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

On the Declarative and Procedural Semantics of Deductive Object-Oriented Systems

Gillian Dobbie and Rodney Topor

Technical Report CS-TR-94/12
24 May, 1994.

Abstract

We present declarative and procedural semantics for a deductive object-oriented language, Gulog. The declarative semantics is based on preferred minimal models. We describe both bottom-up and top-down query evaluation procedures and show that they are sound with respect to the declarative semantics. The results contribute to our understanding of the interaction of inheritance, overiding and deduction in the presence of both functional and set-valued methods, and multiple inheritance.

Publishing Information

A shortened version of this paper will appear in the Journal of Intelligent Information Systems in the Fall of 1994. The ideas were first presented at the Third International Conference on Deductive and Object-Oriented Databases, Dec. 1993

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

The aim of this paper is to propose a simple mathematical foundation for object-oriented systems with deduction. In particular, we present a declarative and procedural semantics for a simple language that includes classes, objects, functional and set-valued methods, (multiple) inheritance and overriding, with the aim of providing better understanding of how to effectively apply and implement these concepts.

Several proposals have previously addressed different aspects of this problem [1, 2, 3, 4, 7, 11, 17, 19, 24]. In general, this work has succeeded in providing insight into the interaction of some of these concepts, but they either describe their semantics by translation into logic programs, and thus providing little direct insight, or they do not address all the concepts.

In [14], we defined and studied a simple deductive object-oriented language, investigating the interaction of deduction, inheritance, and overriding in the simplest possible context. In particular, we considered only functional methods, definite clauses, and simple inheritance. In this paper, we extend this language with set-valued methods and multiple inheritance. These additions make the language more expressive and provide more powerful modelling facilities. The new language is called Gulog. Some of the definitions changed substantially with the addition of set-valued methods. We describe the reasons for these changes and use examples to highlight them. Our main contributions are the definition of a declarative semantics for this richer language and generalizations of our bottom-up and top-down query evaluation procedures.

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. The declarative semantics is based on Przymusinski's perfect model semantics for logic programs [27]. This semantics requires the definition of a priority relation between atoms (and between models). The priority relation between atoms is based on overriding as well as negation. As in Przymusinski's perfect model semantics the definition of preferred models is based on this priority relation. We also define bottom-up and top-down query evaluation procedures. The bottom-up method requires the program to be partitioned according to the priority relation and generalizes Apt et al.'s construction of the standard model of a stratified logic program [6]. The partitioning ensures that clauses defining overriding methods are lower than clauses defining methods which may be overridden. In the top-down procedure, a set of possible answers for a selected literal in a goal is computed, the subset of minimal answers according to a priority relation is selected, and the minimal answers are applied to the rest of the goal. Both procedures are sound with respect to the declarative semantics.

Our semantics for inheritance and overriding is equivalent to that of F-logic [17] and dynamic inheritance in [4]. However, we provide a new characterization of the semantics by showing its relationship to similar methods used in the semantics of logic programs. The approach to overriding in Gulog (and F-logic) is more general than that of *L&O* and the static inheritance in [4], as we allow a method in a superclass to be overridden by a method in a subclass for some instances of the subclass but not others, depending on whether the body of the method definition in the subclass succeeds or not. This generalization appears to be necessary for reasoning about inheritance hierarchies in artificial intelligence [28], if there is no way to describe views in the language [4].

In this paper, we extend the language described in our earlier work [14] with two new language features. The first extension is the inclusion of set-valued methods as introduced in F-logic [17]. This extension allows us to write programs which manipulate sets, though not to use set-valued variables. In both COL [2] and LDL [26], function symbols are used to represent sets. This enables the representation of sets of sets, which is impossible in Gulog or F-logic. The ability to represent sets of sets provides a richer data model, but the price is the need for a higher-order semantics. The second extension is the inclusion of multiple inheritance. We impose a restriction which allows a class to have several superclasses, but restricts each method to inherit its definition from at most one of its superclasses, as in [25]. We briefly discuss how this restriction may be lifted by defining a static ordering on the superclasses of a class, or by including annotations in the program which specify at run-time which superclass should be used for inheritance.

The paper is organised as follows. In Section 2, we introduce the syntax of Gulog. In Section 3, the declarative semantics of Gulog is described. Sections 4 and 5 outline the bottom-up and top-down query evaluation procedures. In Section 6, we briefly describe two approaches to resolving ambiguities that arise due to multiple inheritance. In the final section, we discuss the significance of this work and suggest further work in this area.

2 Syntax and informal semantics

In this section we describe the syntax of declarations, and of programs with respect to a set of declarations. We describe typed substitutions and a class of programs which we are particularly interested in, namely simple programs.

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. The syntax of Gulog is based on that of LOGIN [5] and is similar to the syntax of F-logic [17]. In order to concentrate on the semantics of inheritance and overriding in a well understood context, schema declarations are separated from data definitions, as in the relational approach. As in C++, we distinguish between objects and types (or classes). To maintain a general approach, method overriding can be defined on specific instances of a subtype. Our approach to set-valued methods is similar to that in F-logic. The syntax we use is also similar to that of F-logic, but we believe our semantics with respect to overriding is more natural than the description of overriding in the original version of F-logic, and can be implemented more efficiently [21]. The latest version of F-logic deals with inheritance of set-valued methods in the same way that we do.

Definition A *declaration* is defined as follows:

- If τ and τ' are type symbols then $\tau < \tau'$ is a *type lattice* declaration. We say τ is a subtype of τ', and τ' is a supertype of τ. For any type τ'' such that $\tau' < \tau''$, τ is also a subtype of τ'' and τ'' is a supertype of τ. If $\tau < \tau''$ or $\tau = \tau''$ we write $\tau \leq \tau''$.

- If a is an object symbol and τ is a type symbol then $a : \tau$ is an *object declaration*. We say a is an object of inherited type τ. For any type τ' such that τ' is a supertype of τ, a is also of inherited type τ'.

- If p is an n-ary predicate symbol and $\tau_1, \ldots, \tau_n$ are type symbols then $p(\tau_1, \ldots, \tau_n)$ is a *predicate declaration*. We say the signature of predicate p is $\tau_1 \times \cdots \times \tau_n$.

- If m is an n-ary functional method symbol and $\tau, \tau_1, \ldots, \tau_n, \tau'$ are type symbols then $\tau[m@\tau_1, \ldots, \tau_n \Rightarrow \tau']$ is a *functional method declaration*. We say the signature of functional method m is $\tau \times \tau_1 \times \cdots \times \tau_n \rightarrow \tau'$. If m is a 0-ary functional method symbol, we write $\tau[m \twoheadrightarrow \tau']$.

- If m is an n-ary set-valued method symbol and $\tau, \tau_1, \ldots, \tau_n, \tau'$ are type symbols then $\tau[m@\tau_1, \ldots, \tau_n \Rightarrow\!\!\!\!\!\!\Rightarrow \tau']$ is a *set-valued method declaration.* We say the signature of set-valued method m is $\tau \times \tau_1 \times \cdots \times \tau_n \twoheadrightarrow \tau'$. If m is a 0-ary set-valued method symbol, we write $\tau[m \Rightarrow\!\!\!\!\!\!\Rightarrow \tau']$.

For each object declaration, we introduce another type to the type lattice. Let $a : \tau$ be an object declaration, we introduce type a, such that $a < \tau$ and object a is of type a. With the new type, we are able to generalise the results in [14] to apply to methods which act only on one instance. The relation $<$ is a partial order on the set of type symbols. There is exactly one declaration for each object symbol, predicate symbol, functional method symbol and set-valued method symbol.

We now define data definitions. Data is defined by a program with respect to a set of declarations. As described above, the declarations define a set of types arranged in a lattice, a set of typed object symbols, a set of functional method symbols each with a given signature, a set of set-valued method symbols each with a given signature, and a set of predicate symbols each with a given signature. We do not assume that the type lattice has a top or bottom element. It is possible for the same symbol to be used for both functional and set-valued methods.

In Gulog, variables are typed.

Definition If x is a variable and τ is a type then $x : \tau$ is a *typed variable* defining that variable x is of type τ. A *variable typing* is a set of typed variables where each variable is distinct and will be denoted by Γ.

In this paper, we say an object or variable t is of type τ if τ is its minimum type. If t is of type σ and $\sigma < \tau$ then t is of inherited type τ.

The definitions of terms, atoms, and clauses are similar to those in logic programming except that in Gulog we include type information.

Definition A *term* is defined as follows:

- $\Gamma \vdash x : \tau$ is a term if x is a variable and $x : \tau \in \Gamma$. We usually write $\Gamma \vdash x$ when the type of x is obvious. The variable x can be assigned an object o if and only if o is of inherited type τ.

- $\Gamma \vdash a : \tau$ is a term if a is an object of type τ. We write the term simply as a when the type of a is clear.

The term $\Gamma \vdash t : \tau$ is called a *ground term* if t is not a variable.

Definition An *atom* is defined as follows:

- If p is an n-ary predicate symbol with signature $\tau_1 \times \cdots \times \tau_n$ and there exists a variable typing Γ that contains type assignments for all the variables in $t_1, \ldots, t_n$, and $\Gamma \vdash t_1 : \sigma_1, \ldots, \Gamma \vdash t_n : \sigma_n$ are terms with $\sigma_1 \leq \tau_1, \ldots, \sigma_n \leq \tau_n$, then $\Gamma \vdash p(t_1, \ldots, t_n)$ is a *predicate atom.*

- If m is an n-ary functional method symbol with signature $\tau \times \tau_1 \times \cdots \times \tau_n \rightarrow \tau'$ and there exists a variable typing Γ that contains type assignments for all the variables in $t, t_1, \ldots, t_n$ and t', and $\Gamma \vdash t : \sigma, \Gamma \vdash t_1 : \sigma_1, \ldots, \Gamma \vdash t_n : \sigma_n$, and $\Gamma \vdash t' : \sigma'$ are terms with $\sigma \leq \tau$, $\sigma_1 \leq \tau_1, \ldots, \sigma_n \leq \tau_n$ and $\sigma' \leq \tau'$, then $\Gamma \vdash t[m@t_1, \ldots, t_n \rightarrow t']$ is a *functional method atom.* We say method m is defined on type σ.

- If m is an n-ary set-valued method symbol with signature $\tau \times \tau_1 \times \cdots \times \tau_n \twoheadrightarrow \tau'$ and there exists a variable typing Γ that contains type assignments for all the variables in $t, t_1, \ldots, t_n$ and t', and $\Gamma \vdash t : \sigma, \Gamma \vdash t_1 : \sigma_1, \ldots, \Gamma \vdash t_n : \sigma_n$, and $\Gamma \vdash t' : \sigma'$ are terms with $\sigma \leq \tau$, $\sigma_1 \leq \tau_1, \ldots, \sigma_n \leq \tau_n$ and $\sigma' \leq \tau'$, then $\Gamma \vdash t[m@t_1, \ldots, t_n \twoheadrightarrow t']$ is a *set-valued method atom*. We say method m is defined on type σ.

An atom is a *method atom* if it is a functional method atom or a set-valued method atom. The atom $\Gamma \vdash A$ is a *ground atom* if A contains no variables. Let $\rightsquigarrow$ denote $\rightarrow$ or $\twoheadrightarrow$, and M be a 0-ary method atom $\Gamma \vdash t[m@ \rightsquigarrow t']$. M can be written $\Gamma \vdash t[m \rightsquigarrow t']$.

A clause also has type information.

Definition If $\Gamma_A \vdash A, \Gamma_{B_1} \vdash B_1, \ldots, \Gamma_{B_n} \vdash B_n$ are atoms where the variables common to $\Gamma_A, \Gamma_{B_1}, \ldots, \Gamma_{B_n}$ have the same types then $\Gamma_A \cup \Gamma_{B_1} \cup \cdots \cup \Gamma_{B_n} \vdash (A \leftarrow B_1 \wedge \cdots \wedge B_n)$ is a *clause*. A is the *head* of the clause. $B_1 \wedge \cdots \wedge B_n$ is the *body* of the clause.

We can write the following compound atoms:

- $\Gamma \vdash t[m_1@t_{1,1}, \ldots, t_{1,n} \rightarrow t_1; \ldots; m_j@t_{j,1}, \ldots, t_{j,k} \rightarrow t_j]$, as a shorthand for $\Gamma \vdash t[m_1@t_{1,1}, \ldots, t_{1,n} \rightarrow t_1] \wedge \cdots \wedge t[m_j@t_{j,1}, \ldots, t_{j,k} \rightarrow t_j]$.
- $\Gamma \vdash t[m_1@t_{1,1}, \ldots, t_{1,n} \rightarrow t'[m_j@t_{j,1}, \ldots, t_{j,k} \rightarrow t_j]]$, as a shorthand for $\Gamma \vdash t[m_1@t_{1,1}, \ldots, t_{1,n} \rightarrow t'] \wedge t'[m_j@t_{j,1}, \ldots, t_{j,k} \rightarrow t_j]$.
- $\Gamma \vdash (t[m@t_1, \ldots, t_n \twoheadrightarrow \{t'_1, \ldots, t'_p\}]$, as a shorthand for $\Gamma \vdash t[m@t_1, \ldots, t_n \twoheadrightarrow t'_1] \wedge \cdots \wedge t[m@t_1, \ldots, t_n \twoheadrightarrow t'_p]$.

If $a_1, \ldots, a_p$ are object symbols we can write $\Gamma \vdash t[m@t_1, \ldots, t_n \twoheadrightarrow \{a_1, \ldots, a_p\}] \leftarrow B_1 \wedge \cdots \wedge B_m$ as a shorthand for the p clauses:

$$\begin{array}{l} \Gamma \vdash t[m@t_1, \ldots, t_n \twoheadrightarrow a_1] \leftarrow B_1 \wedge \cdots \wedge B_m \\ \vdots \\ \Gamma \vdash t[m@t_1, \ldots, t_n \twoheadrightarrow a_p] \leftarrow B_1 \wedge \cdots \wedge B_m. \end{array}$$

If $\Gamma \vdash t[m \rightarrow y]$ (resp. $\Gamma \vdash t[m \twoheadrightarrow y]$) is an atom, where $\Gamma \vdash t : \sigma$ is a term then this atom is of type σ. The type of a predicate atom is undefined. If the head of a clause is an atom of type σ, then the clause is of type σ.

Definition A *goal* is a clause of the form $\Gamma \vdash \leftarrow B_1 \wedge \cdots \wedge B_n$, i.e., a clause with an empty head. If Γ is empty we write $\leftarrow B_1 \wedge \cdots \wedge B_n$.

Definition A *program* with respect to a set of declarations is a finite set of clauses. Let P be a program with respect to a set of declarations D. Then $[P]$ with respect to D denotes the set of all ground instances of clauses of P with respect to D. We write P (resp. $[P]$) when the set of declarations under consideration is clear.

Example 2.1 Consider a database which records information about failures in reactor systems. There is a lattice of failures where *total failure* (tf) is a special kind of *feed system failure* (fsf), *total failure* is also a special kind of *condenser failure* (cf), *feed system failure* and *condensor failure* are special kinds of *cooling system failure* (csf), and *cooling system failure* is a special kind of *system failure* (sf). When there is a failure, there are inputs from the system which indicate if measurements like *temperature* and *flow* are *high* or *low*. In this example we record the failure identifier and the date of the failure. We assume the existence of a function which generates today's date and an equivalence predicate. Consider the declarations D:

$r1 : reactor$	$temp\_sensor : part$
$high : degree$	$low : degree$
$f1 : tf$	$csf < sf$
$cf < csf$	$fsf < csf$
$tf < cf$	$tf < fsf$
$reactor[failure@date \Rrightarrow sf]$	$sf[problem \Rightarrow string]$
$sf[temp \Rightarrow degree]$	$sf[flow \Rightarrow degree]$

and the Gulog program P wrt declarations D:

$$
\begin{array}{l}
\{x : reactor, d : date, y : sf, z : string\} \vdash x[failure@d \twoheadrightarrow y] \leftarrow \\
\quad y[problem \rightarrow z] \wedge d = today \\
\{x : sf\} \vdash x[problem \rightarrow \text{“system failure”}] \\
\{x : csf\} \vdash x[problem \rightarrow \text{“cooling system failure”}] \leftarrow x[temp \rightarrow high] \\
\{x : fsf\} \vdash x[problem \rightarrow \text{“cooling system flow high”}] \leftarrow x[flow \rightarrow high] \\
f1[temp \rightarrow high] \\
f1[flow \rightarrow high].
\end{array}
$$

The goal $\{x : reactor, y : date, z : sf\} \vdash \leftarrow x[failure@y \twoheadrightarrow z]$ asks for x, y and z such that reactor x has a system failure z on date y. $\Box$

We define substitutions which are similar to those in logic programming except that they are typed and hence satisfy additional restrictions.

Definition Let $\Gamma \vdash \{x_1/t_1, \ldots, x_n/t_n\}$ be a pair where $x_1, \ldots, x_n$ are distinct variables, $\Gamma \vdash t_1, \ldots, \Gamma \vdash t_n$ are terms, no t_i equals x_i.

$\Gamma \vdash \{x_1/t_1, \ldots, x_n/t_n\}$ is a *substitution* if for each i, $\Gamma \vdash x_i : \tau_i$ and $\Gamma \vdash t_i : \sigma_i$ are terms, and $\sigma_i \leq \tau_i$.

The empty substitution is denoted by ε.

Definition An *expression* $(\Gamma \vdash T)$ is a term, a predicate atom, a method atom, a conjunction of atoms or a clause. A *simple expression* is a term, a predicate atom, or a method atom.

Definition Let $\theta = (\Gamma_\theta \vdash \{x_1/t_1, \ldots, x_k/t_k\})$ be a substitution and $E = (\Gamma_E \vdash T)$ an expression. For any i, such that $1 \leq i \leq k$, let $\Gamma_\theta \vdash x_i : \tau_i$ and $\Gamma_\theta \vdash t_i : \sigma_i$ be terms and $x_i : \tau'_i$ be a typed variable in E (with respect to Γ_E). We say $dom(\theta) = \{t_1, \ldots, t_k\}$ and $vars(T)$ is the set of variables in T.

Construct T' from T by simultaneously replacing each occurrence of a variable x_i in T by t_i.

Construct Γ', a variable typing for the variables in T', as follows. For each variable x in T', if x is some variable t_i, then the typing of x in Γ' is $x : glb(\sigma_i, \tau'_i)$; otherwise the typing of x in Γ' is just that in Γ_E.

If for each variable x_i in $vars(T) \cap dom(\theta)$, t_i is a variable and $glb(\sigma_i, \tau'_i)$ exists, or t_i is an object and $\sigma_i \leq \tau'_i$ then the *instance* $E\theta$ of E is $\Gamma' \vdash T'$; otherwise, $E\theta$ is undefined.

If $E\theta$ is ground, then $E\theta$ is a *ground instance* of E.

Example 2.2 Assume D is the declarations in Example 2.1 and the object declaration $f2{:}sf$. Let E be $\{x{:}fsf, y{:}string\} \vdash x[problem \rightarrow y]$ and substitution θ be $\{x{:}sf, x'{:}cf\} \vdash \{x/x'\}$ wrt declarations D then the instance $E\theta$ is $\{x' : tf, y : string\} \vdash x'[problem \rightarrow y]$.

Let θ_1 be $\{x : sf\} \vdash \{x/f2\}$. $E\theta_1$ is undefined because $sf \not\leq fsf$. $\Box$

Let $S = \{E_1, \ldots, E_n\}$ be a finite set of expressions and θ a substitution. Then $S\theta$ denotes the set $\{E_1\theta, \ldots, E_n\theta\}$.

Definition Let θ' be $\{x_1/t_1, \ldots, x_n/t_n\}$, σ' be $\{y_1/s_1, \ldots, y_m/s_m\}$, and θ and σ be the substitutions $\Gamma_\theta \vdash \theta'$ and $\Gamma_\sigma \vdash \sigma'$.

Construct γ from $\{x_1/t_1\sigma, \ldots, x_n/t_n\sigma, y_1/s_1, \ldots, y_m/s_m\}$ (with respect to Γ_θ and Γ_σ) by deleting any binding $x_i/t_i\sigma$ for which $x_i = t_i\sigma$, and deleting any binding y_j/s_j for which $y_j \in \{x_1, \ldots, x_n\}$.

Construct Γ, the variable typing for the variables in γ as follows. For each variable x in γ, if $x : \tau \in \Gamma_\theta \cup \Gamma_\sigma$ then include $x : \tau$ in Γ, otherwise if $x : \tau \in \Gamma_\theta$ and $x : \tau' \in \Gamma_\sigma$ then include $x : glb(\tau, \tau')$ in Γ.

If for any variable x, where $x : \tau \in \Gamma_\theta$ and $x : \tau' \in \Gamma_\sigma$, $glb(\tau, \tau')$ does not exist, then $\theta\sigma = \emptyset$ else $\theta\sigma = \Gamma \vdash \gamma$.

$\theta\sigma$ is the *composition* of θ and σ.

To ensure that programs have well-defined semantics, we impose restrictions that ensure they are "stratified" and that functional methods really are single-valued. Programs that satisfy these restrictions are called "simple". Before defining the classes of stratified and simple programs, we must define a relationship "possibly overrides" between clauses.

Definition Let D be a set of declarations, P be a program with respect to D, C be $\Gamma \vdash x[m@y_1, \ldots, y_k \rightarrow z] \leftarrow B$, (resp. $\Gamma \vdash x[m@y_1, \ldots, y_k \twoheadrightarrow z] \leftarrow B$) and C' be $\Gamma' \vdash x'[m@y'_1, \ldots, y'_k \rightarrow z'] \leftarrow B'$ (resp. $\Gamma' \vdash x'[m@y'_1, \ldots, y'_k \twoheadrightarrow z'] \leftarrow B'$), where $C, C' \in P$, $\Gamma \vdash x : \sigma$ is a term, $\Gamma' \vdash x' : \tau$ is a term and $\sigma < \tau$. If there are ground instances $C\theta$ of C and $C'\theta'$ of C' such that $x\theta = x'\theta'$, $y_i\theta = y'_i\theta'$ for all i, $1 \leq i \leq k$, and $z\theta \neq z'\theta'$ then clause $C\theta$ *possibly overrides* clause $C'\theta'$ with respect to P and D.

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 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,

- $\mu(A') \leq \mu(A)$, where A' is an atom 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 an atom 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 2.3 Assume the declarations in Example 2.1. Consider the following program P wrt declarations D:

$$\{x : sf\} \vdash x[problem \rightarrow \text{"system failure"}]$$
$$\{x : csf\} \vdash x[problem \rightarrow \text{"cooling system failure"}] \leftarrow x[problem \rightarrow \text{"system failure"}].$$

Then $[P]$ is:

$$f1[problem \rightarrow \text{"system failure"}] \qquad (1)$$

$$f1[problem \rightarrow \text{“cooling system failure”}]$$
$$\leftarrow f1[problem \rightarrow \text{“system failure”}]. \quad (2)$$

Method *problem* is a functional method. It has either the value “system failure” or “cooling system failure”. If it has value “system failure” then it should also have value “cooling system failure” (due to deduction). If it doesn't have value “cooling system failure” then it should have value “system failure” (due to overriding). This is contradictory. Thus we conclude that program P has no natural meaning. Based on the above declarations, we say clause 2 possibly overrides clause 1. The relationship of the ground atoms of $[P]$ requires:

$$\mu(f1[problem \rightarrow \text{“system failure”}]) \leq \mu(f1[problem \rightarrow \text{“cooling system failure”}])$$
$$\mu(f1[problem \rightarrow \text{“cooling system failure”}]) \leq \mu(f1[problem \rightarrow \text{“system failure”}])$$
$$\mu(f1[problem \rightarrow \text{“system failure”}]) < \mu(f1[problem \rightarrow \text{“system failure”}]).$$

As expected P is not i-stratified. □

This definition of i-stratification does differ from that in [14] where there were only functional methods. This is because we were interested in programs where each functional method was defined in only one clause. A set-valued method can be defined in one or more clauses. The following example illustrates the necessity to change the definition.

Example 2.4 Assume appropriate declarations D including those in Example 2.1. Consider program P wrt D:

$$\{x : sf\} \vdash x[severity \twoheadrightarrow high]$$
$$\{x : sf\} \vdash x[severity \twoheadrightarrow low] \leftarrow x[treatment \rightarrow \text{“tell supervisor”}]$$
$$\{x : csf\} \vdash x[severity \twoheadrightarrow high] \leftarrow x[treatment \rightarrow \text{“turn off condenser”}]$$
$$\{x : csf\} \vdash x[severity \twoheadrightarrow low] \leftarrow x[treatment \rightarrow \text{“check history”}].$$

Then $[P]$ is:

$$f1[severity \twoheadrightarrow high] \quad (3)$$
$$f1[severity \twoheadrightarrow low] \leftarrow f1[treatment \rightarrow \text{“tell supervisor”}] \quad (4)$$
$$f1[severity \twoheadrightarrow high] \leftarrow f1[treatment \rightarrow \text{“turn off condenser”}] \quad (5)$$
$$f1[severity \twoheadrightarrow low] \leftarrow f1[treatment \rightarrow \text{“check history”}]. \quad (6)$$

In program $[P]$, clause 6 possibly overrides clause 3, and clause 5 possibly overrides clause 4. Using the definition of i-stratification in [14], $\mu(f1[severity \twoheadrightarrow low]) < \mu(f1[severity \twoheadrightarrow high])$ and $\mu(f1[severity \twoheadrightarrow high]) < \mu(f1[severity \twoheadrightarrow low])$. Based on that definition, this program is not i-stratified even though there are no cycles in the data. Based on the definition of i-stratification given above, $\mu(f1[severity \twoheadrightarrow low]) \leq \mu(f1[severity \twoheadrightarrow high])$, $\mu(f1[severity \twoheadrightarrow high]) \leq \mu(f1[severity \twoheadrightarrow low])$, $\mu(f1[treatment \rightarrow \text{“check history”}] < \mu(f1[severity \twoheadrightarrow high])$, $\mu(f1[treatment \rightarrow \text{“turn off condenser”}] < \mu(f1[severity \twoheadrightarrow low])$, $\mu(f1[treatment \rightarrow \text{“check history”}] \leq \mu(f1[severity \twoheadrightarrow low])$, $\mu(f1[treatment \rightarrow \text{“turn off condenser”}] \leq \mu(f1[severity \twoheadrightarrow high])$, $\mu(f1[treatment \rightarrow \text{“tell supervisor”}] \leq \mu(f1[severity \twoheadrightarrow low])$. As expected P is i-stratified. □

We now present examples which illustrate ambiguities which can occur due to multiple inheritance. The first illustrates possible schema ambiguities and the second possible data ambiguities.

Example 2.5 Consider the system description in Example 2.1 and the following declarations:

$csf < sf$	$cf < csf$
$fsf < csf$	$tf < cf$
$tf < fsf$	
$tf\_error < fsf\_error$	$tf\_error < cf\_error$
$cf[problem \Rightarrow cf\_error]$	$fsf[problem \Rightarrow fsf\_error]$
"call supervisor" $: tf\_error$	"check history" $: tf\_error.$

The type of the value of method *problem* when applied to an object of type *tf* could be *cf_error* inherited from *cf*, *fsf_error* inherited from *fsf*, or *cf_error* and *fsf_error* inherited from *cf* and *fsf*. In Gulog the type must be compatible with both parents. That is, the type of the value of method *problem* when applied to an object of type tf is *tf_error*. □

Thus inheritance of signatures is monotonic as in F-logic.

The following example illustrates alternatives when values of methods are inherited from more than one parent.

Example 2.6 Now consider program P wrt the declarations in Example 2.5:

$$\{x : cf\} \vdash x[problem \rightarrow \text{"call supervisor"}]$$
$$\{x : fsf\} \vdash x[problem \rightarrow \text{"check history"}].$$

An object of type *tf* could inherit the value "call supervisor" for method *problem* from *cf* or inherit the value "check history" for method *problem* from *fsf* or inherit both values. □

For simplicity, we disallow programs with ambiguities due to multiple inheritance in this paper. In Section 6, we briefly discuss alternative ways of dealing with multiple inheritance which are less restrictive but more complicated.

Definition Let P be a program and D a set of declarations. For each type σ in D, $R(\sigma)$ is $\{< m_{\rightarrow}, n, \tau >|$ functional method $m_{\rightarrow}$ of arity n is defined on type τ, and $\sigma \leq \tau\} \cup \{< m_{\twoheadrightarrow}, n, \tau >|$ set-valued method $m_{\rightarrow}$ of arity n is defined on type τ, and $\sigma \leq \tau\}$. Program P is *unambiguous* with respect to D iff for every type σ in D, if $< m_{\rightarrow}, n, \tau_1 >\in R(\sigma)$ (resp. $< m_{\twoheadrightarrow}, n, \tau_1 >\in R(\sigma)$) and $< m_{\rightarrow}, n, \tau_2 >\in R(\sigma)$ (resp. $< m_{\twoheadrightarrow}, n, \tau_2 >\in R(\sigma)$) then either $\tau_1 < \tau_2$ or $\tau_2 < \tau_1$.

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* with respect to 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 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 and x in the conjunction $a[m'@x \rightarrow y] \wedge a[m''@y \rightarrow z]$.

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

- For each clause $\Gamma \vdash t[m@t_1, \ldots, t_n \rightarrow u] \leftarrow C$ in P with respect to D, u is either an object symbol or a variable that is restricted with respect to 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 with respect to D 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 with respect to D is unambiguous with respect to multiple inheritance.

Example 2.7 Assume appropriate declarations D. Consider program P wrt D:

$$\{x : csf\} \vdash x[problem \rightarrow \text{“cooling system failure”}] \leftarrow x[temp \rightarrow high]$$
$$\{x : csf\} \vdash x[problem \rightarrow \text{“cooling system failure”}] \leftarrow x[temp \rightarrow low].$$

This program is not well-defined because $\{x : csf, z : string\} \vdash x[problem \rightarrow z]$ and $\{x : csf, z' : string\} \vdash x[problem \rightarrow z']$ are unifiable. □

Definition A program P with respect to a set of declarations D is *simple* if the following two conditions hold:

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

3 Semantics

In this section we describe the declarative semantics of a program. We define a typed Herbrand interpretation and model. 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.

We start by defining an interpretation of a program P with respect to a set of declarations D. The interpretation assigns a truth value to every ground instance of a clause in P.

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

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

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

Definition Let $\Gamma \vdash t : \tau$ be a term and $\mathcal{V}$ a variable assignment (with respect to Γ). Then the *term assignment* $\mathcal{T}$ (based on $\mathcal{V}$ with respect to Γ) is the assignment to each term $\Gamma \vdash t : \tau$ of an object a 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, $\Gamma \vdash A$ an atom, and $\mathcal{T}$ a term assignment (with respect to Γ). Then an atom is given a *truth value* in I (with respect to $\mathcal{T}$ and Γ) as follows:

- If A is $p(t_1, \ldots, t_n)$, and $t'_1, \ldots, t'_n$ are the term assignments of $t_1, \ldots, t_n$ (with respect to $\mathcal{T}$ and Γ), then $\Gamma \vdash p(t_1, \ldots, t_n)$ is true in I (with respect to $\mathcal{T}$ and Γ) if $p(t'_1, \ldots, t'_n) \in I$.

- If A is $t[m@t_1, \ldots, t_k \rightarrow t_n]$, 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 $\Gamma \vdash t[m@t_1, \ldots, t_k \rightarrow t_n]$ 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 $t[m@t_1, \ldots, t_k \twoheadrightarrow s]$ is a set-valued method atom, 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 $\Gamma \vdash t[m@t_1, \ldots, t_k \twoheadrightarrow s]$ is true in I (with respect to $\mathcal{T}$ and Γ) if $t'[m@t'_1, \ldots, t'_k \twoheadrightarrow s'] \in I$.

A ground instance of a clause is given a truth value in I (with respect to $\mathcal{T}$ and Γ) in the normal way.

Because of the presence of inheritance and overriding, we require one more concept before we can define a "model" of a program.

Definition Let P be a program with respect to a set of declarations D 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.

Definition Let P be a program with respect to a set of declarations D, 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 $I \models P$.

Note that a program with respect to a set of declarations may have more than one model and that not every ground instance of a clause need be true in a model. As in [27], we can define a priority relationship between ground atoms (but using possible overriding instead of negation), and use this to define a preference relationship between models. As in [27], 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 a 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, we define the following rules:

- $A \leq_p A'$, where A is the head of $C\theta$, and A' is a ground atom in the body of $C\theta$,

- $A <_p B'$ 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 an atom in the body of $C'\theta'$,

- $A \leq_p A'$ 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 $C'\theta'$.

and extend it with the following transitivity rules:

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

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 with respect to a set of declarations D. Then N is *preferable* to M ($N \ll M$) if, for every atom A in $N \backslash M$, there is an atom B in $M \backslash N$ such that $A <_p B$. 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 3.1 Assume appropriate declarations D including those in Example 2.1. Consider program P wrt D:

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

and the interpretations:

$$I_1 = \{f1[treatment \rightarrow \text{“turn off condenser”}], f1[severity \twoheadrightarrow low]\},$$
$$I_2 = \{f1[treatment \rightarrow \text{“tell supervisor”}], f1[severity \twoheadrightarrow low], f1[severity \twoheadrightarrow high]\},$$
$$I_3 = \{f1[severity \twoheadrightarrow high]\}.$$

Since all the ground instances of clauses of P are true or possibly overridden in each of these interpretations, I_1, I_2, and I_3 are all models of P. The priority relationship gives the following relationships between the ground atoms of P:

$$f1[severity \twoheadrightarrow low] \leq_p f1[treatment \rightarrow \text{“turn off condenser”}]$$
$$f1[severity \twoheadrightarrow low] \leq_p f1[treatment \rightarrow \text{“tell supervisor”}]$$
$$f1[severity \twoheadrightarrow high] \leq_p f1[severity \twoheadrightarrow low]$$
$$f1[severity \twoheadrightarrow high] <_p f1[treatment \rightarrow \text{“turn off condenser”}]$$

Because $f1[severity \twoheadrightarrow high] <_p f1[treatment \rightarrow \text{“turn off condenser”}]$, I_3 is preferable to I_1. I_3 is preferable to I_2 because there are no atoms in $I_3 \backslash I_2$. As there are no other models preferable to I_3, I_3 is the preferred model, that is $M_P = I_3$. □

Example 3.2 This example illustrates that subsets in subtypes do override sets in supertypes. You will recall that subsets in subtypes did not override sets in supertypes in F-logic before it was revised. Assume a set of declarations including those in Example 2.1. Consider the program:

$$\{x : sf\} \vdash x[severity \twoheadrightarrow \{high, low\}]$$
$$\{x : csf\} \vdash x[severity \twoheadrightarrow \{low\}]$$

This program has two models, $I_1 = \{f1[severity \twoheadrightarrow high], f1[severity \twoheadrightarrow low]\}$, and $I_2 = \{f1[severity \twoheadrightarrow low]\}$. Because there are no atoms in $I_2 \backslash I_1$, I_2 is preferable to I_1 as expected. □

The following results are analogous to Przymusinski's results for stratified logic programs [27]. They motivate our definition of simple programs.

Lemma 3.1 *Let P be a simple program with respect to a set of declarations D in which no clause is possibly overridden by any other clause. Then P with respect to D has a unique minimal model, which is a preferred model.*

Proof Let $\{M_i\}$ be the set of models of P wrt D. The intersection of the models $\cap\{M_i\}$ is a model because the ground instances of clauses of P wrt D are true in $\cap\{M_i\}$ and no clause is overridden by any other clause in P wrt D. In fact the minimal model, M_P, of P wrt D is $\cap\{M_i\}$. Because M_P is the intersection of all the models of P wrt D, it must be unique. Let $M \in \{M_i\}$. There is no atom $A \in M_P \backslash M$, so $M_P \ll M$, M_P is the preferred model of P wrt D. $\Box$

Before proving the following lemma we define "conflicts", "(clause, type) pairs" and an ordering on (clause, type) pairs.

Definition Let A' be the ground method atom $a[m@a_1, \dots, a_k \rightarrow b']$ and A the ground method atom $a[m@a_1, \dots, a_k \rightarrow b]$ (resp. let A' be $(a[m@a_1, \dots, a_k \twoheadrightarrow b']$ and A be $(a[m@a_1, \dots, a_k \twoheadrightarrow b])$. We say A *conflicts* with A' if $b \neq b'$.

We define ground "(clause, type) pairs", an ordering on them, and the "level" of a program.

Definition Let P be a program with respect to a set of declarations D. The set of ground *(clause, type) pairs* of P with respect to D are $\{(p, \tau) \mid p$ is a ground instance of a clause in P with respect to D and τ is the type of the clause of which p is a ground instance$\}$.

Definition The ground (clause, type) pairs of an i-stratified program can be *partitioned* into levels. Let P be a program with respect to a set of declarations D. For all clauses $\Gamma \vdash A \leftarrow B$ of type τ and $\Gamma' \vdash A' \leftarrow B'$ of type τ' in $[P]$ with respect to D, if $\mu(A) < \mu(A')$ or $\Gamma \vdash A \leftarrow B$ possibly overrides $\Gamma' \vdash A' \leftarrow B'$ then $(A \leftarrow B, \tau) < (A' \leftarrow B', \tau')$.

Definition Let $P_1^t \cup \cdots \cup P_k^t$ be a partitioning on the ground (clause, type) pairs of an i-stratified program P with respect to a set of declarations D. Then P with respect to D has *level* k.

Lemma 3.2 *Let P be a simple program with respect to a set of declarations D and $P_1^t \cup \cdots \cup P_{k+1}^t$ be a partitioning of the ground (clause, type) pairs of P as described above. Let P_i be the set of ground instances of clauses of P_i^t for $1 \leq i \leq k+1$. Suppose $P_1 \cup \cdots \cup P_k$ has a unique preferred model M_k. Then there exists a unique minimal extension of M_k to a preferred model M_{k+1} of P.*

Proof We say a (clause, type) pair (C, τ) conflicts with an (atom, type) pair (A', τ'), if the head of clause C conflicts with atom A' and $\tau > \tau'$. Let $M|_i$ be the restriction of the set M to the atoms in the heads of ground instances of clauses in $P_1 \cup \cdots \cup P_i$. Let M_i^t be a set of (atom, type) pairs (A, τ) where A is in M_i and τ is the type of the ground instance of the clause of which A is the head.

Consider the set P_{k+1}^t of ground (clause, type) pairs. Modify P_{k+1}^t by removing all (clause, type) pairs whose head conflicts with an (atom, type) pair in M_k^t. Call the resulting set of clauses P_{res}^t. Let $P_{res} = \{C \mid (C, \tau) \in P_{res}^t\}$.

Consider the program $P^* = P_{res} \cup M_k$. No clause in P^* wrt D is possibly overridden by another clause in P^* wrt D. By Lemma 3.1, P^* wrt D has a unique minimal model, M_{k+1}. We now show that M_{k+1} is the unique preferred model of P wrt D.

We first want to prove that M_{k+1} is a model of P wrt D. We know that M_{k+1} is a model of $P^* = P_{res} \cup M_k$. Let $C = A \leftarrow A_1 \wedge \cdots \wedge A_n \in P_{k+1}$ and suppose $A_1 \wedge \cdots \wedge A_n$ is true in

M_{k+1}. If $C \in P_{res}$ then C is true in M_{k+1}. If $C \notin P_{res}$ then C is overridden by some clause wrt $P_1 \cup \cdots \cup P_k$, D and M_{k+1}.

We next want to prove that M_{k+1} is a preferred model of P wrt D. Suppose M_{k+1} is not a preferred model of P wrt D. Then there exists another model M of P wrt D such that $M \ll M_{k+1}$. That is, $\forall A \in M \backslash M_{k+1}$, $\exists B \in M_{k+1} \backslash M$ such that $A < B$. First note that $M|_k = M_k$. For if $M|_k \subset M_k$, then $M|_k$ is a model of $P_1 \cup \cdots \cup P_k$ wrt D such that $M|_k \ll M_k$, then M_k would not be the preferred model of $P_1 \cup \cdots \cup P_k$ wrt D. If $M_k \subset M|_k$, then M differs from M_{k+1} at a level less than $k+1$, and it would not be possible for M to be a preferred model of P wrt D. Then M is an extension of M_k. But M_{k+1} is the minimal extension of M_k. So M_{k+1} and M only differ on level $k+1$ atoms. As M_{k+1} is the minimal unique extension of M_k it is impossible that $M \ll M_{k+1}$.

Finally we want to prove that M_{k+1} is the only preferred model of P wrt D. Suppose there exists some preferred model M of P wrt D. Then $M \not\ll M_{k+1}$ and $M_{k+1} \not\ll M$. As above $M|_k = M_k$. Because M_{k+1} is the minimal unique extension of M_k then $M_{k+1} \ll M$ which is a contradiction. □

Theorem 3.1 *For every model N of a simple program P with respect to a set of declarations D, there exists a preferred model M such that $M \leq N$.*

Every simple program has exactly one preferred model, which we denote M_P, and for every other model M, we have $M_P \ll M$.

Proof We prove this theorem by induction on the level k of program P wrt D.

If k = 1, then P_1 is a program wrt D in which no clause is overridden by any other clause. In Lemma 3.1 we proved that P_1 wrt D has a preferred model M_1, and M_1 is unique.

Assume that the result holds for $P_1 \cup \cdots \cup P_k$ wrt D, then it has a unique preferred model M_k.

If P wrt D has maximum level $k+1$, we proved in Lemma 3.2 that if M_k is a preferred model of $P_1 \cup \cdots \cup P_k$ wrt D, there is an extension of M_k to M_{k+1} which is a preferred model of $P_1 \cup \cdots \cup P_{k+1}$ wrt D and hence P wrt D. By Lemma 3.2 and the induction hypothesis, P wrt D has a preferred model M_{k+1} and M_{k+1} is unique. □

Theorem 3.2 *Let P be a simple program with respect to a set of declarations D. Then M_P does not depend on the particular i-stratification chosen for P.*

Proof There may be many i-stratifications of a program. The definition of preferred model is independent of a particular i-stratification. □

This preferred model semantics agrees with our intuition about the interaction between inheritance, overriding and deduction. We can now give the definition of a correct answer for a program and a goal.

Definition Let P be a simple program with respect to a set of declarations D and G a goal. An *answer* for $P \cup \{G\}$ is a ground substitution for variables of G. Let G be the goal $\Gamma \vdash \leftarrow A_1 \wedge \cdots \wedge A_k$, and θ an answer for $P \cup \{G\}$. Then θ is a *correct answer* for $P \cup \{G\}$ if $(A_1 \wedge \cdots \wedge A_k)\theta$ is true in M_P.

4 Bottom-up evaluation

In [6], Apt et al. defined a bottom-up procedure to compute the standard model of a stratified logic program. In this section we describe a similar bottom-up procedure to evaluate the preferred model of a simple program.

This procedure differs from the one defined in [6]. It is based on the possibility of overriding, rather than the presence of negation. It also differs from that defined in [14] as the immediate consequence operator is a mapping between typed interpretations rather than interpretations.

In a simple program with only functional methods, there is only one ground instance of a clause in which a value is assigned to functional method m of arity n for type τ so the immediate consequence operator T may be defined to include an atom A in $T(I)$ only if A does not conflict with an atom in I. But in a simple program with set-valued methods, there may be many ground instances of clauses in which a value is assigned to set-valued method m of arity n for type τ, so A may be included in $T(I)$ even if A does conflict with an atom in I *of the same type*.

Example 4.1 Assume appropriate declarations D including those in Example 2.1. Consider program P wrt D:

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

Let $I = \{f1[treatment \rightarrow \text{"tell supervisor"}], f1[severity \twoheadrightarrow high]\}$. Intuitively, we would expect $f1[severity \twoheadrightarrow low] \in T(I)$ because it is not overridden by another clause in $[P]$ wrt I. Using the immediate consequence operator defined in [14], $f1[severity \twoheadrightarrow low]$ conflicts with $f1[severity \twoheadrightarrow high]$, so $f1[severity \twoheadrightarrow low] \notin T(I)$. $\square$

Obviously we require some way of remembering the type on which a set-valued method is defined.

A *typed interpretation* (or t-interpretation) is an extension to an interpretation where each atom is replaced by a pair (A, τ), where A is a ground atom and τ is a type attached to A. The set of all typed interpretations forma complete lattice. The immediate consequence operator is a mapping from typed interpretations to typed interpretations.

Definition The *immediate consequence operator* is defined as follows: Let P be a set of ground (clause, type) pairs, I a t-interpretation and τ a type. Then $T_P(I) = \{\,(A,\tau) \mid (C,\tau)$ is a (clause, type) pair of P, $C = A \leftarrow B_1 \wedge \cdots \wedge B_k$, for each B_i and type τ_i, $(B_i, \tau_i) \in I$, and, if A is a method atom and A' is another method atom with $(A', \tau') \in I$, and $\tau \neq \tau'$, then A and A' do not conflict $\}$.

Definition The *powers of* T_P are then defined:

$T_P \uparrow 0(I) = I$
$T_P \uparrow (n+1)(I) = T_P(T_P \uparrow n(I))$
$T_P \uparrow \omega(I) = \bigcup_{n=0}^{\infty} T_P \uparrow n(I)$.

We described a partial ordering on (clause, type) pairs of a program in Section 3. The following example illustrates why it is necessary to also take typing into account when partitioning the program.

Example 4.2 Assume appropriate declarations including those in Example 2.1. Consider the following program wrt D:

$\{x : sf\} \vdash x[severity \twoheadrightarrow high]$
$\{x : csf\} \vdash x[severity \twoheadrightarrow low]$

Under the partitioning described in [14], the ground instances of these two clauses could be in the same stratum. A possible model for this program would then be $\{f1[severity \twoheadrightarrow high]\}$. Using the ordering described in Section 3, the model of this program is $\{f1[severity \twoheadrightarrow low]\}$ as expected. $\square$

Then M_P^* is computed using the immediate consequence operator to find a fixpoint of P_1, $P_1 \cup P_2$ and so on.

$$
\begin{aligned}
M_1' &= T_{P_1}\!\uparrow\omega(\emptyset),\\
M_i' &= T_{P_i}\!\uparrow\omega(M_{i-1}'), \text{ for } 1 < i \leq n\\
M_n &= \{A \mid (A, t) \in M_n'\}\\
M_P^* &= M_n.
\end{aligned}
$$

Example 4.3 Assume appropriate declarations D including those in Example 2.1. Consider program P wrt D in Example 3.1 with the clause $f1[treatment \rightarrow \text{"turn off condenser"}]$. One of the possible i-stratifications of P gives the following partitioning on ground (clause, type) pairs:

$$
\begin{aligned}
P_1 = \{&(f1[treatment \rightarrow \text{"turn off condenser"}], f1), (f1[severity \twoheadrightarrow low] \leftarrow\\
&f1[treatment \rightarrow \text{"turn off condenser"}], csf)\},\\
P_2 = \{&(f1[severity \twoheadrightarrow low] \leftarrow f1[treatment \rightarrow \text{"tell supervisor"}], sf),\\
&(f1[severity \twoheadrightarrow high], sf)\}.
\end{aligned}
$$

Then M_P^* is computed as follows:

$$
\begin{aligned}
&T_{P_1}\!\uparrow 0(\emptyset) = \emptyset\\
&T_{P_1}\!\uparrow 1(\emptyset) = \{(f1[treatment \rightarrow \text{"turn off condenser"}], f1)\}\\
&T_{P_1}\!\uparrow 2(\emptyset) = \{(f1[treatment \rightarrow \text{"turn off condenser"}], f1), (f1[severity \twoheadrightarrow low], csf)\}\\
&M_1' = \{(f1[treatment \rightarrow \text{"turn off condenser"}], f1), (f1[severity \twoheadrightarrow low], csf)\}\\
&T_{P_2}\!\uparrow 0(M_1') = \{(f1[treatment \rightarrow \text{"turn off condenser"}], f1),\\
&\qquad (f1[severity \twoheadrightarrow low], csf)\}\\
&M_2' = \{(f1[treatment \rightarrow \text{"turn off condenser"}], f1), (f1[severity \twoheadrightarrow low], csf)\}\\
&M_P^* = \{f1[treatment \rightarrow \text{"turn off condenser"}], f1[severity \twoheadrightarrow low]\}
\end{aligned}
$$

□

Note that each M_j is a model for $\bigcup_{i \leq j} P_i$, and that M_P^* is a model of P, indeed the preferred model of P, in this example. In fact, these properties hold for all simple programs, and the following key result holds.

Theorem 4.1 *Let P be a simple program with respect to a set of declarations D. Then M_P^* does not depend on the particular i-stratification chosen for P, and M_P^* is equal to the preferred model M_P of P.*

Proof The construction of M_{k+1} in Theorem 3.1 is precisely the construction of the model M_P^*, so M_P^* is equal to M_p. In Theorem 3.2 we proved that this construction is independent of the chosen i-stratification. □

A problem with this procedure is the difficulty of assigning strata to the ground instances of clauses before starting the computation, and we are investigating means of avoiding this task. Another problem is that, like other naive bottom-up evaluation procedures, this one is not goal directed, indicating the need for optimizations perhaps based on program transformation and suggesting the investigation of goal-directed top-down procedures such as those considered in the next section.

5 Top-down evaluation

In this section we outline an alternative evaluation procedure which might be more efficient in practice. This top-down procedure is based on typed unification and a variant of SLD-resolution. The resolution procedure differs from SLD-resolution as ground computed answers for a selected subgoal are computed, filtered to eliminate overridden answers, and then substituted in the rest of the goal. The answers derived by this procedure are ground.

Definition Let S be a finite set of simple expressions, where a simple expression is a term, a method atom, or a predicate atom. A substitution θ is called a *unifier* for S if $S\theta$ is a singleton. A unifier θ for S is called the *most general unifier* (mgu) for S if, for each unifier σ of S, there exists a substitution γ such that $\sigma = \theta\gamma$.

Definition Let S be a finite set of simple expressions $\{\Gamma_1 \vdash E_1, \ldots, \Gamma_n \vdash E_n\}$ where the variables in each E_i are disjoint. The *disagreement set*, $D_k = \Gamma \vdash D'_k$, of S is defined as follows. Locate the leftmost symbol position at which not all E_i, $1 \leq i \leq n$, have the same symbol and extract from each expression in S the symbol at that position. Let D'_k be the set of all such symbols, and Γ be the variable typing of the variables in D'_k.

The unification algorithm takes a finite set of simple expressions S as input, and outputs the most general unifier if the set is unifiable, otherwise it reports that the set is not unifiable. It is defined as follows:

1. Put $k = 0$, $\sigma_0 = \varepsilon$.

2. If $S\sigma_k$ is a singleton then stop and σ_k is the mgu of S; otherwise find the disagreement set $D_k = \Gamma \vdash D'_k$ of $S\sigma_k$.

3. Let v and t be in D'_k. If v is a variable and the type of t is less than or equal to the type of v then put $\sigma_{k+1} = \sigma_k(\Gamma \vdash \{v/t\})$, else if v and t are both variables where $\Gamma \vdash v : \tau$ and $\Gamma \vdash t : \tau'$ are terms and there exists a greatest lower bound (glb) of (τ, τ') then $\Gamma' = \Gamma - \{t : \tau'\} \cup \{t : glb(\tau, \tau')\}$ and $\sigma_{k+1} = \sigma_k(\Gamma' \vdash \{v/t\})$, increment k and go to 2; otherwise stop, S is not unifiable σ_k is $\emptyset$.

Note that because there are no function symbols in our language, no occurs check is necessary.

Example 5.1 Assume the declarations given in Example 2.1. Let S be a set of expressions $\{\{x{:}cf, z{:}string\} \vdash x[problem \rightarrow z], \{x'{:}fsf\} \vdash x'[problem \rightarrow$ "cooling system flow high"$]\}$ to be unified. Then the disagreement set D_0 is $\{x : cf, x' : fsf\} \vdash \{x, x'\}$, and the unifier σ_1 is $\{x : cf, x' : tf\} \vdash \{x/x'\}$. The type of x' is a common subtype of the types of x and x'. The next disagreement set D_1 is $\{z{:}string\} \vdash \{z,$ "cooling system flow high"$\}$, and the next (and most general) unifier σ_2 is $\{x : cf, x' : tf, z : string\} \vdash \{x/x', z/$"cooling system flow high"$\}$. Thus, $S\sigma_2 = \{\{x' : tf\} \vdash x'[problem \rightarrow$ "cooling system flow high"$]\}$. □

We can now introduce our top-down evaluation procedure. It is similar to SLDNF-resolution except that our procedure must deal with answers which are only possibly true for method atoms. That is, answers which could be overridden. For a selected atom, a set of possible answers is computed, and a minimisation operation is applied to find which of the possible answers has not been overridden.

We define the "standard form" of an atom where the value of a method atom is a variable. If an atom in a query is $\leftarrow t[m \rightarrow b]$, where b is an object symbol, this atom may unify with the head of a clause that is overridden by another clause and hence return an incorrect answer.

To ensure we compute the expected answer, the value of the atom in the query must be a variable. We illustrate this problem and a solution in the following example.

Example 5.2 Assume appropriate declarations D including those in Example 2.1. Consider program P' wrt D which comprises program P in Example 3.1 and the clause $f1[treatment \rightarrow$ "turn off condenser"]. Consider also the goal $G = \leftarrow f1[severity \twoheadrightarrow high]$. G unifies with $\{x : sf\} \vdash x[severity \twoheadrightarrow high]$ so G appears to be true when in fact it should have been overridden. Rewriting G as $\{y : degree\} \leftarrow f1[severity \twoheadrightarrow y] \wedge y = high$ produces the expected results. $\square$

A method atom is in standard form if it has the form $\Gamma \vdash t[m@t_1, \ldots, t_n \rightarrow x]$ (resp. $\Gamma \vdash t[m@t_1, \ldots, t_n \twoheadrightarrow x]$), where $\Gamma \vdash t$, $\Gamma \vdash t_1, \ldots \Gamma \vdash t_n$ are terms, and x is a variable. Every predicate atom is in standard form. If an atom is in standard form we call it a standard atom.

An atom not in standard form can easily be rewritten as the conjunction of two standard atoms. Let A be a non-standard atom $\Gamma \vdash t[m@t_1, \ldots, t_n \rightarrow a]$ (resp. $\Gamma \vdash t[m@t_1, \ldots, t_n \twoheadrightarrow a]$), where a is an object. Let x be a variable, Γ' be $\Gamma \cup \{x : \tau\}$ where a is of type τ, and $=$ the equality predicate. Then $\Gamma' \vdash t[m@t_1, \ldots, t_n \rightarrow x] \wedge x = a$ (resp. $\Gamma \vdash t[m@t_1, \ldots, t_n \twoheadrightarrow x] \wedge x = a$) is the standard form of A.

We assume that R is a computation rule that selects atoms other than the equality atoms initially, then ground equality atoms. The ground equality atoms act as filters rather than answer generators.

We now describe the top-down evaluation procedure. Assume initially that there is a way to find the computed answers for $P \cup \{A\}$ where A is an atom. (We will define a way later.) We call these answers *atom answers*. We start by defining a goal tree.

Definition Let P be a program with respect to a set of declarations D and $G_i = \Gamma \vdash \leftarrow A_1 \wedge \cdots \wedge A_m \wedge \cdots \wedge A_k$ be a goal. Then G_{i+1} is *derived* from G_i using substitution θ_{i+1} if the following conditions hold

- $\Gamma \vdash A_m$ is an atom in G_i called the selected atom.
- θ_{i+1} is an atom answer for $P \cup \{\Gamma \vdash \leftarrow A_m\}$.
- G_{i+1} is the goal $(\Gamma \vdash \leftarrow A_1 \wedge \cdots \wedge A_{m-1} \wedge A_{m+1} \wedge \cdots \wedge A_k)\theta_{i+1}$.

Definition Let P be a program with respect to a set of declarations D and G a goal. A *goal derivation* of $P \cup \{G\}$ consists of a finite sequence $G_0 = G, G_1, \ldots, G_n$ of goals, a sequence $\theta_1, \ldots, \theta_n$ of substitutions such that for each i, G_{i+1} is derived from G_i with substitution θ_{i+1}. Either

- the last goal is empty, or
- the last goal is $\Gamma \vdash \leftarrow A_1 \wedge \cdots \wedge A_m \wedge \cdots \wedge A_k$, $\Gamma \vdash A_m$ is the selected atom and there is no atom answer for $P \cup \{\Gamma \vdash \leftarrow A_m\}$.

Definition Let P be a program with respect to a set of declarations D and G be a goal. A *goal refutation* of $P \cup \{G\}$ consists of a sequence $G_0 = G, G_1, \ldots, G_n = \square$ of goals, a sequence $\theta_1, \ldots, \theta_n$ of substitutions such that for each i, G_{i+1} is derived from G_i with substitution θ_{i+1}.

A goal refutation of $P \cup \{G\}$ via R is a goal refutation of $P \cup \{G\}$ in which a computation rule R is used to select atoms.

Definition Let P be a program with respect to a set of declarations D and G a goal. A *goal tree* for $P \cup \{G\}$ is defined as follows:

- The root node is G.
- Each node in the tree is a (possibly empty) goal.
- Let $\Gamma \vdash \leftarrow A_1 \wedge \cdots \wedge A_m \wedge \cdots \wedge A_k$ be a non-leaf node in the tree and suppose that $\Gamma \vdash A_m$ is selected. For each atom answer θ for $P \cup \{\Gamma \vdash \leftarrow A_m\}$, this node has a child which is $(\Gamma \vdash \leftarrow A_1 \wedge \cdots \wedge A_{m-1} \wedge A_{m+1} \wedge \cdots \wedge A_k)\theta$.
- Let $\Gamma \vdash \leftarrow A_1 \wedge \cdots \wedge A_m \wedge \cdots \wedge A_k$ be a leaf node in the tree and suppose that $\Gamma \vdash A_m$ is selected. There is no atom answer for $P \cup \{\Gamma \vdash \leftarrow A_m\}$.

Definition Let P be a program with respect to a set of declarations D and G a goal. A *computed answer* for $P \cup \{G\}$ is the substitution obtained by restricting the composition $\theta_1 \ldots \theta_n$ to variables of G, where $\theta_1, \ldots, \theta_n$ is a sequence of substitutions used in a goal refutation of $P \cup \{G\}$ via R.

We have defined a goal tree and the computed answer for $P \cup \{G\}$, assuming there is a definition for "atom answer". We now define "atom answer".

Definition Let P be a program with respect to a set of declarations D and $G = \Gamma \vdash \leftarrow A_m$ be a standard atom. Suppose $C = \Gamma \vdash A \leftarrow A_1 \wedge \cdots \wedge A_k$ is a clause of P where the variables of G and C are standardized apart. An *atom derivation* of $P \cup \{G\}$ consists of goals G, G', type τ, substitution θ and clause C where

- θ is the mgu of $\Gamma \vdash A_m$ and $\Gamma \vdash A$,
- G' is the goal $(\Gamma \vdash \leftarrow A_1 \wedge \cdots \wedge A_k)\theta$, and
- τ is the type of $\Gamma \vdash A$.

To determine the atom answers, it is necessary to compare possible answers. We can only compare ground instances of answers. We now define ground instances of (substitution, type) pairs, possible answers and atom answers.

Definition Let P be a program with respect to a set of declarations D. Suppose θ is $\Gamma \vdash \theta'$ and τ is a type. The *ground instances* of (θ, τ) are $\{(\sigma, \tau) \mid \sigma \in [\Gamma \vdash \theta']\}$, where $[\Gamma \vdash \theta']$ is the set of all ground instances of θ' consistent with Γ and P with respect to D. We write $[\theta, \tau]$ for the set of ground instances of (θ, τ).

Example 5.3 Consider the program P' wrt D in Example 5.2 and the (substitution, type) pair $S = (\{x : sf, x' : tf, y : degree\} \vdash \{x/x', y/high\}, sf)$. The ground instances of S are $(\{x : sf, x' : tf, y : degree\} \vdash \{x/f1, x'/f1, y/high\}, sf)$. $\Box$

Definition Let P be a program with respect to a set of declarations D and $\Gamma \vdash A$ a standard atom. There is an atom derivation for each clause in P whose head unifies with $\Gamma \vdash A$. The goals derived are G_i, with type τ_i, and substitutions θ_i. For goal G_i, suppose $\Theta'_i = \{\theta'_{i1}, \ldots, \theta'_{in_i}\}$ are the computed answers for $P \cup \{G_i\}$. Then the substitutions $\theta_{i1} = \theta_i\theta'_{i1}, \ldots, \theta_{in} = \theta_i\theta'_{in_i}$. The set of *possible answers* for $P \cup \{\Gamma \vdash \leftarrow A\}$ is the set of (substitution, type) pairs $\{[\theta_{i1}, \tau_i], \ldots, [\theta_{in_i}, \tau_i]\}$ restricted to the variables of $\Gamma \vdash A$.

Definition Let P be a program with respect to a set of declarations D, $\Gamma \vdash A$ a standard atom, and Ψ a set of possible answers for $P \cup \{\Gamma \vdash \leftarrow A\}$. Let $A = t[m@t_1, \ldots, t_n \rightarrow t']$ (resp. $A = t[m@t_1, \ldots, t_n \twoheadrightarrow t']$). A possible answer $\psi_1 = (\theta_1, \tau_1) \in \Psi$ is *smaller* than another possible answer $\psi_2 = (\theta_2, \tau_2) \in \Psi$ if $A\theta_1$ conflicts with $A\theta_2$, and $\tau_1 < \tau_2$ with respect to method m. In this case we write $\psi_1 < \psi_2$. We say that ψ_k is *minimal* with respect to Ψ if there is no $\psi_j \in \Psi$ such that $\psi_j < \psi_k$. Let $\Gamma \vdash A$ be a predicate atom. Every possible answer for $P \cup \{\Gamma \vdash \leftarrow A\}$ is minimal.

Definition Let P be a program with respect to a set of declarations D.

- Suppose $\Gamma \vdash A$ is an atom in standard form. Then the *atom answers* for $P \cup \{\Gamma \vdash \leftarrow A\}$ are the minimal possible answers for $P \cup \{\Gamma \vdash \leftarrow A\}$.
- Otherwise suppose $\Gamma \vdash A$ is not in standard form. Let $\Gamma' \vdash A_1 \wedge A_2$ be the standard form of $\Gamma \vdash A$. Then the *atom answers* for $P \cup \{\Gamma \vdash \leftarrow A\}$ are the computed answers for $P \cup \{\Gamma' \vdash \leftarrow A_1 \wedge A_2\}$ restricted to the variables of $\Gamma \vdash A$.

Example 5.4 Consider program P' wrt D in Example 5.2, the goal $G = \{x'{:}tf, y{:}degree\} \vdash \leftarrow x'[severity \twoheadrightarrow y] \wedge x'[treatment \rightarrow \text{"turn off condenser"}]$, and the goal trees in Figure 1.

Let $A = \{x'{:}tf, y{:}degree\} \vdash x'[severity \twoheadrightarrow y]$ be the selected atom. Atom A unifies with $\{x : sf\} \vdash x[severity \twoheadrightarrow high]$ with $\theta = \{x : sf, x' : tf, y : degree\} \vdash \{x/x', y/high\}$. Atom A unifies with the head of $\{x : sf\} \vdash x[severity \twoheadrightarrow low] \leftarrow x[treatment \rightarrow \text{"tell supervisor"}]$ with $\theta = \{x : sf, x' : tf, y : degree\} \vdash \{x/x', y/low\}$. Atom A unifies with the head of $\{x : csf\} \vdash x[severity \twoheadrightarrow low] \leftarrow x[treatment \rightarrow \text{"turn off condenser"}]$ with $\theta = \{x : csf, x' : tf, y : degree\} \vdash \{x/x', y/low\}$.

The possible computed answers for $P' \cup \{\{x'{:}tf, y{:}degree\} \vdash \leftarrow x'[severity \twoheadrightarrow y]\}$ are $\Theta = \{(\{x'{:}tf, y{:}degree\} \vdash \{x'/f1, y/high\}, sf), (\{x'{:}tf, y{:}degree\} \vdash \{x'/f1, y/low\}, csf)\}$. The computed answers are the minimal possible computed answers, $\Theta_1 = \{\{x' : tf, y : degree\} \vdash \{x'/f1, y/low\}\}$.

The atom answer for $P' \cup \{f1[treatment \rightarrow \text{"turn off condenser"}]\}$ is $\theta = \emptyset$, where θ is the computed answer for $P' \cup \{\{y{:}string\} \vdash f1[treatment \rightarrow y] \wedge y = \text{"turn off condenser"}\}$.

Thus the computed answer for $P' \cup \{G\}$ is $\{\{x' : tf, y : degree\} \vdash \{x'/f1, y/low\}\}$. $\Box$

In the proof of following theorem we use induction on the rank of a goal tree. We first define the rank of a goal tree.

Definition Let P be a program with respect to a set of declarations D and G a goal. The *rank* of a goal tree for $P \cup \{G\}$ is 0 if the root node is $G = \Box$.

Definition Let P be a program with respect to a set of declarations D and G a goal. The *rank* of a goal tree for $P \cup \{G\}$ is $j + 1$ if:

- The root node is G.
- For every non-leaf node $\Gamma_i \vdash \leftarrow A_{i1} \wedge \cdots \wedge A_{ik_i}$, for every atom $\Gamma_i \vdash A_{in}$ where $1 \leq n \leq k_i$, either
  - $\Gamma_i \vdash A_{in}$ is not in standard form. Let the standard form of $\Gamma_i \vdash A_{in}$ be $\Gamma' \vdash A'_1 \wedge A'_2$. Then the rank of the goal tree for $P \cup \{\Gamma' \vdash \leftarrow A'_1 \wedge A'_2\}$ is j_{in}. Or
  - $\Gamma_i \vdash A_{in}$ is in standard form and the maximum rank of the resulting goal trees for clauses with which $\Gamma_i \vdash A_{in}$ unifies is j_{in}.

  Then j is the maximum j_{in}.

Example 5.5 Consider the goal trees in Figure 1. The goal trees for $P' \cup \{\{y{:}string\} \vdash \leftarrow f1[treatment \rightarrow y] \wedge y = \text{"turn off condenser"}\}$ and $P' \cup \{\{x'{:}tf, y{:}string\} \vdash \leftarrow x'[treatment \rightarrow y] \wedge y = \text{"turn off condenser"}\}$ have rank 1. The goal tree for $P' \cup \{\{x'{:}tf\} \vdash \leftarrow x'[treatment \rightarrow \text{"turn off condenser"}]\}$ has rank 2. The goal tree for $P' \cup \{\{x'{:}tf, y{:}degree\} \vdash \leftarrow x'[severity \twoheadrightarrow y] \wedge x'[treatment \rightarrow \text{"turn off condenser"}]\}$ has rank 3. $\Box$

The following soundness result holds.

$$\{x' : tf, y : degree\} \vdash \;\leftarrow x'[severity \twoheadrightarrow y] \wedge x'[treatment \rightarrow \text{“turn off condenser”}]$$

$$\Big|\; x'/f1,\; y/low$$

$$\leftarrow f1[treatment \rightarrow \text{“turn off condenser”}]$$

$$\Big|$$

$$\Box$$

$$\{x' : tf\} \vdash \;\leftarrow x'[treatment \rightarrow \text{“tell supervisor”}]$$

$$\{x' : tf\} \vdash \;\leftarrow x'[treatment \rightarrow \text{“turn off condenser”}]$$

$$\Big|\; x'/f1$$

$$\Box$$

$$\{x' : tf, y : string\} \vdash \;\leftarrow x'[treatment \rightarrow y] \wedge y = \text{“turn off condenser”}$$

$$\Big|\; x'/f1,\; y/\text{“turn off condenser”}$$

$$\leftarrow \text{“turn off condenser”} = \text{“turn off condenser”}$$

$$\Big|$$

$$\Box$$

$$\{y : string\} \vdash \;\leftarrow f1[treatment \rightarrow y] \wedge y = \text{“turn off condenser”}$$

$$\Big|\; y/\text{“turn off condenser”}$$

$$\leftarrow \text{“turn off condenser”} = \text{“turn off condenser”}$$

$$\Big|$$

$$\Box$$

Figure 1: Goal trees for $P' \cup \{G\}$

Lemma 5.1 *Let P be a simple program and G a goal. Let the rank of the goal tree for $P \cup \{G\}$ be 1. Every computed answer for $P \cup \{G\}$ is a correct answer for $P \cup \{G\}$.*

Proof Let G be the goal $\Gamma \vdash \leftarrow A_1 \wedge \cdots \wedge A_n$ and $\theta_1, \ldots, \theta_n$ be the sequence of atom answers used in a goal refutation of $P \cup \{G\}$ via R. Let M_P be the preferred model of P wrt D. We have to show that $(\Gamma \vdash \leftarrow A_1 \wedge \cdots \wedge A_n)\theta_1 \ldots \theta_n$ is true in M_P.

The rank of $P \cup \{G\}$ is 1. Thus the atoms in G are in standard form and the restriction of P contains only facts.

We use induction on the length of the goal refutation of $P \cup \{G\}$ via R. Suppose the length is 1. Then G is the standard atom $\Gamma \vdash \leftarrow A_1$. There is a clause $\Gamma \vdash A$ in P such that $(\Gamma \vdash A)\theta_1 = (\Gamma \vdash A_1)\theta_1$, where $(\Gamma \vdash A)\theta_1$ is not overridden by an instance of another clause in P wrt D. It follows that $(\Gamma \vdash A_1)\theta_1$ is true in M_P.

Assume the result holds where the length of the goal refutation of $P \cup \{G\}$ via R is $n-1$.

Suppose the length of the goal refutation of $P \cup \{G\}$ via R is n. Let $\Gamma \vdash A_m$ be the selected atom. There is an instance of a clause $\Gamma \vdash A$ in P with which $\Gamma \vdash A_m$ unifies which is not overridden by an instance of another clause in P wrt D. By the induction hypothesis $(\Gamma \vdash A_1 \wedge \cdots \wedge A_{m-1} \wedge A_{m+1} \wedge \cdots \wedge A_n)\theta_1 \ldots \theta_n$ is true in M_P. $(\Gamma \vdash A_m)\theta_1 \ldots \theta_n = (\Gamma \vdash A)\theta_1 \ldots \theta_n$ is true in M_P. Thus $(\Gamma \vdash A_1 \wedge \cdots \wedge A_n)\theta_1 \ldots \theta_n$ is true in M_P. $\Box$

Theorem 5.1 *Let P be a simple program with respect to a set of declarations D, and G a goal. Every computed answer for $P \cup \{G\}$ is a correct answer for $P \cup \{G\}$.*

Proof Let G be the goal $\Gamma \vdash \leftarrow A_1 \wedge \cdots \wedge A_n$ and $\theta_1, \ldots, \theta_n$ be the sequence of atom answers used in a goal refutation of $P \cup \{G\}$ via R. Let M_P be the preferred model of P wrt D. We have to show that $(\Gamma \vdash \leftarrow A_1 \wedge \cdots \wedge A_n)\theta_1 \ldots \theta_n$ is true in M_P.

We use induction on the rank of the goal tree for $P \cup \{G\}$. Suppose the rank is 1.

By lemma 5.1 $(\Gamma \vdash A_1 \wedge \cdots \wedge A_n)\theta_1 \ldots \theta_n$ is true in M_P.

Assume the result holds for goal trees for $P \cup \{G\}$ of rank $\leq q-1$.

Suppose the rank of the goal tree for $P \cup \{G\}$ is q. Again we use induction on the length of the refutation of $P \cup \{G\}$ via R. Suppose the length is 1. Then G is the atom $\Gamma \vdash \leftarrow A_1$. Either

- $\Gamma \vdash A_1$ is a standard atom. Then there is a ground instance of a clause $\Gamma \vdash A \leftarrow A'_1 \wedge \cdots \wedge A'_k$ in P such that $(\Gamma \vdash A_1)\theta' = (\Gamma \vdash A)\theta'$ which is not overridden by an instance of another clause in P wrt D. Then either
  - the rank of the tree for $P \cup \{(\Gamma \vdash \leftarrow A'_1 \wedge \cdots \wedge A'_k)\theta'\}$ is $< q-1$. Then by the main induction hypothesis $(\Gamma \vdash A_1)\theta_1$ is true in M_P. Or
  - the rank of the tree for $P \cup \{(\Gamma \vdash \leftarrow A'_1 \wedge \cdots \wedge A'_k)\theta'\}$ is $q-1$. Then there is a refutation of $P \cup \{(\Gamma \vdash \leftarrow A'_1 \wedge \cdots \wedge A'_k)\theta'\}$ with atom answers $\theta'_1, \ldots, \theta'_j$. Let $\theta_1 = \theta'\theta'_1 \ldots \theta'_j$. By the main induction hypothesis $(\Gamma \vdash \leftarrow A'_1 \wedge \cdots \wedge A'_k)\theta_1$ is true in M_P thus $(\Gamma \vdash A_1)\theta_1$ is true in M_P.
- $\Gamma \vdash A_1$ is not in standard form. Let $\Gamma' \vdash A'_1 \wedge A'_2$ be the standard form of $\Gamma \vdash A_1$. There is a goal tree for $P \cup \{\Gamma' \vdash \leftarrow A'_1 \wedge A'_2\}$ of rank $\leq q-1$. There is a refutation of $P \cup \{\Gamma' \vdash \leftarrow A'_1 \wedge A'_2\}$ via R with computed answer θ_1. By the induction hypothesis, $(\Gamma' \vdash A'_1 \wedge A'_2)\theta_1$ is true in M_P, thus $(\Gamma \vdash A_1)\theta_1$ is true in M_P.

Assume the result holds for refutations of $P \cup \{G\}$ via R of length $n-1$.

Suppose the length of the refutation of $P \cup \{G\}$ via R is n. Then G is $\Gamma \vdash \leftarrow A_1 \wedge \cdots \wedge A_n$. Let $\Gamma \vdash A_m$ be the selected atom. Either

- $\Gamma \vdash A_m$ is a standard atom. Then there is a ground instance of a clause $\Gamma \vdash A \leftarrow A'_1 \wedge \cdots \wedge A'_k$ in P such that $(\Gamma \vdash A_m)\theta' = (\Gamma \vdash A)\theta'$ which is not overridden by an instance of another clause in P wrt D. Then either
  - the rank of the tree for $P \cup \{(\Gamma \vdash \leftarrow A'_1 \wedge \cdots \wedge A'_k)\theta'\}$ is $< q-1$. Then by the main induction hypothesis $(\Gamma \vdash A_m)\theta_1 \ldots \theta_n$ is true in M_P. Or
  - the rank of the tree for $P \cup \{(\Gamma \vdash \leftarrow A'_1 \wedge \cdots \wedge A'_k)\theta'\}$ is $q-1$. Then there is a refutation of $P \cup \{(\Gamma \vdash \leftarrow A'_1 \wedge \cdots \wedge A'_k)\theta'\}$ with atom answers $\theta'_1, \ldots, \theta'_j$. Let $\theta = \theta'\theta'_1 \ldots \theta'_j$. By the main induction hypothesis $(\Gamma \vdash \leftarrow A'_1 \wedge \cdots \wedge A'_k)\theta_1 \ldots \theta_n$ is true in M_P thus $(\Gamma \vdash A_m)\theta_1 \ldots \theta_n$ is true in M_P.
- $\Gamma \vdash A_m$ is not in standard form. Let $\Gamma' \vdash A_1 \wedge A_2$ be the standard form of $\Gamma \vdash A_m$. The rank of the tree for $P \cup \{\Gamma' \vdash \leftarrow A_1 \wedge A_2\}$ is $\leq q-1$. By the main induction hypothesis $(\Gamma \vdash A_m)\theta_1 \ldots \theta_n$ is true in M_P.

By the inner induction hypothesis $(\Gamma \vdash \leftarrow A_1 \wedge \cdots \wedge A_{m-1} \wedge A_{m+1} \wedge \cdots \wedge A_n)\theta_1 \ldots \theta_n$ is true in M_P thus $(\Gamma \vdash \leftarrow A_1 \wedge \cdots \wedge A_n)\theta_1 \ldots \theta_n$ is true in M_P. □

6 Other approaches to multiple inheritance

In the introduction, we described three approaches for dealing with ambiguities due to multiple inheritance. In the previous sections, we adopted the most restrictive approach because it complicated the semantics least. In this section we investigate the other two approaches.

The first approach allows the definition of a static ordering on types with respect to methods in the schema declaration to determine from which subclass to inherit. Obviously such an ordering applies only to the method it is specified on. If no such ordering is given for a method then the partial ordering given by the type declarations applies. Where ambiguities due to multiple inheritance could arise a static ordering must be specified.

Example 6.1 Consider the following declarations where a static ordering is not specified, $fsf < csf$, $cf < csf$, $csf < sf$ and $sf[problem \Rightarrow string]$. Then $fsf < cf < sf$ and $csf < cf < sf$, and there is a possibility of ambiguity.

Now consider the following declarations $tf < fsf$, $tf < cf$, $fsf < csf$, $cf < csf$, $csf < sf$ and $sf[problem \Rightarrow string]$ *where* $tf < fsf < csf < cf < sf$. The static ordering is specified in the *where* part of the method declaration. Now $tf < fsf < csf < cf < sf$ wrt functional method *problem*.
□

In order to incorporate static ordering in Gulog, the only changes which are necessary are to the definitions which include type ordering (like possible overriding). The ordering on types in such definitions must now include the static ordering on types with respect to methods. We believe that the obvious generalizations to the theorems still hold.

The second approach, dynamic ordering, adds flexibility to the language, solving not only the problem caused by multiple inheritance but also providing a facility for monotonic inheritance. A dynamic ordering is specified in the program. A role atom is used to indicate which class a method definition is inherited from.

Example 6.2 Consider the following declarations $tf < fsf$, $tf < cf$, $fsf < csf$, $cf < csf$, $csf < sf$ and $sf[problem \Rightarrow string]$ and the program:

$\{x : sf\} \vdash x[problem \rightarrow$ "system failure"$]$
$\{x : csf\} \vdash x[problem \rightarrow$ "cooling system failure"$]$
$\{x : tf, y : string\} \vdash x[problem \rightarrow y] \leftarrow x : csf[problem \rightarrow y]$.

In this program, $x : csf[problem \rightarrow y]$ is a role atom. The clause $\{x : tf, y : string\} \vdash x[problem \rightarrow y] \leftarrow x : csf[problem \rightarrow y]$ specifies that the rule for functional method $problem$ for objects of inherited type csf is to be applied to objects of inherited type tf. □

The definitions and procedures described in this paper change dramatically with the inclusion of role atoms in the language. This approach is still under investigation. Early results show that such a language is a generalization of Gulog, and there is a query evaluation procedure for a language with role atoms. The results are described in [13].

7 Discussion

We have defined a simple language which incorporates many common concepts from other deductive object-oriented systems, to present a relatively simple and natural declarative semantics for the language, to describe corresponding evaluation procedures, and hence to gain an understanding of the main features of deductive object-oriented systems. To simplify the presentation, negation was not included. As shown in [13], while negation does complicate the semantics and evaluation procedures, it does not alter the relationship between deduction, inheritance and overriding.

The object-oriented features which we concentrated on were inheritance, including multiple inheritance, overriding, functional and set-valued methods. Many of the logics which have been described in the literature do not address overriding [3, 7, 10, 18, 23] or multiple inheritance [1, 7, 10, 18, 22, 23]. One of the logics in which both overriding and multiple inheritance are considered is F-logic [17]. F-logic is a lot richer than Gulog. In particular, in F-logic, it is possible to query the schema. However the semantics of overriding in F-logic is complex and overriding is not addressed in the proof theory. *L&O* [24] is intended as a practical programming language. The semantics of *L&O* are given by a translation to Prolog. Similarly, in [4] the authors study the features of deductive object-oriented languages using translations from their languages to datalog with negation.

To allow us to concentrate on the semantics of multiple inheritance, overriding, deduction, and functional and set-valued methods, we have separated the schema from the data and included strong typing. The separation of the schema from the data follows the same approach in relational databases. Strong typing has been introduced in logic in [16] and in the object-oriented setting in [9]. We believe this provides a well-defined and well understood context in which to study the main features of deductive object-oriented logics.

In the body of this paper, we addressed the most restrictive approach to conflicts due to multiple inheritance because it simplified the presentation. The effects of the other approaches are briefly discussed in Section 6.

We described a declarative semantics, and bottom-up and top-down query evaluation procedures for Gulog. We have shown that the bottom-up and top-down procedures described in Sections 4 and 5 are sound with respect to the declarative semantics for simple programs. We believe that the bottom-up procedure is complete and the top-down procedure would be complete if some tabling mechanism [12] was added to eliminate infinite branches. When defining the declarative semantics and bottom-up evaluation procedures directly, some other approaches treat the rules of each class as a separate module, find an interpretation for that module and then combine them to give an interpretation for the program [8]. In our approach, we treat the program as a whole, which coincides more closely with the logic programming approach, and recognise overriding in the definitions for stratification and model. The approach described in this paper provides a more direct explanation of the semantics of inheritance and overriding than that taken by Abiteboul et al. in [4] and us in [15], where the semantics of $datalog^{meth}$ and Gulog (respectively) were described by a translation into datalog.

Our bottom-up evaluation procedure suffers from the same inefficiencies as other naive bottom-up procedures and we believe that similar optimizations can be applied. Similarly the top-down procedure provides the basic intuition for such a procedure. Laenens and Vermier [20] adopted a similar approach except that the authors do not distinguish between classes and objects; they do not distinguish between functional and set-valued methods; and models are based on the more complicated 3-valued semantics. The advantage of our procedures is that they provide some understanding of the interaction of inheritance, overriding and deduction and there is no need to translate the programs and queries.

We have also shown that many object-oriented features can be captured in a first-order language which has many analogies with datalog. An advantage of these analogies is that work which has been carried out in the area of logic programming can be adapted to this language.

The results in this paper hold for "simple" programs. In [15], we consider a less restrictive class of programs and provide semantics and evaluation procedures for this class of programs via a translation to modularly stratified logic programs.

We believe the approach we have taken to defining the semantics of a deductive object-oriented language has been effective and in the future we will include other features such as other approaches to dealing with multiple inheritance, monotonic inheritance, negation and aggregates in the semantics.

Acknowledgements

We thank Michael Lawley, Alexandre Lefebvre and Christopher Higgins for useful discussions and suggestions. We are also grateful to the referees for their helpful comments.

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] K. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, 1988.

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

[8] S. Brass and U. Lipeck. Semantics of inheritance in logical object specifications. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Proceedings of the Second International Conference on Deductive and Object-Oriented Databases*, pages 411–430, Munich, Germany, 1991.

[9] L. Cardelli. A semantics of multiple inheritance. *Semantics of Data Types*, pages 51–67, 1984. published as Lecture Notes in Computer Science 173 by S pringer-Verlag.

[10] 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.

[11] 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.

[12] S. W. Dietrich. Extension Tables: Memo Relations in Logic Programming. In *Proc. 3rd Symposium on Logic Programming*, pages 264–272, San Francisco, California, 1987.

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

[14] 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.

[15] 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.

[16] P. M. Hill and R. W. Topor. A semantics for typed logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 1–62. MIT Press, Cambridge, Massachusetts, 1992.

[17] 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.

[18] M. Kifer and J. Wu. A logic for object-oriented logic programming (Maier's O-logic revisited). In *Proc. 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Philadelphia, PA, March, 1989.

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

[20] E. Laenens and D. Vermeir. Assumption-free semantics for ordered logic programs: On the relationship between well-founded and stable partial models. *Journal of Logic and Computation*, 1(2):159–185, 1992.

[21] M. J. Lawley. A Prolog interpreter for F-logic. Unpublished report, Griffith University, 1993.

[22] Y. Lou and Z. M. Ozsoyoglu. LLO: An object-oriented deductive language with methods and method inheritance. In *Proc. of the ACM SIGMOD International Conference on the Management of Data*, pages 198–207, 1991.

[23] D. Maier. A logic for objects. Technical Report CS/E-86-012, Oregon Graduate Center, Beaverton, OR, 1986.

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

[25] I. S. Mumick and K. A. Ross. SWORD: A declarative object-oriented database. In S. Ceri, K. Tanaka, and S. Tsur, editors, *Proceedings of the Third International Conference on Deductive and Object-Oriented Databases*, Phoenix, 1993.

[26] S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases*. Computer Science Press, New York, 1989.

[27] 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.

[28] D. S. Touretzky. *The Mathematics of Inheritance Systems*. Morgan Kaufmann, Los Altos, CA, 1986.