$$
$$\{x : csf\} \vdash x[severity \twoheadrightarrow low] \leftarrow x[treatment \rightarrow \text{"tell supervisor"}]$$

Ground clause $f1[severity \twoheadrightarrow low] \leftarrow f1[treatment \rightarrow \text{"tell supervisor"}]$ possibly overrides $f1[severity \twoheadrightarrow high] \leftarrow f1[treatment \rightarrow \text{"don't know"}]$,
and ground clause $f1[severity \twoheadrightarrow high] \leftarrow f1[treatment \rightarrow \text{"turn off condensor"}]$ possibly overrides $f1[severity \twoheadrightarrow low] \leftarrow \neg f1[treatment \rightarrow \text{"don't know"}]$.
Note that $f1[severity \twoheadrightarrow low] \leftarrow f1[treatment \rightarrow \text{"tell supervisor"}]$ does not possibly override $f1[severity \twoheadrightarrow low] \leftarrow \neg f1[treatment \rightarrow \text{"don't know"}]$,
and $f1[severity \twoheadrightarrow high] \leftarrow f1[treatment \rightarrow \text{"turn off condensor"}]$ does not possibly override $f1[severity \twoheadrightarrow high] \leftarrow f1[treatment \rightarrow \text{"don't know"}]$. □

A program is inheritance-stratified if there are no cycles in the definition of a method. That is the definition of a method does not depend on an inherited definition of the same method.

Definition A program P 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,

- $\mu(A') \leq \mu(A)$, where A' is a positive literal in the body of the clause instance $C\theta$, and A is the head of the clause instance $C\theta$,
- $\mu(A') < \mu(A)$, where $\neg A'$ is a negative literal in the body of the clause instance $C\theta$, and A is the head of the clause instance $C\theta$,
- $\mu(B') < \mu(A)$ for every ground instance $C'\theta'$ which possibly overrides $C\theta$, where B' is a positive literal or $\neg B'$ is a negative literal in the body of $C'\theta'$ and A is the head of $C\theta$, and

- $\mu(A') \leq \mu(A)$ for every ground instance $C'\theta'$ which possibly overrides $C\theta$, where A' is the head of $C'\theta'$, and A is the head of $C\theta$.

Example 2.2 Consider program P given in Example 2.1. Using the definition of i-stratification for programs with only functional methods given in [11], where the head of a possibly overriding clause was mapped to a smaller number than the head of the possibly overridden clause, this program is not i-stratified. Based on the definition given above the mapping is:

$$\begin{array}{l}
\mu(f1[treatment \rightarrow \text{“tell supervisor”}]) \leq \mu(f1[severity \twoheadrightarrow low]) \\
\mu(f1[treatment \rightarrow \text{“turn off condensor”}]) \leq \mu(f1[severity \twoheadrightarrow high]) \\
\mu(f1[treatment \rightarrow \text{“don't know”}]) < \mu(f1[severity \twoheadrightarrow low]) \\
\mu(f1[treatment \rightarrow \text{“don't know”}]) \leq \mu(f1[severity \twoheadrightarrow high]) \\
\mu(f1[severity \twoheadrightarrow low]) \leq \mu(f1[severity \twoheadrightarrow high]) \\
\mu(f1[severity \twoheadrightarrow high]) \leq \mu(f1[severity \twoheadrightarrow low]) \\
\mu(f1[treatment \rightarrow \text{“tell supervisor”}]) < \mu(f1[severity \twoheadrightarrow high]) \\
\mu(f1[treatment \rightarrow \text{“turn off condensor”}]) < \mu(f1[severity \twoheadrightarrow low]).
\end{array}$$

As expected P is i-stratified. □

For simplicity, we disallow programs with ambiguities due to multiple inheritance in this paper. (We are currently investigating the semantics and evaluation of programs in which a static ordering is used to determine the inheritance order.) Suppose that τ is a direct subtype of σ_1 and σ_2, and that $\sigma_1 < \sigma$ and $\sigma_2 < \sigma$. We say that program P is *unambiguous* (wrt multiple inheritance) if every method m (of arity n) that can be applied to objects of type τ is not defined on σ'_1 nor σ'_2, where $\sigma_i \leq \sigma'_i < \sigma$, for $i = 1, 2$.

Example 2.3 Assume appropriate declarations including declarations 2 to 11. Consider the following program P:

$$\{x : sf\} \vdash x[severity \twoheadrightarrow high] \leftarrow x[treatment \rightarrow \text{“don't know”}] \tag{19}$$
$$\{x : csf\} \vdash x[severity \twoheadrightarrow high] \leftarrow x[treatment \rightarrow \text{“turn off condensor”}] \tag{20}$$
$$\{x : tf\} \vdash x[severity \twoheadrightarrow high] \leftarrow x[treatment \rightarrow \text{“tell supervisor”}] \tag{21}$$

This program is ambiguous (wrt multiple inheritance). Intuitively, it is possible that the body of clause 21 is false, in which case there is no reason to inherit clause 19 (which defines method *severity* on types rvf and vcf due to inheritance) in preference to clause 20 (which defines method *severity* on types cf and fsf) or vice-versa. □

Now, to ensure that functional methods really are single-valued, we impose the following range-restriction condition on programs.

Definition A variable z is *restricted* wrt the terms t, $t_1, \ldots, t_n$ in the conjunction C if C contains a method atom $t'[m'@t'_1, \ldots, t'_k \rightarrow z]$ and any variable in t', $t'_1, \ldots, t'_k$ either occurs in t, $t_1, \ldots, t_n$ or is itself restricted wrt t, $t_1, \ldots, t_n$ in the remainder of C.

For example, the variable z is restricted wrt a and x in the conjunction $a[m'@x \rightarrow y] \wedge a[m''@y \rightarrow z]$.

Definition A program P is *well-defined* if the following conditions hold:

- For each clause $\Gamma \vdash t[m@t_1, \ldots, t_n \rightarrow u] \leftarrow C$ in P, u is either an object symbol or is a variable that is restricted wrt t, $t_1, \ldots, t_n$ in C. This is the usual “safety” condition.

- For each functional method m of signature $\tau \times \tau_1 \times \cdots \times \tau_n \rightarrow \tau'$, and for each type $\sigma \leq \tau$, program P does *not* contain two clauses:

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

  such that $\Gamma \vdash t : \sigma$, $\Gamma \vdash t' : \sigma$, and atoms $t[m@t_1, \ldots, t_n \rightarrow x]$ and $t'[m@t'_1, \ldots, t'_n \rightarrow x']$ (where x and x' are new variables) are unifiable. This together with the first condition ensures that functional methods really are single-valued.

- P is unambiguous with respect to multiple inheritance.

Example 2.4 Assume appropriate declarations. Consider the program P

$$\{x : sf\} \vdash x[severity \rightarrow high] \leftarrow x[treatment \rightarrow \text{“don't know”}]$$
$$\{x : sf\} \vdash x[severity \rightarrow low] \leftarrow \neg x[treatment \rightarrow \text{“don't know”}].$$

This program is not well-defined. $\square$

With the above definitions we can now define “simple” programs.

Definition A program P is *simple* if the following two conditions hold:

- P is i-stratified, and
- P is well-defined.

3 Semantics of Gulog

In this section we formally define the semantics of a Gulog program. In particular, we define interpretations, models, and preferred models for a given program.

We define an interpretation of a program P with respect to a set of declarations D.

Definition The *typed Herbrand base* of P is the set of all ground atoms which can be formed from the typed objects, typed predicates, and typed methods in P with respect to D.

A *typed Herbrand interpretation* I for P consists of a set of well-typed ground atoms of P with respect to D.

Definition Let Γ be a set of variable typings. A *variable assignment* $\mathcal{V}$ (wrt Γ) is an assignment to each variable x of an object o such that, if $x : c \in \Gamma$ and $o : c' \in D$, then $c' \leq c$.

Definition Let Γ be a set of variable typings and $\mathcal{V}$ a variable assignment (wrt Γ). Then the *term assignment* $\mathcal{T}$ (based on $\mathcal{V}$ wrt Γ) is the assignment to each term t of an object o as follows:

- If t is a variable x, then t is assigned $\mathcal{V}(x)$.
- If t is an object, then t is assigned itself.

Definition Let I be an interpretation, Γ a set of variable typings, and $\mathcal{T}$ a term assignment (wrt Γ). Then a literal is given a truth value in I (wrt $\mathcal{T}$ and Γ) as follows:

- If $p(t_1, \ldots, t_n)$ is a predicate atom, and $t'_1, \ldots, t'_n$ are the term assignments of $t_1, \ldots, t_n$ (wrt $\mathcal{T}$), then $p(t_1, \ldots, t_n)$ is true in I (wrt $\mathcal{T}$ and Γ) if $p(t'_1, \ldots, t'_n) \in I$.

- If $t[m@t_1,\ldots,t_k \rightarrow t_n]$ is a functional method atom, and $t',t'_1,\ldots,t'_k,t'_n$ are the term assignments of $t,t_1,\ldots,t_k,t_n$ (wrt $\mathcal{T}$), then $t[m@t_1,\ldots,t_k \rightarrow t_n]$ is true in I (wrt $\mathcal{T}$ and Γ) if $t'[m@t'_1,\ldots,t'_k \rightarrow t'_n] \in I$.
- If $t[m@t_1,\ldots,t_k \twoheadrightarrow t_n]$ is a set-valued method atom, and $t',t'_1,\ldots,t'_k,t'_n$ are the term assignments of $t,t_1,\ldots,t_k,t_n$ (wrt $\mathcal{T}$), then $t[m@t_1,\ldots,t_k \twoheadrightarrow t_n]$ is true in I (wrt $\mathcal{T}$ and Γ) if $t'[m@t'_1,\ldots,t'_k \twoheadrightarrow t'_n] \in I$.
- If $\neg p(t_1,\ldots,t_n)$ is a negative literal and $t'_1,\ldots,t'_n$ are the term assignments of $t_1,\ldots,t_n$ (wrt $\mathcal{T}$ and Γ), then $\neg p(t_1,\ldots,t_n)$ is true in I (wrt $\mathcal{T}$ and Γ) if $p(t'_1,\ldots,t'_n) \notin I$.

A ground clause is given a truth value in I in the normal way. Before defining a "model", we define the term "overrides" which is defined in terms of "possibly overrides".

Definition Let P be a program and I an interpretation. A ground clause C *overrides* a ground clause C' with respect to P and I if:

- C possibly overrides C' with respect to P, and
- the bodies of C and C' are both true in I.

Example 3.1 Assume appropriate declarations including declarations 2 to 7, and $f1 : csf$. Consider program P:

$$\{x : sf\} \vdash x[severity \twoheadrightarrow high]$$
$$\{x : csf\} \vdash x[severity \twoheadrightarrow low] \leftarrow x[treatment \rightarrow \text{“tell supervisor”}]$$

and interpretation $I_2 = \{f1[severity \twoheadrightarrow low], f1[treatment \rightarrow \text{“tell supervisor”}]\}$. The ground clause $f1[severity \twoheadrightarrow low] \leftarrow f1[treatment \rightarrow \text{“tell supervisor”}]$ overrides $f1[severity \twoheadrightarrow high]$ with respect to P and I_2. $\square$

Definition Let P be a program and I an interpretation. We say that I is a *model* of P if, for every ground instance C' of a clause in P, either:

- C' is true in I, or
- there exists another ground instance C of a clause in P such that C overrides C' with respect to I.

If I is a model of P, we write $I \models P$.

Example 3.2 Consider the declarations and program P given in Example 3.1. Consider interpretations $I_1 = \{f1[severity \twoheadrightarrow high]\}$ and $I_2 = \{f1[severity \twoheadrightarrow low], f1[treatment \rightarrow \text{“tell supervisor”}]\}$. Because every ground instance of a clause in P is true in I_1, $I_1 \models P$. Because the ground clause $f1[severity \twoheadrightarrow low] \leftarrow f1[treatment \rightarrow \text{“tell supervisor”}]$ overrides $f1[severity \twoheadrightarrow high]$ with respect to P and I_2, and there are no other ground instances of clauses in P, $I_2 \models P$. $\square$

Note that, unlike common usage, not every clause instance need be true in one of our models. Note also that a program may have more than one model. To identify a preferred model, we define a priority relation between ground atoms and then define preferred models using this relation. Our definitions are based on Przymusinski's perfect model semantics for logic programs [16].

Definition Let P be a program. We define a priority relation $<$ based on P between ground atoms. We write $a \leq b$ to denote $a < b$ or $a = b$. For every ground instance $C\theta$ of a clause C in P, we define:

- $A \leq A'$, where A is the head of $C\theta$, and A' is a ground positive literal in the body of $C\theta$,
- $A < A'$, where A is the head of $C\theta$, and $\neg A'$ is a ground negative literal in the body of $C\theta$,
- $A < B'$ for every ground instance $C'\theta'$ of C' in P, which possibly overrides $C\theta$, where A is the head of $C\theta$ and B' is a literal in the body of $C'\theta'$,
- $A \leq A'$ for every ground instance $C'\theta'$ of C' in P, which possibly overrides $C\theta$, where A is the head of $C\theta$ and A' is the head of $C'\theta'$.

Note also that:

- if $A \leq B$ and $B \leq C$, then $A \leq C$,
- if $A \leq B$ and $B < C$ (resp. $D < A$) then $A < C$ (resp. $D < B$).

The goal of the following preference relation is to minimise higher priority atoms as much as possible.

Definition Suppose that M and N are two distinct models of a program P. Then N is *preferable* to M ($N \ll M$) if, for every ground atom A in $N \backslash M$, there is a ground atom B in $M \backslash N$ such that $A < B$. We write $N \leq M$ if $N \ll M$ or $N = M$. We say model N is a *preferred* model of P if there are no models of P preferable to N.

Example 3.3 Consider the declarations and program P given in Example 3.1. The interpretations $I_1 = \{f1[severity \twoheadrightarrow high]\}$ and $I_2 = \{f1[severity \twoheadrightarrow low], f1[treatment \rightarrow$ "tell supervisor"$]\}$ are both (minimal) models of the program P. Because $f1[severity \twoheadrightarrow high] < f1[treatment \rightarrow$ "tell supervisor"$]$ and there is no atom $A \in I_2 \backslash I_1$ such that there exists $B \in I_1 \backslash I_2$, where $A < B$, $I_1 \ll I_2$. □

Our definition of preferred model is different from the definition of perfect model for Datalog programs. The priority relation is based on overriding and negation rather than just negation. For Gulog programs an atom in the head of a possibly overriding clause has higher priority than atoms in the body of the clause it possibly overrides. Like in the case of Datalog programs, negative literals in the body of a clause have higher priority than the atom in the head of the clause.

The definitions of "answer" and "correct answer" are standard except, in Gulog, answers are ground.

Definition Let P be an i-stratified program with preferred model M_P and G a goal. An *answer* for $P \cup \{G\}$ is a ground substitution for variables of G.

Let G be the goal $\Gamma \vdash \leftarrow A_1 \wedge \ldots \wedge A_k$, and θ an answer for $P \cup \{G\}$. Then θ is a *correct answer* for $P \cup \{G\}$ if $(\Gamma \vdash A_1 \wedge \ldots \wedge A_k)\theta$ is true in M_P.

A Gulog program may have more than one preferred model. However, in [11], we proved that, in the absence of sets, negation and multiple inheritance, every simple program has exactly one preferred model. In this case, the preferred model provides a natural semantics for Gulog programs. We also provided both bottom-up and top-down query evaluation procedures for this class of program and proved that these procedures are sound with respect to the declarative semantics for simple programs. Below we show that the preferred model also provides a natural semantics in the extended language, by applying results from the semantics of logic programming.

4 Translation to Datalog

In this section we give a translation from Gulog programs to Datalog programs. This translation extends the translation from C-logic to first-order logic [8], by including the translation of overriding. A similar translation is given by McCabe for *L&O* in [15]. The semantics of inheritance with overriding in *L&O* is different from the semantics of inheritance with overriding in Gulog. The translation we describe is similar to dynamic rewriting described in [4]. We extend dynamic rewriting by distinguishing between functional and set-valued methods, and allowing multiple inheritance.

The object and hierarchy declarations of a Gulog program are included in the translated program. An object declaration $a : \tau$ is translated to an atom $\$\tau(a)$. A type hierarchy declaration $\sigma < \tau$ is translated to a clause $\$\tau(x) \leftarrow \$\sigma(x)$. A variable typing $\{x_1 : \sigma_1, \cdots, x_n : \sigma_n\}$ is translated to a conjunction of atoms $\$\sigma_1(x_1) \wedge \ldots \wedge \$\sigma_n(x_n)$. A predicate atom $p(t_1, \ldots, t_n)$ is left unchanged in the translation. A functional method atom with arity n $t[m@t_1, \ldots, t_n \rightarrow t']$ is translated to an $(n+2) - ary$ predicate $\$m(t, t_1, \ldots, t_n, t')$. A set-valued method atom with arity n, $t[m@t_1, \ldots, t_n \twoheadrightarrow t']$ is translated to $\$\$m(t, t_1, \ldots, t_n, t')$. Negative literals retain their negation in the translation, for example $\neg A$ is translated to $\neg A'$ where A' is the translation of A.

Let $\Gamma \vdash A$ be a predicate atom, and B the body of a clause, the clause $\Gamma \vdash A \leftarrow B$, is translated to the clause $A' \leftarrow \Gamma' \wedge B'$, where A' is the translation of A, Γ' is the translation of Γ, B' is the translation of B.

If a clause defines a method on a type, and another defines the same method on a subtype, these clauses must be considered together. Let $\Gamma \vdash A(t, t_1, \ldots, t_k, t')$ be a method atom for method m of arity k on instances t of type τ, with arguments $t_1, \ldots, t_k$, and value t'. The clause defining this method is $C = \Gamma \vdash A(t, t_1, \ldots, t_k, t') \leftarrow B$. Let $\sigma_1, \ldots, \sigma_n$ be all the (direct or indirect) subtypes of τ that have method m of arity k defined on their instances t_i of type σ_i by clauses: $C_i = \Gamma_i \vdash A(t_i, t_{i1}, \ldots, t_{ik}, t_i') \leftarrow B_i$, for $1 \leq i \leq n$. The clause defining method m of arity k on instances of type τ is translated to the Datalog clauses:

$$app_\tau(t, t_1, \ldots, t_k) \leftarrow \Gamma' \wedge B'.$$
$$A'(t, t_1, \ldots, t_k, t') \leftarrow \Gamma' \wedge B' \wedge \neg app_{\sigma_1}(t, t_1, \ldots, t_k) \wedge \cdots \wedge \neg app_{\sigma_n}(t, t_1, \ldots, t_k).$$

where $A'(t, t_1, \ldots, t_k, t')$ is the translation of $A(t, t_1, \ldots, t_k, t')$, Γ' is the translation of Γ, and B' is the translation of B.

Inheritance with overriding in a Gulog program can thus easily be translated to inference with negation in a Datalog program.

Example 4.1 In this example we record the failure identifier and the date of the failure, unless a malfunction has been detected. We assume a function which generates todays date, and an equivalence predicate exist. Assume appropriate declarations including:

$$\begin{array}{ll} r1 : reactor & temp\_sensor : part \\ f1 : tf & csf < sf \\ cf < csf & fsf < csf. \end{array}$$

Consider the Gulog program P based on these declarations:

$$\{x : reactor, d : date, y : sf, z : string, w : part\} \vdash x[failure@d \twoheadrightarrow y] \leftarrow$$
$$\quad y[problem \rightarrow z] \wedge d = today \wedge \neg y[malfunction \rightarrow w]$$
$$\{x : sf\} \vdash x[problem \rightarrow \text{“system failure”}]$$
$$\{x : csf\} \vdash x[problem \rightarrow \text{“cooling system failure”}] \leftarrow x[temp \rightarrow high]$$
$$\{x : fsf\} \vdash x[problem \rightarrow \text{“cooling system flow high”}] \leftarrow x[flow \rightarrow high].$$

This is translated to the Datalog program P':

$$
\begin{array}{ll}
\$reactor(r1). & \$part(temp\_sensor). \\
\$tf(f1). & \$sf(x) \leftarrow \$csf(x). \\
\$csf(x) \leftarrow \$cf(x). & \$csf(x) \leftarrow \$fsf(x).
\end{array}
$$

$$
\begin{array}{l}
\$\$failure(x,d,y) \leftarrow \$reactor(x) \wedge \$date(d) \wedge \$sf(y) \wedge \$string(z) \wedge \$part(w) \wedge \\
\quad \$problem(y,z) \wedge d = today \wedge \neg\$malfunction(y,w). \\
\$problem(x, \text{“system failure”}) \leftarrow \$sf(x) \wedge \neg app_{csf}(x) \wedge \neg app_{fsf}(x). \\
\$problem(x, \text{“cooling system failure”}) \leftarrow \$csf(x) \wedge \$temp(x, high) \wedge \neg app_{fsf}(x). \\
\$problem(x, \text{“cooling system flow high”}) \leftarrow \$fsf(x) \wedge \$flow(x, high). \\
app_{reactor}(x,d) \leftarrow \$reactor(x) \wedge \$date(d) \wedge \$sf(y) \wedge \$string(z) \wedge \$part(w) \wedge \\
\quad \$problem(y,z) \wedge d = today \wedge \neg\$malfunction(y,w). \\
app_{sf}(x) \leftarrow \$sf(x). \\
app_{csf}(x) \leftarrow \$csf(x) \wedge \$temp(x, high). \\
app_{fsf}(x) \leftarrow \$fsf(x) \wedge \$flow(x, high).
\end{array}
$$

□

We now describe the translation of an interpretation of a Gulog program to an interpretation of a Datalog program. Given an interpretation I of program P (wrt a set of declarations D) the translation I' of I (wrt P) is constructed as follows. Each predicate or method atom in I is translated to a predicate atom in I' as given above. Every object in D is assigned atoms in I' which are consistent with the object and type hierarchy declarations in D. We call these atoms *type atoms* and the related predicates *type predicates*. Let $C = \Gamma \vdash A(t, t_1, \ldots, t_n, t') \leftarrow B$ be a clause in P defining the method m of arity n on instances t of type τ. For every instance $C\theta$ of C consistent with the typing Γ, $app_\tau(t, t_1, \ldots, t_n)\theta \in I'$ if for every literal B_i in B, B_i is a positive literal and $B_i\theta \in I$ or B_i is a negative literal $\neg B_{in}$ and $B_{in}\theta \notin I$. We call these atoms *app-atoms* and the related predicates *app-predicates*.

The translation from Gulog to Datalog programs described in this section is like the dynamic rewriting from $datalog^{meth}$ to Datalog described by Abiteboul et al. in [4]. In their rewriting, redundant subgoals are generated in the clauses which define the applicable predicates (*app*-predicates). They claim that their rewriting is well-founded for positive $datalog^{meth}$ programs. They do not have a similar result for $datalog^{meth}$ programs which contain negative literals. We show below that any i-stratified Gulog program is translated to a locally stratified Datalog program.

5 Properties of the translation

In this section we study the relationships between the original Gulog program P and its Datalog translation P', between the preferred model of P and the perfect model of P', and between computed answers with respect to P' and correct answers with respect to P.

Lemma 5.1 *Let P be an i-stratified program and P' be the Datalog translation of P. Then P' is locally stratified.*

Proof As P is i-stratified, a mapping μ consistent with the definition of i-stratification exists on P. We now demonstrate that a mapping μ' consistent with the definition of local stratification exists on P'. We start by defining the construction of the mapping μ' based on the mapping μ, the program P, and its translation P'.

- In μ' each ground type atom of P' is assigned to the lowest level.

- For each ground atom $(\Gamma \vdash B)\theta$, which is in the body of a clause $C\theta$, and not in the head of a clause $C'\theta'$ in P, and is assigned level i in μ, the translation of $(\Gamma \vdash B)\theta$ is assigned level i in μ'.

- For each ground atom $(\Gamma \vdash A)\theta$, which is the head of a clause $C\theta$ in P, and is assigned level i in μ, the translation of $(\Gamma \vdash A)\theta$ is assigned level $i+1$ in μ'.

- For each ground instance of a clause $(app_\tau(t, t_1, \ldots, t_k) \leftarrow B)\theta$ in P', let $B_i\theta$ (resp. $\neg B_i\theta$) be a literal in $B\theta$ with the greatest level j. If $B_i\theta$ is a positive literal then $app_\tau(t, t_1, \ldots, t_k)\theta$ is assigned level j in μ', else if $\neg B_i\theta$ is a negative literal then $app_\tau(t, t_1, \ldots, t_k)\theta$ is assigned level $j+1$ in μ'.

We now show that if there is a mapping μ on P consistent with the definition of i-stratification then μ', a mapping constructed using the procedure above, is a mapping on P' consistent with the definition of local stratification. Suppose $C = \Gamma \vdash A \leftarrow B$ is a clause in P, where $\Gamma \vdash A$ is a typed predicate atom. This clause is translated to the clause $C' = A' \leftarrow \Gamma' \wedge B'$ in P', where A' is the translation of A, Γ' is the translation of Γ, and B' is the translation of B. Based on the above construction, for every ground instance $C'\theta$ of C', for each positive literal B'_i in B', $\mu'(B'_i\theta) \leq \mu'(A'\theta)$, for each negative literal $\neg B'_i$ in B', $\mu'(B'_i\theta) < \mu'(A'\theta)$, and for each atom Γ'_i in Γ', $\mu'(\Gamma'_i\theta) < \mu'(A'\theta)$.

Suppose $C = \Gamma \vdash A(t, t_1, \ldots, t_k, t') \leftarrow B$ is a clause in P, where $\Gamma \vdash A(t, t_1, \ldots, t_k, t')$ is a typed method atom for method m of arity k on instances t of type τ with arguments $t_1, \ldots, t_k$, and value t'. Suppose also that $C_i = \Gamma_i \vdash A(t_i, t_{i1}, \ldots, t_{ik}, t'_i) \leftarrow B_i$, for $1 \leq i \leq n$ are clauses in P for method m of arity k defined on instances t_i of types $\sigma_1, \ldots, \sigma_n$ which are all (direct or indirect) subtypes of τ. Clause C is translated to the following clauses in P':

$$app_\tau(t, t_1, \ldots, t_k) \leftarrow \Gamma' \wedge B'.$$
$$A'(t, t_1, \ldots, t_k, t') \leftarrow \Gamma' \wedge B' \wedge \neg app_{\sigma_1}(t, t_1, \ldots, t_k) \wedge \cdots \wedge \neg app_{\sigma_n}(t, t_1, \ldots, t_k).$$

where $A'(t, t_1, \ldots, t_k, t')$ is the translation of $A(t, t_1, \ldots, t_k, t')$, Γ' is the translation of Γ and B' is the translation of B. In the level mapping μ, $\mu(A(t_i, t_{i1}, \ldots, t_{ik}, t'_i)\theta_i) \leq \mu(A(t, t_1, \ldots, t_k, t')\theta)$, where $\{t_i/a, t_{ij}/a_j, t'_i/a'_i\} \in \theta_i$ and $\{t/a, t_j/a_j, t'/a'\} \in \theta$ and $a'_i \neq a'$, for $1 \leq j \leq k$. Based on the above construction of μ', $\mu'(app_{\sigma_i}(t, t_1, \ldots, t_k)\theta) < \mu'(A'(t, t_1, \ldots, t_k, t')\theta)$, $\mu'(app_\tau(t, t_1, \ldots, t_k)\theta) \leq \mu'(A'(t, t_1, \ldots, t_k, t')\theta)$, $\mu'(\Gamma'\theta) \leq \mu'(app_\tau(t,$
$t_1, \ldots, t_k)\theta)$, for each positive literal B'_i in B', $\mu'(B'_i\theta) \leq \mu'(app_\tau(t, t_1, \ldots, t_k)\theta)$, and for each negative literal B'_i in B', $\mu'(B'_i\theta) < \mu'(app_\tau(t, t_1, \ldots, t_k)\theta)$.

We have demonstrated that if there is a mapping on P consistent with the definition of i-stratification then there is a mapping on P' consistent with the definition of local stratification. $\square$

In the above construction of μ', a level mapping on P', it is possible to assign all the type atoms to the lowest level because they depend only on other type atoms.

The following example illustrates the construction of a level mapping for the translation of a program based on the level mapping of the original program as described in the proof above.

Example 5.1 Consider program P in Example 4.1. A level mapping μ on P assigns atoms $f1[flow \rightarrow high], f1[temp \rightarrow high], f1[problem \rightarrow$ "cooling system flow high"], $f1[malfunction \rightarrow temp\_sensor]$ to 1; and $f1[problem \rightarrow$ "cooling system failure"], $f1[problem \rightarrow$ "system failure"], $r1[failure@23Sept \rightarrow f1]$ to 2.

Consider the program P' in Example 4.1. A level mapping μ' on P' can be constructed as given in the proof of Lemma 5.1. The atoms $\$fsf(f1), \$cf(f1), \$csf(f1), \$sf(f1), \$flow(f1,$

$high), \$temp(f1, high), \$part(temp\_sensor), \$reactor(r1), \$string($"system failure"$), \$string$ ("cooling system flow high"), $\$string($"cooling system failure"$), app_{sf}(f1), app_{fsf}(f1), app_{csf}$ $(f1), \$malfunction(f1, temp\_sensor)$ are assigned 1; $\$problem(f1,$ "cooling system flow high") is assigned 2; and $\$problem(f1,$ "cooling system failure"), $\$problem(f1,$ "system failure"), $app_{reactor}(r1, 23Sept), \$\$failure(r1, 23Sept, f1)$ are assigned 3. □

Lemma 5.2 *Let P be a program and M a model of P. Let P' be the translation of P and M' the translation of M (wrt P). Then M' is a model of P'.*

Proof We need to show that every ground instance of a clause C' in P' is true in M'.

Suppose $C = \Gamma \vdash A \leftarrow B$ is a clause in P, where $\Gamma \vdash A$ is a typed predicate atom. Then there is a clause $C' = A' \leftarrow \Gamma' \wedge B'$ in P', where A' is the translation of A, Γ' is the translation of Γ, and B' is the translation of B. Because M is a model of P, $C\theta$ is true in M for all ground instances $C\theta$ of C. As every atom in M is mapped to an atom in M', $M' \models C'\theta$ for every θ for which $C\theta$ is true in M. For every other ground instance of C', $C'\theta'$, $M' \models C'\theta'$ because $\Gamma'\theta'$ is false in M'.

Now suppose $C = \Gamma \vdash A(t, t_1, \ldots, t_p, t') \leftarrow B$ is a clause in P, where $\Gamma \vdash A(t, t_1, \ldots, t_p, t')$ is a typed method atom for method m of arity p on instances t of type τ with arguments $t_1, \ldots, t_p$, and value t'. Suppose also that $C_i = \Gamma_i \vdash A(t_i, t_{i1}, \ldots, t_{ip}, t'_i) \leftarrow B_i$, for $1 \leq i \leq n$, are clauses in P for method m of arity p defined on instances t_i of types $\sigma_1, \ldots, \sigma_n$ which are all (direct or indirect) subtypes of τ. Clause C is translated to the following clauses in P':

$$C'_\tau = app_\tau(t, t_1, \ldots, t_p) \leftarrow \Gamma' \wedge B'.$$
$$C' = A'(t, t_1, \ldots, t_p, t') \leftarrow \Gamma' \wedge B' \wedge \neg app_{\sigma_1}(t, t_1, \ldots, t_p) \wedge \cdots \wedge \neg app_{\sigma_n}(t, t_1, \ldots, t_p).$$

where $A'(t, t_1, \ldots, t_p, t')$ is the translation of $A(t, t_1, \ldots, t_p, t')$, Γ' is the translation of Γ and B' is the translation of B.

We first consider substitutions which do not satisfy Γ. As every atom in M is mapped to an atom in M', $M' \models C'_\tau\theta$ and $M' \models C'\theta$, for all instances of $C'_\tau\theta$ and $C'\theta$, where θ does not satisfy Γ because $\Gamma'\theta$ is false in M'.

We next consider substitutions which do satisfy Γ. For all ground instances $C\theta$ of C, there are two cases we must consider. Either $C\theta$ is true in M or there is an instance $C_k\theta_k$ of a clause C_k which overrides $C\theta$.

In the first case $C\theta$ is true in M and there is no θ_i such that $(\Gamma_i \vdash B_i)(\theta_i)$ is true in M, where $\{t_i/a, t_{ij}/a_j, t'_i/a'_i\} \in \theta_i$, $\{t/a, t_j/a_j, t'/a'\} \in \theta$ and $a'_i \neq a'$, for $1 \leq j \leq p$. Then $M' \models C'_\tau\theta$ and $M' \not\models app_{\sigma_i}(t, t_1, \ldots, t_p)\theta$, for $1 \leq i \leq n$, so $M' \models C'\theta$.

In the second case $C_k\theta_k$ is true in M, and $A(t, t_1, \ldots, t_p, t')\theta$ is not true in M, where $\{t_k/a, t_{kj}/a_j, t'_k/a'_k\} \in \theta_k$, $\{t/a, t_j/a_j, t'/a'\} \in \theta$ and $a'_k \neq a'$, for $1 \leq j \leq p$. Then $M' \models app_{\sigma_k}(t, t_1, \ldots, t_p)\theta$, and $A'(t, t_1, \ldots, t_p, t')\theta$ is not true in M', so $M' \models C'\theta$, and because of the way M' is constructed, $M' \models C'_\tau\theta$.

We have shown that M' is a model of P'. □

The next lemma, which demonstrates a relationship between a preferred model of a program and the perfect model of its translation, uses the concept of "negative dependency". The literals in the body of a ground clause C, which is possibly overridden by a ground clause C', depend negatively on the head of the clause C'.

Definition We define a *dependency graph* for a program P with respect to a set of declarations D as follows. Let $[P]$ denote the set of all ground instances of clauses in P. The nodes in the graph correspond to the ground atoms in the Herbrand base of P wrt D. A positive arc $\phi \stackrel{+}{\leftarrow} \psi$ connects a pair of nodes ϕ, ψ if there is a clause $\phi \leftarrow \ldots \wedge \psi \wedge \ldots$ in $[P]$. A negative arc

$\phi \overset{-}{\leftarrow} \psi$ connects a pair of nodes ϕ, ψ if there is a clause $\phi \leftarrow \ldots \wedge \neg\psi \wedge \ldots$ in $[P]$. A positive arc $\phi \overset{+}{\leftarrow} \psi$ connects a pair of nodes ϕ, ψ if there are clauses $C_2 = \phi \leftarrow B_2$ and $C_1 = \psi \leftarrow B_1$ in $[P]$, where C_1 possibly overrides C_2. A negative arc $\phi \overset{-}{\leftarrow} \psi$ connects a pair of nodes ϕ, ψ if there are clauses $C_2 = \phi \leftarrow B$ and $C_1 = A \leftarrow \ldots \wedge \psi \wedge \ldots$ or $C_1 = A \leftarrow \ldots \wedge \neg\psi \wedge \ldots$ in $[P]$, where C_1 possibly overrides C_2. We say a ground atom A_1 *depends on* a ground atom A_2 if there is a path from A_2 to A_1 in the dependency graph. We say a ground atom A_1 *depends negatively on* a ground atom A_2 if there is a path from A_2 to A_1 and at least one of the arcs in the path is a negative arc.

Lemma 5.3 *Let P be an i-stratified program (wrt a set of declarations D) and P' the Datalog translation of P. Let M be a preferred model of P and M' the translation of M. Then M' is the perfect model of P'.*

Proof Suppose M' is not the perfect model of P'. Then there is a model $N' \ll M'$ such that $N' \models P'$.

Consider the translation from N' to N defined as follows.

- Remove all type atoms and *app*-atoms from N', and
- translate the remaining atoms in N' to predicate atoms, functional and set-valued method atoms in N.

The resulting interpretation N is clearly a model of P. We now show $N' \ll M'$ implies that $N \ll M$, which contradicts the assumption that M' is not perfect. Assume there is an $A \in N \backslash M$ then there is an $A' \in N' \backslash M'$ (where A' is the translation of A) and there is a $B' \in M' \backslash N'$ such that $A' < B'$.

We need to show that if $B' \in M' \backslash N'$ then there is a $B \in M \backslash N$. That is, B' is not a type atom and if B' is a *app*-atom then there is another atom B^* in $M' \backslash N'$ which is not a type or *app*-atom. Due to the construction of M', M' contains the minimal model of the type atoms. This part of the interpretation is based on the declarations D so any type atom in M' is also in N'. If B' is a *app*-atom then either there is a type atom B^* in $M' \backslash N'$ or there is a predicate or method atom B^* in $M' \backslash N'$ on which B' depends in P'. We proved above that there are no type atoms in $M' \backslash N'$. Thus B^* is a predicate or method atom so we can assign B^* to B'. As B' is the image of a predicate or method atom B then $B \in M \backslash N$. So there is a $B \in M \backslash N$.

We now need to show that if $A' < B'$ then $A < B$. $A < B$ if A is negatively dependent on B in P. Due to the way P' is constructed from P, if $A' < B'$ in P' then $A < B$.

With these results, if $N' \ll M'$ then $N \ll M$. As M is a preferred model of P, this is a contradiction so M' is the perfect model of P'. $\Box$

Proposition 5.1 *Let P be an i-stratified program. Then P has a unique preferred model.*

Proof Suppose M and N are preferred models of P. Let M', N' and P' be translations of M, N and P respectively. Then the translations M' and N' are perfect models of P' (by lemma 5.3). Because P' is locally stratified (lemma 5.1), P' has a unique perfect model so P must have a unique preferred model. $\Box$

In [11] we showed that the unique preferred model provided a natural semantics for the restricted language defined there. We have shown above that the unique preferred model also provides a natural semantics for the extended language defined here.

Proposition 5.2 *Let P be a i-stratified program and Q an atomic query $\Gamma \vdash\ \leftarrow A$. Let P' be the translation of P and Q' the translation of Q. If θ is a computed answer for $P' \cup \{Q'\}$, then θ is a correct answer for $P \cup \{Q\}$.*

We postpone the proof of this proposition to Section 6 where we prove a more general result.

In [10], we described bottom-up and top-down query evaluation procedures for a subset of this language. Proposition 5.2 provides an alternative procedure for evaluating more general Gulog queries.

6 Generalizations

Some programs have intuitively well-defined meanings, but no or several preferred models. In this section we identify a class of programs that includes the simple programs, and have an intuitive and well-defined semantics, which we present. This class of programs takes computation into account. It bears the same relationship to simple programs as modularly stratified Datalog programs bears to locally stratified Datalog programs.

In the following example we present a program which is not simple and describe the intuitive meaning of this program.

Example 6.1 Assume appropriate declarations D including $f1 : rvf$, $rvf < psf$ and $psf < sf$. Let P (wrt D) be:

$\{x : sf\} \vdash x[treatment \twoheadrightarrow \text{“call supervisor”}]$
$\{x : psf\} \vdash x[treatment \twoheadrightarrow \text{“lower pressure”}] \leftarrow x[pressure \rightarrow high]$
$\{x : rvf\} \vdash x[treatment \twoheadrightarrow \text{“open relief valve”}] \leftarrow x[treatment \twoheadrightarrow \text{“lower pressure”}]$.

Program P is not i-stratified. However we believe it has an intuitive meaning. There is no reason to believe that $f1[pressure \rightarrow high]$ is true. Thus there is no reason to believe that $f1[treatment \twoheadrightarrow \text{“lower pressure”}]$ or $f1[treatment \twoheadrightarrow \text{“open relief valve”}]$ is true. The natural meaning of the program is $f1[treatment \twoheadrightarrow \text{“call supervisor”}]$. $\square$

To introduce this class of programs we first need some preliminary definitions. Our definition of m-stratified is based on the definition for modular stratification [17]. We give the relevant revised definitions here to show how they are altered to deal with inheritance with overriding in Gulog rather than negation in Datalog.

Definition A predicate or method p *depends on* a predicate or method q if there is a sequence of clauses $r_0, \ldots, r_{n-1}$ with methods or predicates $p_0, \ldots, p_{n-1}$ in the head respectively such that:

- $p = p_0$ and $q = p_n$, and
- for $i = 1, \ldots, n$, p_i appears in the body of r_{i-1}.

A predicate or method p is *recursive* in a predicate or method q if p depends on q and q depends on p.

Definition Let F be a component (i.e., a subset of clauses) of a program P. F is a *complete component* if for every predicate or method p appearing in the head of clause F,

- all clauses in P with head p are in F, and
- if p is recursive in a predicate or method q, then all clauses in P with head q are in F.

If the predicate or method p appears in the head of a clause in F then we say p *belongs to* F. If the predicate q appears in the body of a clause in F, but does not belong to F, then we say q is *used by* F.

Recursiveness is an equivalence relation between predicates and methods. Every predicate and method has a unique minimal complete component to which it belongs. The minimum complete components have a natural relation associated with them: $F_1 \sqsubset F_2$ if some predicate belonging to F_1 is used by F_2. $\prec$ is the transitive closure of $\sqsubset$ as the *dependency* relation between components. From now on when we refer to a component of a program, we mean a minimal complete component.

We say a component F_k has *level* 1 if there is no component F such that $F \prec F_k$. A component F_k has *level* $n+1$ if there is a component F with level n such that $F \sqsubset F_k$.

Definition Let F be a program component, and S be the set of predicates and methods used by F. Suppose S is fully defined by an (typed Herbrand) interpretation M, i.e., every typed ground instance of a predicate or method in S is either true or false in M. Form $I(F)$, the typed instantiation of F, by substituting typed objects for all typed variables in the clauses of F. Delete from $I(F)$ all clauses having a subgoal whose predicate or method is in S but which is false in M. From the remaining clauses, delete all subgoals having predicates or methods in S (which must be true) to leave a set of instantiated clauses $R_M(F)$. $R_M(F)$ is the *reduction of F modulo M*.

We give a constructive definition of m-stratified. It is based on the construction of the well-founded model of a program which is briefly described in [17].

Definition Let $\prec$ be the dependency relation between components. We say that P is *m-stratified* with associated model M_P if,

- for every component F_i of P,
  - $F_i' = \bigcup_{F \prec F_i} F$ is m-stratified with associated model M_i', and
  - $R_{M_i'}(F_i)$ is i-stratified with (unique) preferred model M_i, and
- $M_P = \bigcup\{M_i \mid F_i$ is a component of P with associated model $M_i\}$.

The next example demonstrates the construction of the associated model of a program.

Example 6.2 Consider program P of Example 6.1. Component $F_1 = \emptyset$ defines the method *pressure*. Component F_2 defines the method *treatment*:

$\{x : sf\} \vdash x[treatment \twoheadrightarrow$ "call supervisor"$]$
$\{x : psf\} \vdash x[treatment \twoheadrightarrow$ "lower pressure"$] \leftarrow x[pressure \rightarrow high]$
$\{x : rvf\} \vdash x[treatment \twoheadrightarrow$ "open relief valve"$] \leftarrow x[treatment \twoheadrightarrow$ "lower pressure"$]$.

The methods used by F_2 are $S = \{pressure\}$. S is fully defined by the Herbrand interpretation $M_1 = \emptyset$. The typed instantiation of F_2 is:

$\{f1 : sf\} \vdash f1[treatment \twoheadrightarrow$ "call supervisor"$]$
$\{f1 : psf\} \vdash f1[treatment \twoheadrightarrow$ "lower pressure"$] \leftarrow f1[pressure \rightarrow high]$
$\{f1 : rvf\} \vdash f1[treatment \twoheadrightarrow$ "open relief valve"$] \leftarrow f1[treatment \twoheadrightarrow$ "lower pressure"$]$.

$R_{M_2'}(F_2)$ is:

$\{f1 : sf\} \vdash f1[treatment \twoheadrightarrow$ "call supervisor"$]$
$\{f1 : rvf\} \vdash f1[treatment \twoheadrightarrow$ "open relief valve"$] \leftarrow f1[treatment \twoheadrightarrow$ "lower pressure"$]$.

$M_2 = \{f1[treatment \twoheadrightarrow$ "call supervisor"$]\}$. The reductions of each of the components of program P is i-stratified so P is m-stratified. The associated model of P is
$\{f1[treatment \twoheadrightarrow$ "call supervisor"$]\}$. $\square$

We now define the class of m-simple programs which generalize simple programs in the same way that m-stratification generalizes i-stratification.

Definition A program P is *m-simple* if the following conditions hold:

- P is m-stratified, and
- P is well-defined.

Lemma 6.1 *Let P be an i-stratified program. Then P is m-stratified.*

Proof As program P is i-stratified, each component F of program P is i-stratified. The reduction of each component is also i-stratified so program P is m-stratified. $\Box$

Lemma 6.2 *Let P be an m-stratified program. Then P has a unique preferred model which equals the associated model of P.*

Proof We prove this by induction on the maximum level of components in P. Assume the maximum level of components in P is 1 and F_1 is a component at this level. As $M_1' = \emptyset$, $R_{M_1'}(F_1) = F_1$, and F_1 is i-stratified, then F_1 has a unique preferred model which equals the associated model of F_1.

Assume the result holds where the maximum level of components of P is $n-1$. Then $\bigcup_{F \preceq F_{n-1}}(F)$ has a unique preferred model which equals the associated model $M_{F_{n-1}} = \bigcup\{M_i \mid F_i \preceq F_{n-1}, F_i \text{ has associated model } M_i\}$.

Assume the maximum level of components in P is n and F_n is a component at this level. By the induction hypothesis F_n' has preferred model M_n'. Let M_n be the preferred model of $R_{M_n'}(F_n)$. Suppose $M_n \cup M_n'$ is not the preferred model of $\bigcup_{F \preceq F_n}(F)$. Then there exists a model M' preferable to $M_n \cup M_n'$. That is, for all $A \in M' \backslash M_n \cup M_n'$, there exists $B \in M_n \cup M_n' \backslash M'$ such that $A < B$. Due to the way $R_{M'i_n}(F_n)$ is constructed, this is a contradiction. We have shown $M_n \cup M_n'$ is a preferred model of F_n. We now want to show that it is the only preferred model of F_n, that is for every other model M', $M_n \cup M_n' \ll M'$. Due to the construction of $M_n \cup M_n'$, there are two cases where there is an $A \in M_n \cup M_n' \backslash M'$:

- There is a clause in $[P]$, $A \leftarrow \ldots \wedge \neg B \wedge \ldots$. As $A \in M_n \cup M_n' \backslash M'$, $B \in M' \backslash M_n \cup M_n'$, where $A < B$.
- There are two clauses in $[P]$, $C = A \leftarrow \ldots \wedge L \wedge \ldots$ and $C' = A' \leftarrow \ldots \wedge B \wedge \ldots$, where C' possibly overrides C. As $A \in M_n \cup M_n' \backslash M'$, there is a $B \in M' \backslash M_n \cup M_n'$, where $A < B$.

So for every $A \in M_n \cup M_n' \backslash M'$, there exists $B \in M' \backslash M_n \cup M_n'$, where $A < B$. Thus $M_n \cup M_n' \ll M_n'$ and $M_n \cup M_n'$ is the unique preferred model of F_n.

It follows that the preferred model of an m-stratified program P equals the associated model of P. $\Box$

It follows that m-stratified programs strictly generalise i-stratified programs and that the associated model of an i-stratified program P is the unique preferred model of P.

The class of m-simple programs extends the class of simple programs. It is an interesting class of programs because each m-simple program has a unique associated model, and query evaluation procedures can be defined for this class of programs.

The remaining results describe the relationship between m-simple programs and their translations. The first lemma which draws a parallel between the associated model of an m-stratified program and the well-founded model of its translation uses the concept of "equivalence" of models.

Definition Let P be a Gulog program and P' the Datalog translation of P. We say M_P is *equivalent* to $M_{P'}$ if the translation of every atom in M_P is in $M_{P'}$ and, for every atom A' in $M_{P'}$ that is the translation of a Gulog method or predicate atom A, A is in M_P.

Lemma 6.3 *Let P be an m-stratified program with associated model M_P and P' the Datalog translation of P. Then P' is modularly stratified with well-founded model $N_{P'}$ which is equivalent to M_P.*

Proof We need to show that P' is modularly stratified. To do this we must show for every component G_j of P':

- $G'_j = \bigcup_{G \prec G_j} G$ is modularly stratified with well-founded model N'_j, and
- $R_{N'_j}(G_j)$ is locally stratified with (unique) perfect model N_j.

We also need to show that M_P is equivalent to $N_{P'}$, where $N_{P'} = \bigcup\{N_j \mid G_j$ is a component of P' with well-founded model $N_j\}$.

A component F in P is translated to one or more components G_i, $i \geq 1$ in P'. One component G_k of G_i will define predicates $m_1, \ldots, m_n$ which are translations of the predicates and methods defined in F. We say that G_k *corresponds to* F. The other components G_i, $i \neq k$, will define type and *app*-predicates and $G_i \prec G_k$.

We now prove by induction on the maximum level of components in P that P' is modularly stratified with well-founded model $N_{P'}$ which is equivalent to M_P.

Assume the maximum level of components in P is 1 and F_1 is a component at this level then $M'_1 = \emptyset$, $R_{M'_1}(F_1) = F_1$, and F_1 is i-stratified with preferred model M_1. There is a corresponding component G_{k_1} of P'. Obviously $G'_{k_1} = \bigcup_{G \prec G_{k_1}} G$ is locally stratified (by lemma 5.1), hence modularly stratified and has a well-founded model $N'_{k_1} = \bigcup_{G_i \prec G_{k_1}} N_i$. Also because F_1 is i-stratified, $R_{N'_i}(G_i)$, for $i \leq k_1$, is locally stratified with perfect model N_i. The preferred model of F_1 is equivalent to the perfect model of $\bigcup_{G \preceq G_{k_1}} G$ (by lemma 5.3). Hence the associated model of F_1 is equivalent to the well-founded model of $\bigcup_{G \preceq G_{k_1}} G$.

Assume the above results hold where the maximum level of components of P is $n-1$. Assume F_{n-1} is a component at this level of P, where F_{n-1} corresponds to a component G_{k_2} of P'. The program $\bigcup_{G \preceq G_{k_2}} G$ is modularly stratified and its well-founded model $\bigcup\{N_j \mid G_j \preceq G_{k_2}, G_j$ has well-founded model $N_j\}$ is equivalent to $\bigcup\{M_i \mid F_i \preceq F_{n-1}, F_i$ has associated model $M_i\}$.

Assume the maximum level of components in P is n and F_n is a component at this level. There is a component G_{k_3} of P' which corresponds to F_n. The components G_i, for $k_2 < i < k_3$, define type and *app*-predicates. These components are locally stratified with perfect models N_i. By the induction hypothesis and because the components G_i, for $k_2 < i < k_3$, are locally stratified with perfect models N_i, $G'_{k_3} = \bigcup_{G \prec G_{k_3}} G$ is modularly stratified with well-founded model $N'_{k_3} = \bigcup_{G_i \prec G_{k_3}} N_i$. Model N'_{k_3} is equivalent to M'_n so $R_{N'_{k_3}}(G_{k_3})$ is a translation of $R_{M'_n}(F_n)$ and N_{k_3} is equivalent to M_n.

Because the associated model of a component of a Gulog program is equivalent to the well-founded model of the corresponding Datalog component the associated model M_P is equivalent to $N_{P'}$. □

The next example shows that the associated model of program P in Example 6.1 is equivalent to the well-founded model of the Datalog translation of P.

Example 6.3 Consider program P, and its associated model in Examples 6.1 and 6.2. Consider P', the translation of P:

$\$rvf(f1).$
$\$psf(x) \leftarrow \$rvf(x).$
$\$sf(x) \leftarrow \$psf(x).$
$\$\$treatment(x, \text{“call supervisor”}) \leftarrow \$sf(x) \wedge \neg app_{psf}(x) \wedge \neg app_{rvf}(x).$
$\$\$treatment(x, \text{“lower pressure”}) \leftarrow \$psf(x) \wedge \$pressure(x, high) \wedge \neg app_{rvf}(x).$
$\$\$treatment(x, \text{“open relief valve”}) \leftarrow \$rvf(x) \wedge \$\$treatment(x, \text{“lower pressure”}).$
$app_{sf}(x) \leftarrow \$sf(x).$
$app_{psf}(x) \leftarrow \$psf(x) \wedge \$pressure(x, high).$
$app_{rvf}(x) \leftarrow \$rvf(x) \wedge \$\$treatment(x, \text{“lower pressure”}).$

Component G_1 defines predicate $\$rvf$. The model of $R_{\emptyset}(G_1)$ is $N_1 = \{\$rvf(f1)\}$. Component G_2 defines predicate $\$psf$. The model of $R_{N_2'}(G_2)$ is $N_2 = \{\$psf(f1)\}$. Component G_3 defines the predicate $\$sf$. The model of $R_{N_3'}(G_3)$ is $N_3 = \{\$sf(f1)\}$. Component G_4 defines predicate $\$pressure$. The model of $R_{\emptyset}(G_4)$ is $N_4 = \emptyset$. Component G_5 defines predicate app_{sf}. The model of $R_{N_5'}(G_5)$ is $N_5 = \{app_{sf}(f1)\}$. Component G_6 defines predicate app_{psf}. The model of $R_{N_6'}(G_6)$ is $N_6 = \emptyset$. Component G_7 defines predicates $\$\$treatment$ and app_{rvf}:

$\$\$treatment(x, \text{“call supervisor”}) \leftarrow \$sf(x) \wedge \neg app_{psf}(x) \wedge \neg app_{rvf}(x).$
$\$\$treatment(x, \text{“lower pressure”}) \leftarrow \$psf(x) \wedge \$pressure(x, high) \wedge \neg app_{rvf}(x).$
$\$\$treatment(x, \text{“open relief valve”}) \leftarrow \$rvf(x) \wedge \$\$treatment(x, \text{“lower pressure”}).$
$app_{rvf}(x) \leftarrow \$rvf(x) \wedge \$\$treatment(x, \text{“lower pressure”}).$

Component G_7 corresponds to component F_2. The typed instantiation of G_7 is:

$\$\$treatment(f1, \text{“call supervisor”}) \leftarrow \$sf(f1) \wedge \neg app_{psf}(f1) \wedge \neg app_{rvf}(f1).$
$\$\$treatment(f1, \text{“lower pressure”}) \leftarrow \$psf(f1) \wedge \$pressure(f1, high) \wedge \neg app_{rvf}(f1).$
$\$\$treatment(f1, \text{“open relief valve”}) \leftarrow \$rvf(f1) \wedge \$\$treatment(f1, \text{“lower pressure”}).$
$app_{rvf}(f1) \leftarrow \$rvf(f1) \wedge \$\$treatment(f1, \text{“lower pressure”}).$

$R_{N_7'}(G_7)$ is:

$\$\$treatment(f1, \text{“call supervisor”}) \leftarrow \neg app_{rvf}(f1).$
$\$\$treatment(f1, \text{“open relief valve”}) \leftarrow app_{rvf}(f1).$
$app_{rvf}(f1) \leftarrow \$\$treatment(f1, \text{“lower pressure”}).$

The model of $R_{N_7'}(G_7)$ is $N_7 = \{\$\$treatment(f1, \text{“call supervisor”})\}$. The reduction of each of the components of P' is locally stratified so P' is modularly stratified. The well-founded model of P' is $\{\$rvf(f1), \$psf(f1), \$sf(f1), \$\$treatment(f1, \text{“call supervisor”})\}$. Clearly the associated model of P and the well-founded model of P' are “equivalent”. $\Box$

The construction of the associated model of a Gulog program is analagous to the construction of the well-founded model of a Datalog program. This analogy allows us to claim that this is an appropriate intuitive semantics for Gulog programs.

The definition of “answer” and “correct answer” for m-stratified programs is a natural extension of the definition for i-stratified programs.

Proposition 6.1 *Let P be an m-stratified program and Q an atomic query $\Gamma \vdash \leftarrow A$. Let P' be the translation of P and Q' the translation of Q. If θ is a computed answer for $P' \cup \{Q'\}$, then θ is a correct answer for $P \cup \{Q\}$.*

Proof The query Q is translated to query $Q' = \leftarrow \Gamma' \wedge A'$, where Γ' is the translation of Γ and A' is the translation of A. Let M_P be the well-founded model of P and M_P' the well-founded model of P'. If θ is the computed answer for $P' \cup \{Q'\}$ then $M_P' \models (\Gamma' \wedge A')\theta$ and so $(\Gamma \vdash A)\theta$ is true in M_P (by lemma 6.3). $\Box$

We illustrate this with an example.

Example 6.4 Consider program P in Example 6.1 and the query $Q = \{x{:}rvf, y{:}string\} \vdash \leftarrow x[treatment \twoheadrightarrow y]$. The program is translated to P' in Example 6.3 and Q is translated to $Q' = \leftarrow \$rvf(x) \wedge \$string(y) \wedge \$\$treatment(x, y)$. $P' \cup \{Q'\}$ has computed answer $\{x/f1, y/$"call supervisor"$\}$, which corresponds to a correct answer for $P \cup \{Q\}$, $\{x : rvf, y : string\} \vdash \{x/f1, y/$"call supervisor"$\}$. $\Box$

Proposition 5.2 thus provides an alternative procedure to those described in [10] for evaluating Gulog queries. In particular, efficient goal directed procedures developed for Datalog may be used. Proposition 6.1 provides an evaluation procedure for a larger class of Gulog queries.

7 Conclusions

In this paper we described Gulog, a deductive object-oriented logic which captures inheritance with overriding, functional and set-valued methods, negation and multiple inheritance. We compared inheritance with overriding in Gulog programs, with negation in Datalog programs. We found there is a close relationship in that every Gulog program can be translated to an effectively equivalent Datalog program. We identified a restricted class of programs, i-stratified programs, which corresponds to local stratification of the program's translation. We showed the equivalence between the preferred model of an i-stratified program and the perfect model of its translation. We then extended i-stratification to m-stratification, which takes computation into account, and showed that m-stratification of a Gulog program corresponds to modular stratification of the program's translation. The equivalence between the total associated model of a Gulog program and the total well-founded model of its translation can be used to provide an alternative evaluation procedure to that described in [10] for a larger class of Gulog queries.

In the future we intend to study the consequences of extending Gulog with monotonic inheritance, more general multiple inheritance, arithmetic and aggregate functions, and derived types. Currently inheritance with overriding in Gulog is non-monotonic. Monotonic inheritance allows inherited values to accumulate. This is necessary, for example, when someone has many names. A person can have a name as a person, and another name as a writer, and they can both be valid names. In this paper we disallow ambiguities due to multiple inheritance. There are two other ways of dealing with multiple inheritance which we intend to investigate. The first involves defining an ordering on definitions at the schema level. The second involves roles, where the programmer can specify explicitly which definition to inherit in the program. Arithmetic operations are necessary in any serious system. Aggregates provide a way to group data before operating on it. Derived types facilitate views.

In conclusion, we believe that our simple model captures the main features of inheritance with overriding in a deductive object-oriented logic and that complex object-oriented concepts can be described in terms of simpler Datalog concepts. We also believe that adding other features to the logic will not alter these relationships.

Acknowledgement

We are grateful to Michael Lawley, Alexandre Lefebvre and Christopher Higgins for discussions and helpful suggestions.

References

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

[2] S. Abiteboul and S. Grumbach. COL: A logic-based language for complex objects. In F. Bancilhon and P. Buneman, editors, *Advances in Database Programming Languages*, pages 347–374. ACM Press/Addison-Wesley (Frontier Series), New York, 1990.

[3] S. Abiteboul and P. C. Kanellakis. Object identity as a query language primitive. In *Proc. of the ACM SIGMOD International Conference on the Management of Data*, pages 159–173, 1989.

[4] S. Abiteboul, G. Lausen, H. Uphoff, and E. Waller. Methods and rules. In *Proc. of the ACM SIGMOD International Conference on the Management of Data*, pages 32–41, Washington, DC, 1993.

[5] H. Ait-Kaci and R. Nasr. LOGIN: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185–215, 1986.

[6] C. Beeri. A formal approach to object-oriented databases. *Data and Knowledge Engineering*, 5:353–382, 1990.

[7] B. Chandrasekaran and W. Punch. Data validation during diagnosis, a step beyond traditional sensor validation. In *Proceedings AAAI-87 6th National Conference on Artificial Intelligence*, Seattle, Washington, July, 1987.

[8] W. Chen and D. Warren. C-Logic of complex objects. In *Proc. 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Philadelphia, PA, March, 1989.

[9] C. Delobel, M. Kifer, and Y. Masunaga, editors. *Proceedings of the Second International Conference on Deductive and Object-Oriented Databases*, Munich, Germany, 1991. Springer-Verlag. Published as Lecture Notes in Computer Science 566 by Springer-Verlag.

[10] G. Dobbie. *Foundations of Deductive Object-Oriented Database Systems*. thesis, University of Melbourne, 1994. In preparation.

[11] G. Dobbie and R. W. Topor. A model for inheritance and overriding in deductive object-oriented systems. In *Proc. 16th Australian Computer Science Conference*, pages 625–634, Brisbane, Australia, 1993.

[12] G. Dobbie and R. W. Topor. Representing inheritance and overriding in Datalog. In *Proc. of the Deductive Database Workshop in conjunction with ICLP'93*, Budapest, Hungary, 1993.

[13] 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.

[14] W. Kim, J.-M. Nicolas, and S. Nishio, editors. *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, Kyoto, Japan, 1989.

[15] F. G. McCabe. *Logic and Objects*. Prentice Hall, 1992.

[16] T. 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.

[17] 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.