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

Department of Computer Science

PO Box 600
Wellington
New Zealand

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

Resolving Ambiguities caused by Multiple Inheritance

Gillian Dobbie and Rodney Topor

Technical Report CS-TR-96/14

Abstract

In object-oriented languages, multiple inheritance can cause ambiguities when methods of the same name are inherited from more than one superclass of a given class. In C++, qualifiers can be used to explicitly state which method should be inherited. We describe a mathematical foundation for an object-oriented language that uses qualifiers or *roles* to resolve such ambiguity. Our theory also allows us to model the role "super". For languages with dynamic overriding [1], it further allows us to model monotonic inheritance of multi-valued methods. Finally, we describe a possible implementation of query evaluation in our language. This work extends that presented in [5].

Publishing Information

This paper was presented at the Fourth International Conference on Deductive and Object-Oriented Databases, DOOD '95, Singapore, December 1995.

Author Information

Gillian Dobbie	Rodney Topor
Department of Computer Science	School of Computing and Information Technology
Victoria University of Wellington	Griffith University
PO Box 600	Nathan, Qld 4111
New Zealand	Australia
`gill@comp.vuw.ac.nz`	`rwt@cit.gu.edu.au`

1 Introduction

In object-oriented languages with multiple inheritance, a class may inherit methods from more than one superclass. For example, class *teaching assistant* might inherit methods directly from classes *student* and *employee*. One of the problems with multiple inheritance is that an ambiguity arises when a method is defined in more than one superclass. For example, suppose *Bob* is a *teaching assistant* who has *phone_number* 387 2651 as a *student* and *phone_number* 282 2493 as an *employee*, which definition should be inherited for *Bob* the *teaching assistant*? This issue is addressed in different ways in object-oriented programming languages. Eiffel forces derived classes to rename superclass methods that conflict [8]. Self prioritizes superclasses [13]. CLOS merges member "slots" (instance variables) with the same name into a single slot [9], as did the earlier Flavors. Smalltalk renders same names for instance variables of subclasses illegal [6]. C++ declares an error if a conflict arises, but a class qualifier can be used to explicitly disambiguate [12]. That is, using C++ in the above example it is possible to explicitly state that *Bob* the *teaching assistant* inherits the definition of *phone_number* from one of the superclasses, say *student*. In this paper, we follow the approach used in C++ and provide a syntax, semantics and an evaluation procedure for an object-oriented language that has roles that enable us to qualify methods, and a syntactic condition that enables the detection of ambiguities due to multiple inheritance should they arise.

As well as providing a means of resolving ambiguities due to multiple inheritance, roles allow the modeling of monotonic inheritance in a language that has dynamic overriding [1]. With dynamic overriding, if a method is defined in a class and redefined in its subclass then when the method is applied to an object in the subclass, the definition from the subclass is applied. For example, suppose method *legal_names* is defined in classes *person* and *female*, where *female* is a subclass of *person*. Then, the definition in *female* is applied to objects that belong to class *female*. It is possible that both definitions should be applied. That is, an object which belongs to class *female* returns the union of the results returned by methods *legal_names* in *female* and *legal_names* in *person*. This is called monotonic inheritance and can be modeled in a language with dynamic overriding and roles.

When modeling monotonic inheritance in this way, it is tedious and prone to error to have to explicitly state the name of the class that is inherited from when it is simply the superclass and there is no ambiguity. To alleviate this problem we introduce the word *super*. This is not a new concept, it is used in object-oriented languages like Smalltalk [6] and Objective-C [3]. Thus, in the above example where the class *female* monotonically inherits method *legal_names* from its superclass then it can be specified by generically qualifying the method *legal_names* with *super* rather than explicitly with *person*.

In this paper, we introduce a mathematical foundation for an object-oriented language with roles and discuss properties of the language. This work extends that presented in [5]. For space reasons, the proofs are omitted. They can be found in [4]. In Section 2, we outline the syntax of the language, and describe a syntactic condition for programs to be unambiguous with respect to multiple inheritance. We describe a class of programs called "simple programs" which are stratified and unambiguous with respect to multiple inheritance. In Section 3, we describe a natural semantics for programs in this language, based on preferred models [10]. Each simple program has exactly one preferred model, which corresponds to the natural meaning of the program. In Section 4, we describe an evaluation procedure for queries with respect to programs in this language. We show that computed answers are correct with respect to the semantics of Section 3. This evaluation involves translating roled programs and queries to Datalog (with negation) and evaluating their Datalog images. Conclusions and further work are outlined in Section 5.

2 Syntax

In this paper, in order to keep the definitions and semantics clear and simple, we consider only positive programs with multi-valued methods, and do not differentiate between types and classes. Adding negation has the usual consequences and the consequences of including functional methods (as well as multi-valued methods) are explained in [4].

In the language described in this paper, methods are defined by deductive rules as in the languages described in [1] and [7].

Throughout this paper we use the following notation. The letters τ and σ denote type symbols, a denotes an object symbol, p denotes a predicate symbol, m denotes a method symbol, x, y denote variable symbols, and s, t, u terms. Subscripts and superscripts are also used.

2.1 Programs and Goals

We first describe the syntax of declarations, which define a set of types arranged in a lattice, a set of typed object symbols, a set of method symbols each with a given signature, and a set of predicate symbols each with a given signature.

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', τ' is a supertype of τ. We also define $<_t$ to be the transitive closure of $<$. Then, if $\tau <_t \tau'$, we say τ is a subtype of τ' and τ' is a supertype of τ. Also, if $\tau <_t \tau'$ or $\tau = \tau'$, we write $\tau \leq_t \tau'$.

- 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'. When we refer to the type of an object we mean the most specific type of the object.

- 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 of arity n is $\tau_1 \times \cdots \times \tau_n$.

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

As in [2], the maximal element in the type lattice is $\top$ and the minimal element is $\perp$. In a set of declarations, there is exactly one declaration for each object symbol, n-ary predicate symbol, n-ary method symbol and the relation $<_t$ is a non-reflexive partial order on the type symbols. It is possible for one symbol to be used for a method and a predicate, or for methods of different arities, or for predicates of different arities.

Example 2.1 Consider the following set of declarations D:

$female < person$	$writer < person$
$female\_writer < female$	$female\_writer < writer$
$mary{:}female\_writer$	$person[legal\_names@date \Rrightarrow string]$
$female[maiden\_name \Rrightarrow string]$	$writer[pen\_name@date \Rrightarrow string]$
$person[last\_name@date \Rrightarrow string]$.	

Then, *female_writer* is a subtype of *female* which in turn is a subtype of *person*. Similarly, *female_writer* is a subtype of *writer* which is a subtype of *person*. The object *mary* is of inherited type $female\_writer$, and consequently $female$, $writer$ and $person$. The set of declarations D also gives signatures for 0-ary method $maiden\_name$ and 1-ary methods $legal\_names$, pen_name and $last\_name$. $\Box$

We now define data definitions. Data is defined by a program with respect to a set of declarations. The following definitions in this section are with respect to a set of declarations. So that we can construct non-ground terms and clauses, we need to assign types to the variables.

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

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 a if and only if a 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 obvious.

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

Definition Given a variable typing Γ, a *roled atom* is defined as follows:

- If p is an n-ary predicate symbol with signature $\tau_1 \times \cdots \times \tau_n$ and Γ contains the typings for the variables in t_i, and $\Gamma \vdash t_i{:}\sigma_i$ is a term, where $\sigma_i \leq \tau_i$ $(1 \leq i \leq n)$, then $\Gamma \vdash p(t_1, \ldots, t_n)$ is a *roled predicate atom.*
- If m is a n-ary method symbol with signature $\tau \times \tau_1 \times \cdots \times \tau_n \Rrightarrow \tau'$ and Γ contains the typings of all the variables in t, t_i, and t', and $\Gamma \vdash t{:}\sigma, \Gamma \vdash t_i{:}\sigma_i$ $(1 \leq i \leq n)$, and $\Gamma \vdash t'{:}\sigma'$ are terms where $\sigma \leq \sigma_s \leq \tau, \sigma_i \leq \tau_i$, and $\sigma' \leq \tau'$ then $\Gamma \vdash t{::}\sigma_s[m@t_1, \ldots, t_n \twoheadrightarrow t']$ is a *roled method atom.*

We have defined roled predicate and roled method atoms in order to refer to roled atoms whatever the kind of atom.

Definition Given a variable typing Γ, roled atoms $\Gamma \vdash A$, $\Gamma \vdash B_1, \ldots,$ $\Gamma \vdash B_n$, $\Gamma \vdash A \leftarrow B_1 \wedge \cdots \wedge B_n$ is a *roled clause.* The roled atom $\Gamma \vdash A$ is the *head* of the clause and $\Gamma \vdash B_1 \wedge \cdots \wedge B_n$ is the *body.*

The definition for roled goal follows from this.

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

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

Example 2.2 This example illustrates monotonic inheritance. The following is a program P with respect to the set of declarations D in Example 2.1:

$$\{x{:}person, y{:}year, z{:}string\} \vdash x[legal\_names@y \twoheadrightarrow z] \leftarrow x[last\_name@y \twoheadrightarrow z]$$
$$\{x{:}female, y{:}year, z{:}string\} \vdash x[legal\_names@y \twoheadrightarrow z] \leftarrow x[maiden\_name \twoheadrightarrow z]$$
$$\{x{:}female, y{:}year, z{:}string\} \vdash x[legal\_names@y \twoheadrightarrow z] \leftarrow$$
$$x{::}person[legal\_names@y \twoheadrightarrow z]$$
$$\{x{:}writer, y{:}year, z{:}string\} \vdash x[legal\_names@y \twoheadrightarrow z] \leftarrow x[pen\_name@y \twoheadrightarrow z]$$
$$\{x{:}writer, y{:}year, z{:}string\} \vdash x[legal\_names@y \twoheadrightarrow z] \leftarrow$$
$$x{::}person[legal\_names@y \twoheadrightarrow z]$$
$$mary[last\_name@1992 \twoheadrightarrow jones]$$
$$mary[maiden\_name \twoheadrightarrow smith]$$

This program states that a *person's* legal name in a given year is their last name in that year, a *female's* legal name in a given year is her maiden name and her legal name as a person in that year, a *writer's* legal name in a given year is their pen name in that year and their legal name as a person in that year. *Mary's* last name in 1992 is *jones* and her maiden name is *smith*. □

2.2 Substitutions

Ground instances of clauses and programs are used in the following section. We now define substitution, how to combine two substitutions, and instances of clauses and programs. These concepts are similar to the corresponding concepts in logic programming, except that they are typed and hence must satisfy additional restrictions.

Definition Let D be a set of declarations. Let $x_1, \ldots, x_n$ be distinct variables, and $\Gamma \vdash t_1, \ldots, \Gamma \vdash t_n$ be terms, where no t_i equals x_i. Then $\Gamma \vdash \{x_1/t_1, \ldots, x_n/t_n\}$ is a *substitution* if, for each i, for some τ_i and σ_i, $\Gamma \vdash x_i : \tau_i$ and $\Gamma \vdash t_i : \sigma_i$ are terms, and $\sigma_i \leq_t \tau_i$ with respect to D.

The empty substitution is denoted by ε.

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

We now describe how substitutions are applied to expressions.

Definition Let D be a set of declarations. 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 $dom(\theta) = \{x_1, \ldots, x_k\}$, $range(\theta) = \{t_1, \ldots, t_k\}$ and $vars(T)$ is the set of all variables in T.

Definition Let D be a set of declarations and θ and σ be the substitutions $\Gamma_\theta \vdash \theta'$ and $\Gamma_\sigma \vdash \sigma'$ where θ' is $\{x_1/t_1, \ldots, x_n/t_n\}$, and σ' is $\{y_1/s_1, \ldots, y_m/s_m\}$. We construct the instance $E\theta$ of E as follows:

- 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 some t_i is x, then the typing of x in Γ' is $x{:}glb(\sigma_i, \tau_i')$ [1] with respect to D; otherwise the typing of x in Γ' is just that in Γ_E.

[1] *glb* is greatest lowest bound

- If for each variable x_i in $vars(T) \cap dom(\theta)$, t_i is a variable and $glb(\sigma_i, \tau_i')$ with respect to D is not $\bot$, or t_i is an object and $\sigma_i \leq_t \tau_i'$ with respect to D 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 2.3 Assume an appropriate set of declarations including those in Example 2.1. Consider the expression $E = \{x{:}writer, y{:}date, z{:}string\} \vdash x[legal\_names@y \twoheadrightarrow z]$ and the substitution $\theta = \{x'{:}female\_writer, y'{:}date, z'{:}string\} \vdash \{x/x',\ y/y', z/z'\}$. Then, the instance $E\theta = \{x'{:}female\_writer, y'{:}date, z'{:}string\} \vdash x'[legal\_names@y' \twoheadrightarrow z']$.

Now consider, substitution $\theta_1 = \{x'{:}person, y'{:}date, z'{:}string\} \vdash \{x/x', y/y',\ z/z'\}$. Instance $E\theta_1$ is undefined, because $person \not\leq_t writer$. $\Box$

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

We now describe how substitutions are composed. We construct the composition $\theta\sigma$ of θ and σ as follows:

- 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 $x{:}\tau \in \Gamma_\theta$ and $x{:}\tau' \in \Gamma_\sigma$ then include $x{:}glb(\tau, \tau')$ with respect to D in Γ, otherwise if $x{:}\tau \in \Gamma_\theta$ and there is no type specified for x in Γ_σ, or $x{:}\tau \in \Gamma_\sigma$ and there is no type specified for x in Γ_θ then include $x{:}\tau$ in Γ.

- If for any variable x, where $x{:}\tau \in \Gamma_\theta$ and $x{:}\tau' \in \Gamma_\sigma$, and $glb(\tau, \tau')$ with respect to D is $\bot$, then $\theta\sigma = \emptyset$, else $\theta\sigma = \Gamma \vdash \gamma$.

Example 2.4 Assume an appropriate set of declarations including those in Example 2.1. Let $\sigma = \{x_1{:}writer, y_2{:}person, y_3{:}female\_writer\} \vdash \{x_1/mary, y_2/x_1, y_3/mary\}$ and $\theta = \{x_1{:}person, x_2{:}writer, x_3{:}female\_writer, y_2{:}person, y_3{:}female\_writer\} \vdash \{x_1/y_2, x_2/y_3, x_3/mary\}$. Then, the composition of θ and σ is $\theta\sigma = \{x_1{:}writer, x_2{:}writer, x_3{:}female\_writer, y_2{:}person, y_3{:}female\_writer\} \vdash \{x_2/mary, x_3/mary, y_2/x_1, y_3/mary\}$.

Now, let $\sigma = \{x_2{:}female, x_3{:}female\} \vdash \{x_2/x_3\}$ and $\theta = \{x_1{:}person, x_2{:}writer\} \vdash \{x_1/x_2\}$. Then, the composition $\theta\sigma = \emptyset$ because $x_2{:}writer \in \{x_1{:}person, x_2{:}writer\}$ and $x_2{:}female \in \{x_2{:}female, x_3{:}female\}$, and $glb(writer, female) = \bot$. $\Box$

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

2.3 Simple programs

To ensure that programs have well-defined semantics, we impose restrictions that ensure they are "stratified" and unambiguous with respect to multiple inheritance. Before defining the classes of stratified and simple programs, we must define the relationships "unroled equivalent" and "possibly overrides" between ground instances of roled clauses.

Definition The *unroled equivalent* of a roled method atom $\Gamma \vdash t{::}\sigma_s[m@t_1,\ \ldots, t_n \twoheadrightarrow t']$ is $\Gamma \vdash t[m@t_1, \ldots, t_n \twoheadrightarrow t']$. For example, $mary[legal\_names@6Feb95 \twoheadrightarrow$ "$smith$"] is the unroled equivalent of $mary{::}person[legal\_names@6Feb95 \twoheadrightarrow$"$smith$"]. The *unroled equivalent* of a roled predicate atom is itself.

Definition Let D be a set of declarations, P be a program with respect to D, C be $\Gamma \vdash t{::}\tau_t[m@t_1, \ldots, t_k \twoheadrightarrow t'] \leftarrow B$, and C' be $\Gamma' \vdash s{::}\sigma_s[m@s_1, \ldots, s_k \twoheadrightarrow s'] \leftarrow B'$, where $C, C' \in P$, $\Gamma \vdash t : \sigma$ is a term, $\Gamma' \vdash s : \tau$ is a term and $\sigma <_t \tau$. 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 and D.

Definition A roled 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 unroled equivalents of ground roled atoms to the set of non-negative integers such that, for every ground instance $C\theta$ of every clause C in P,

- $\mu(B^*) \leq \mu(A^*)$, where B is an atom in the body of the clause instance $C\theta$, A is the head of the clause instance $C\theta$, B^* is the unroled equivalent of B, and A^* is the unroled equivalent of A,
- $\mu(B^*) < \mu(A^*)$ for every ground instance $C'\theta'$ that possibly overrides $C\theta$, where B' is a roled atom in the body of $C'\theta'$, A is the head of the clause instance $C\theta$, $A \neq B'$, B^* is the unroled equivalent of B', and A^* is the unroled equivalent of A,
- $\mu(A^{**}) \leq \mu(A^*)$ for every ground instance $C'\theta'$ that possibly overrides $C\theta$, where A' is the head of $C'\theta'$, A is the head of $C\theta$, A^* is the unroled equivalent of A, and A^{**} is the unroled equivalent of A'.

Informally, programs in which a value of a method depends on the inherited value of the same method have no natural meaning. However, programs in which a value of a method depends on the value of the same method where the type is explicitly stated does have a natural meaning.

Example 2.5 Assume an appropriate set of declarations D including $student < person$ and $jo{:}student$. Consider the program P with respect to D:

$$\{x{:}person\} \vdash x[transport \twoheadrightarrow walks]$$
$$\{x{:}student\} \vdash x[transport \twoheadrightarrow cycles] \leftarrow x[transport \twoheadrightarrow walks].$$

Then $[P]_D$ is:

$$jo{::}person[transport \twoheadrightarrow walks]$$
$$jo{::}student[transport \twoheadrightarrow cycles] \leftarrow jo{::}student[transport \twoheadrightarrow walks].$$

The i-stratification of $[P]_D$ requires:

$$\mu(jo[transport \twoheadrightarrow walks]) \leq \mu(jo[transport \twoheadrightarrow cycles])$$
$$\mu(jo[transport \twoheadrightarrow walks]) < \mu(jo[transport \twoheadrightarrow walks])$$
$$\mu(jo[transport \twoheadrightarrow cycles]) \leq \mu(jo[transport \twoheadrightarrow walks]),$$

which is unsatisfiable. Hence P is not i-stratified.

Now, consider the program P' with respect to D:

$$\{x{:}person\} \vdash x[transport \twoheadrightarrow walks]$$
$$\{x{:}student\} \vdash x[transport \twoheadrightarrow cycles] \leftarrow x{::}person[transport \twoheadrightarrow walks].$$

Then $[P']_D$ is:

$$jo{::}person[transport \twoheadrightarrow walks]$$
$$jo{::}student[transport \twoheadrightarrow cycles] \leftarrow jo{::}person[transport \twoheadrightarrow walks].$$

The i-stratification of $[P']_D$ requires:

$$\mu(jo[transport \twoheadrightarrow walks]) \leq \mu(jo[transport \twoheadrightarrow cycles])$$
$$\mu(jo[transport \twoheadrightarrow cycles]) \leq \mu(jo[transport \twoheadrightarrow walks]),$$

which is satisfiable. Hence P' is i-stratified. $\Box$

A program is unambiguous with respect to multiple inheritance if there is only one value that can be inherited for every method in that program. One way to ensure this condition is to check it at run time. However, it is preferable to be able to check this condition at compile time. The following syntactic condition ensures that programs are unambiguous in this sense.

Definition Let P be a program with respect to a set of declarations D. For each type σ in D, let $R(\sigma)$ be $\{\langle m_{\twoheadrightarrow}, n, \tau, m'_{\twoheadrightarrow}, n', \tau'\rangle \mid$ for each method atom $m'_{\twoheadrightarrow}$ of arity n' of type τ' in the body of every clause that defines method $m_{\twoheadrightarrow}$ of arity n on type τ, $\sigma \leq_t \tau$, and $m'_{\twoheadrightarrow} = \_$, $n' = \_$, and $\tau' = \_$ if there are no method atoms in the body of the definition $\}$. Program P is *unambiguous* with respect to multiple inheritance if and only if for every type σ in D, if $\langle m_{\twoheadrightarrow}, n, \tau_1, m'_{1,\twoheadrightarrow}, n'_1, \tau'_1\rangle \in R(\sigma)$ and $\langle m_{\twoheadrightarrow}, n, \tau_2, m'_{2,\twoheadrightarrow}, n'_2, \tau'_2\rangle \in R(\sigma)$, where $\tau_1 \neq \tau_2$, then either $\tau_1 <_t \tau_2$ or $\tau_2 <_t \tau_1$, or if $\tau_1 \not<_t \tau_2$ and $\tau_2 \not<_t \tau_1$ then there is a tuple $\langle m_{\twoheadrightarrow}, n, \sigma, m_{\twoheadrightarrow}, n, \tau\rangle$ such that $\sigma <_t \tau$ and there is a tuple $\langle m_{\twoheadrightarrow}, n, \tau, m'_{\twoheadrightarrow}, n', \tau'\rangle \in R(\sigma)$.

Example 2.6 Consider program P with respect to declarations D in Example 2.2.
$R(person) = \{\langle legal\_names, 1, person, last\_name, 1, person\rangle\}$
$R(female) = \{\langle legal\_names, 1, person, last\_name, 1, person\rangle,$
$\langle legal\_names, 1, female, last\_name, 1, person\rangle,$
$\langle legal\_names, 1, female, maiden\_name, 0, female\rangle\}$
$R(writer) = \{\langle legal\_names, 1, person, last\_name, 1, person\rangle,$
$\langle legal\_names, 1, writer, last\_name, 1, person\rangle,$
$\langle legal\_names, 1, writer, pen\_name, 1, writer\rangle\}$
$R(female\_writer) = \{\langle legal\_names, 1, person, last\_name, 1, person\rangle,$
$\langle legal\_names, 1, female, last\_name, 1, person\rangle,$
$\langle legal\_names, 1, female, maiden\_name, 0, female\rangle,$
$\langle legal\_names, 1, writer, last\_name, 1, person\rangle,$
$\langle legal\_names, 1, writer, pen\_name, 1, writer\rangle\}$.
So, $\langle legal\_names, 1, female, maiden\_name, 0, female\rangle \in R(female\_writer)$ and $\langle legal\_names,$ $1, writer, pen\_name, 1, writer\rangle \in R(female\_writer)$ where $female \neq writer$, and $female \not<_t$ $writer$, and $writer \not<_t female$, and there is no tuple $\langle legal\_names, 1, female\_writer, legal\_names,$ $1, \tau\rangle$ such that $female\_writer <_t \tau$ and $\langle legal\_names, 1, \tau, m', n', \tau'\rangle \in R(female\_writer)$. Thus, this program is not unambiguous due to multiple inheritance. This result is not surprising. Intuitively, when $legal\_names$ is applied to objects of type $female\_writer$, it is not clear whether the definitions should be inherited from type $female$ or type $writer$.

Now, consider program P with the added clause

$$\{x{:}female\_writer, , y{:}year, z{:}string\} \vdash x[legal\_names@y \twoheadrightarrow z] \leftarrow$$
$$x{::}female[legal\_names@y \twoheadrightarrow z].$$

with respect to the appropriate declarations D.
Now, $R(female\_writer) = \{\langle legal\_names, 1, person, last\_name, 1, person\rangle,$
$\langle legal\_names, 1, female, last\_name, 1, person\rangle,$
$\langle legal\_names, 1, female, maiden\_name, 0, female\rangle,$
$\langle legal\_names, 1, writer, last\_name, 1, person\rangle,$
$\langle legal\_names, 1, writer, pen\_name, 1, writer\rangle,$

$\langle legal\_names, 1, female\_writer, legal\_names, 1, female\rangle\}$.

There is a tuple $\langle legal\_names, 1, female\_writer, legal\_names, 1, female\rangle$ such that $female\_writer <_t female$ and a tuple $\langle legal\_names, 1, female, maiden\_name, 0, female\rangle \in R(female\_writer)$. Thus, this new program is unambiguous with respect to multiple inheritance. $\Box$

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

- P with respect to D is i-stratified, and
- P with respect to D is unambiguous with respect to multiple inheritance.

3 Semantics

In this section, we describe the declarative semantics of roled programs. As in logic programming, the meaning of a program can be given by a preferred model. A simple program may have more than one model, so we define a priority relation that is used to identify preferred models of a program.

The interpretation assigns a truth value to every ground instance of a clause in a program. We first describe the assignment of a truth value to an atom.

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

- If A is $\Gamma \vdash p(t_1, \ldots, t_n)$, and $t'_1, \ldots, t'_n$ are the term assignments of $t_1, \ldots, t_n$ (with respect to $\mathcal{T}$ and Γ), then A is true in I (with respect to $\mathcal{T}$ and Γ) if $p(t'_1, \ldots, t'_n) \in I$.
- If A is $t{::}\sigma[m@t_1, \ldots, t_k \twoheadrightarrow t_n]$ and $t', t'_1, \ldots, t'_k, t'_n$ are the term assignments of $t, t_1, \ldots, t_k, t_n$ (with respect to $\mathcal{T}$ and Γ), then A is true in I (with respect to $\mathcal{T}$ and Γ) if $t'{::}\tau[m@t'_1, \ldots, t'_k \twoheadrightarrow t'_n] \in I$, where $\sigma \leq \tau$, and there is no atom $t'{::}\sigma'[m@t'_1, \ldots, t'_k \to t''_n] \in I$ such that $t''_n \neq t'_n$ and $\sigma \leq \sigma' < \tau$.

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

We now define the "model" and "preferred model" of a roled program.

Definition Let P be a roled program and I an interpretation. We say that I is a *roled model* of P if, for every ground clause $C \in [P]$, C is true in I.

We need the notion of minimal atom when we define r-preferable model.

Definition A roled method atom $t{::}\tau[m@t_1, \ldots, t_k \twoheadrightarrow t_n]$ in an interpretation I is *minimal* if there is no other atom $t{::}\sigma[m@t_1, \ldots, t_k \twoheadrightarrow t'_n]$ in I such that $\sigma < \tau$. Every predicate atom in an interpretation is *minimal*.

Example 3.1 Assume a set of declarations D including $failure{:}cf$, $cf < csf$ and $csf < sf$. Consider the program P with respect to D:

$\{x{:}sf\} \vdash x[treatment \twoheadrightarrow$ "tell supervisor"$]$
$\{x{:}cf\} \vdash x[treatment \twoheadrightarrow$ "lower temperature"$]$

The two interpretations
$I_1 = \{failure{::}cf[treatment \twoheadrightarrow$ "lower temperature"$]$,
$failure{::}sf[treatment \twoheadrightarrow$ "tell supervisor"$]\}$ and

$I_2 = \{failure{::}csf[treatment \twoheadrightarrow$ "lower temperature"],
$failure{::}sf[treatment \twoheadrightarrow$ "tell supervisor"]$\}$ are models of P.

In model I_1, $failure{::}cf[treatment \twoheadrightarrow$ "lower temperature"] is a minimal atom and in model I_2, $failure{::}csf[treatment \twoheadrightarrow$ "lower temperature"] is a minimal atom. These atoms are the only minimal atoms in I_1 and I_2 (respectively). $\Box$

The goal of the following preference relation is to give priority to minimal atoms.

Definition Suppose that M and N are two distinct models of a roled program P with respect to a set of declarations D. Then, N is *r-preferable* to M ($N \ll M$) if, for every atom A in $N - M$, there is a minimal atom B in M such that $A = t{::}\sigma[m@t_1, \ldots, t_k \twoheadrightarrow t_n]$, $B = t{::}\tau[m@t_1, \ldots, t_k \twoheadrightarrow t_n]$, and $\sigma < \tau$. We write $N \leq M$ if $N \ll M$ or $N = M$. We say model N is an *r-preferred* model of P if there are no models of P r-preferable to N.

Example 3.2 Consider program P with respect to the declarations D of Example 3.1 that has two models. Because $failure{::}cf[treatment \rightarrow$ "lower temperature"] $\in I_1 - I_2$ and $failure{::}csf[treatment \rightarrow$ "lower temperature"] is a minimal atom in I_2, and this condition does not hold for any atoms in $I_2 - I_1$, model I_1 is r-preferable to I_2. $\Box$

The following theorem shows that each simple program has a unique r-preferred model.

Theorem 3.1 *Let D be a set of declarations, and P with respect to D be a simple program. Then P with respect to D has exactly one r-preferred model, which we denote M_P. For every model M for P with respect to D, we have $M_P \ll M$.*

The proof of this theorem is based on M_P being the smallest model in which all the ground clauses of the program are true, and the model that contains the minimal atoms. The second part of the theorem is proved by contradiction.

As indicated in the examples shown, the unique r-preferred model of a simple program corresponds to the natural meaning of the program.

4 Evaluation

In this section, we give a translation from roled programs to Datalog (with negation), thus providing an evaluation procedure for roled goals with respect to roled programs.

An object declaration $a{:}\tau$ is translated to an atom $\$type(a, \tau)$. A type hierarchy declaration $\sigma < \tau$ is translated to $\$sub(\sigma, \tau)$. There are clauses included in the translation that represent the transitive closure of the type lattice

$$\begin{aligned}
&\$sub^{+}(x, y) \leftarrow \$sub(x, y).\\
&\$sub^{+}(x, y) \leftarrow \$sub(x, z) \wedge \$sub^{+}(z, y).\\
&\$sub^{*}(x, x) \leftarrow \$type(y, x).\\
&\$sub^{*}(x, y) \leftarrow \$sub^{+}(x, y).
\end{aligned}$$

A variable typing $\Gamma = \{x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n\}$ is translated to a conjunction of atoms $\Gamma' = \$type(x_1, y_1) \wedge \$sub^{*}(y_1, \sigma_1) \wedge \cdots \wedge \$type(x_n, y_n) \wedge \$sub^{*}(y_n, \sigma_n)$. The atom $\$m(\sigma, t, t_1, \ldots, t_n, t')$ is a translation of a method atom $t{::}\sigma[m@t_1, \ldots, t_n \twoheadrightarrow t']$. A predicate atom is left unchanged in the translation. A clause $\Gamma \vdash A \leftarrow B$ where A is a roled predicate atom is translated to $A' \leftarrow \Gamma' \wedge B'$ where A' is the translation of A, Γ' is the translation of Γ and B' is the translation of B. For any set of clauses that define a method m of arity k

$$\{t_p{:}\tau_p\} \cup \Gamma_p \vdash t_p[m@t_{1_p}, \ldots, t_{k_p} \twoheadrightarrow t_{n_p}] \leftarrow B_p$$
$$\vdots$$
$$\{t_1{:}\tau_1\} \cup \Gamma_1 \vdash t_1[m@t_{1_1}, \ldots, t_{k_1} \twoheadrightarrow t_{n_1}] \leftarrow B_1$$
$$\{t{:}\sigma\} \cup \Gamma \vdash t[m@t_{1_\sigma}, \ldots, t_{k_\sigma} \twoheadrightarrow t_{n_\sigma}] \leftarrow B$$

where $\sigma < \tau_1 < \cdots < \tau_p$, there are clauses in the translation

$$\$m(\sigma, t, t_{1_\sigma}, \ldots, t_{k_\sigma}, t_{n_\sigma}) \leftarrow B' \wedge \$type(t, y) \wedge \$sub^*(y, \sigma) \wedge \Gamma' \quad (1)$$
$$app_{\$m}(\sigma, t, t_{1_\sigma}, \ldots, t_{k_\sigma}) \leftarrow B' \wedge \$type(t, y) \wedge \$sub^*(y, \sigma) \wedge \Gamma' \quad (2)$$
$$\$m(\tau_1, t_1, t_{1_1}, \ldots, t_{k_1}, t_{n_1}) \leftarrow B'_1 \wedge \$type(t_1, y) \wedge \$sub^*(y, \tau_1) \wedge \Gamma'_1 \quad (3)$$
$$app_{\$m}(\tau_1, t_1, t_{1_1}, \ldots, t_{k_1}) \leftarrow B'_1 \wedge \$type(t_1, y) \wedge \$sub^*(y, \tau_1) \wedge \Gamma'_1 \quad (4)$$
$$\vdots$$
$$\$m(\tau_p, t_p, t_{1_p}, \ldots, t_{k_p}, t_{n_p}) \leftarrow B'_p \wedge \$type(t_p, y) \wedge \$sub^*(y, \tau_p) \wedge \Gamma'_p \quad (5)$$
$$app_{\$m}(\tau_p, t_p, t_{1_p}, \ldots, t_{k_p},) \leftarrow B'_p \wedge \$type(t_p, y) \wedge \$sub^*(y, \tau_p) \wedge \Gamma'_p \quad (6)$$
$$\$m(\sigma, t, t_1, \ldots, t_k, x_n) \leftarrow \$m(\tau_1, t, t_1, \ldots, t_k, x_n) \wedge \$type(t, y) \wedge \$sub^*(y, \sigma) \quad (7)$$
$$\wedge \Gamma' \wedge \neg app_{\$m}(\sigma, t, t_1, \ldots, t_k)$$
$$\$m(\tau_1, t, t_1, \ldots, t_k, x_n) \leftarrow \$m(\tau_2, t, t_1, \ldots, t_k, x_n) \wedge \$type(t, y) \wedge \quad (8)$$
$$\$sub^*(y, \tau_1) \wedge \Gamma'_1 \wedge \neg app_{\$m}(\sigma, t, t_1, \ldots, t_k) \wedge \neg app_{\$m}(\tau_1, t, t_1, \ldots, t_k)$$
$$\vdots$$
$$\$m(\tau_{p-1} t, t_1, \ldots, t_k, x_n) \leftarrow \$m(\tau_p, t, t_1, \ldots, t_k, x_n) \wedge \$type(t, y) \wedge \quad (9)$$
$$\$sub^*(y, \tau_{p-1}) \wedge \Gamma'_{p-1} \wedge \neg app_{\$m}(\sigma, t, t_1, \ldots, t_k) \wedge \neg app_{\$m}(\tau_1, t, t_1, \ldots, t_k) \wedge$$
$$\cdots \wedge \neg app_{\$m}(\tau_{p-1}, t, t_1, \ldots, t_k)$$
$$\$m(z, t, t_1, \ldots, t_k, x_n) \leftarrow \$m(\sigma, t, t_1, \ldots, t_k, x_n) \wedge \$type(t, z) \wedge \$sub^+(z, \sigma) \quad (10)$$
$$\$m(z, t, t_1, \ldots, t_k, x_n) \leftarrow \$m(\tau_1, t, t_1, \ldots, t_k, x_n) \wedge \$type(t, z) \wedge \quad (11)$$
$$\$sub^+(\sigma, z) \wedge \$sub^+(z, \tau_1)$$
$$\vdots$$
$$\$m(z, t, t_1, \ldots, t_k, x_n) \leftarrow \$m(\tau_p, t, t_1, \ldots, t_k, x_n) \wedge \$type(t, z) \wedge \quad (12)$$
$$\$sub^+(\tau_{p-1}, z) \wedge \$sub^+(z, \tau_p),$$

where B' is the translation of B, B'_1 is the translation of B_1, B'_p is the translation of B_p, Γ' is the translation of Γ, Γ'_1 is the translation of Γ_1, Γ'_p is the translation of Γ_p.

The translation of clauses with methods in the head is particularly important. This is where inheritance is considered in the translation. Clauses 1 to 6 give the value of the method when applied to an object of the specified type, including subtypes. Clauses 7 to 9 deal with overriding and clauses 10 to 12 incorporate inheritance. The value of a method for an object of a type which is smaller than any which have the method defined on them, must inherit from the smallest type on which the method is defined. The value of a method for an object of a type between two types on which the method has been defined inherits from the greater of the two types.

We illustrate the translation in the following example.

Example 4.1 Assume a set of declarations D including $failure{:}cf$, $csf < sf$, and $cf < csf$. Consider the program P with respect to the declarations D:

$$\{x{:}sf\} \vdash x[problem \twoheadrightarrow \text{“system failure”}]$$
$$\{x{:}cf\} \vdash x[problem \twoheadrightarrow \text{“over efficient condenser”}] \leftarrow x[condenser \twoheadrightarrow low]$$
$$failure[condenser \twoheadrightarrow low].$$

This is translated to the Datalog program P':

$\$type(failure, cf).$
$\$sub(cf, csf).$
$\$sub(csf, sf).$
$\$sub^+(x, y) \leftarrow \$sub(x, y).$
$\$sub^+(x, y) \leftarrow \$sub(x, z) \wedge \$sub^+(z, y).$
$\$sub^*(x, x) \leftarrow \$type(y, x).$
$\$sub^*(x, y) \leftarrow \$sub^+(x, y).$
$\$problem(sf, x, \text{“system failure”}) \leftarrow \$type(x, y) \wedge \$sub^*(y, sf).$
$app_{\$problem}(sf, x) \leftarrow \$type(x, y) \wedge \$sub^*(y, sf).$
$\$problem(cf, x, \text{“over efficient condenser”}) \leftarrow \$type(x, y) \wedge \$sub^*(y, cf) \wedge$
$\quad \$condenser(cf, x, low).$
$app_{\$problem}(cf, x) \leftarrow \$type(x, y) \wedge \$sub^*(y, cf) \wedge \$condenser(cf, x, low).$
$\$problem(cf, x, y) \leftarrow \$problem(sf, x, y) \wedge \$type(x, y) \wedge \$sub^*(y, cf) \wedge$
$\quad \neg app_{\$problem}(cf, x)$
$\$condenser(cf, failure, low) \leftarrow \$type(failure, y) \wedge \$sub^*(y, cf).$
$app_{\$condenser}(cf, failure) \leftarrow \$type(failure, y) \wedge \$sub^*(y, cf).$

$\$problem(z, x, y) \leftarrow \$problem(sf, x, y) \wedge \$type(x, z) \wedge \$sub^+(cf, z) \wedge$
$\quad \$sub^+(z, sf).$
$\$problem(z, x, y) \leftarrow \$problem(cf, x, y) \wedge \$type(x, z) \wedge \$sub^+(z, cf).$
$\$condenser(z, failure, y) \leftarrow \$condenser(cf, failure, y) \wedge \$type(failure, z) \wedge$
$\quad \$sub^+(z, cf).$ $\square$

We now describe a translation from an interpretation of a roled program to an interpretation of a Datalog program. Given an interpretation I of a roled program P (with respect to a set of declarations D) the translation I' of I (with respect to P) is constructed as follows. Each predicate or method atom in I is translated to a predicate or method atom as given above. For any set of atoms which are translations of method atoms for method m of arity k $\{\$m(\tau_1, t, t_1, \ldots, t_k, t_{\tau_1}), \$m(\tau_2, t, t_1, \ldots, t_k, t_{\tau_2}), \ldots, \$m(\tau_p, t, t_1, \ldots, t_k, t_{\tau_p})\}$ in I' such that there are no atoms $\$m(\sigma_i, t, t_1, \ldots, t_k, t_{\sigma_i})$ in I', where $1 \leq i \leq p+1$ and $\sigma_1 < \tau_1 < \sigma_2 < \tau_2 < \cdots < \sigma_p < \tau_p < \sigma_{p+1}$, there is an atom $\$m(\sigma_i, t, t_1, \ldots, t_k, t_{\tau_i})$ included in I' for every $t{:}\sigma_i$ in D, $1 \leq i \leq p$. For every declaration $o{:}\tau$ in D there is an atom $\$type(o, \tau)$ included in I'. For every declaration $\tau < \sigma$ in D there is an atom $\$sub(\tau, \sigma)$ in I'. The transitive closure of $\$sub$ is described by $\$sub^*$ and $\$sub^+$ in I'. The applied clauses are described by *app*-predicates in I'.

Example 4.2 Consider program P and the set of declarations D given in Example 4.1. The interpretation $I = \{failure{::}sf[problem \twoheadrightarrow \text{“system failure”}], failure{::}\ cf[problem \twoheadrightarrow \text{“over efficient condenser”}], failure{::}\ cf[condenser \twoheadrightarrow low]\}$ is an interpretation of P with respect to D .

Translating these atoms gives $I_t = \{\$problem(sf, failure, \text{“system failure”}), \$problem(cf, failure, \text{“over efficient condenser”}), \$condenser(cf, failure, low)\}$. Considering the types between cf and sf gives $I_b = \{\$problem(csf, failure, \text{“system failure”})\}$. Considering the declarations gives $I_D = \{\$type(failure, cf), \$sub(csf, sf), \$sub(cf, csf), \$sub^*(csf, sf),$ $\$sub^*(cf, csf), \$sub^*(cf, sf)\}$. Finally, considering the clauses which are applied gives $I_A = \{app_{\$problem}(cf, failure), app_{\$problem}(sf, failure), app_{\$condenser}\ (cf, failure)\}$.

Then I', the translation of I with respect to D, is $I_t \cup I_b \cup I_D \cup I_A$. $\Box$

The following two lemmas are used in the proof of the correctness of this translation, stated as Theorem 4.1 below.

Lemma 4.1 *Let D be a set of declarations, P with respect to D be a positive simple roled program and P' be the Datalog translation of P with respect to D. Then P' is locally stratified.*

Lemma 4.2 *Let D be a set of declarations, P with respect to D be a positive simple roled program and M the r-preferred model of P with respect to D. Let P' be the translation of P with respect to D and M' the translation of M (with respect to P and D). Then M' is the preferred model of P' [10].*

The following theorem shows that standard query evaluation can be used to compute answers for roled goals on roled programs after they have been rewritten and the expected answer is computed.

Theorem 4.1 *Let D be a set of declarations, P with respect to D be a positive simple program and Q an atomic goal $\Gamma \vdash \leftarrow A$. Let P' be the translation of P with respect to D and Q' be the translation of Q with respect to D. If θ is a computed answer for $P' \cup \{Q'\}$, then θ is a correct answer for $P \cup \{Q\}$.*

The proof of this theorem reasons that if M is the preferred model of P with respect to D then the translation of M, M' is the perfect model of P'. Thus if θ is a computed answer for $P' \cup \{Q'\}$ then $(\Gamma \vdash \leftarrow A)\theta$ is true in M.

5 Discussion

This paper provides a mathematical foundation for an object-oriented language that uses qualifiers or roles to resolve ambiguities due to multiple inheritance. Roles also facilitate the use of the word "super" to indicate that the method in the classes superclass is to be used. For object-oriented languages with dynamic overriding [1], roles further allow the modeling of monotonic inheritance of multi-valued methods.

In this paper, we outlined the syntax and semantics of the language and provided a query evaluation procedure. The syntax included the declaration of types and the relationships between the types, as well as the data definition. We described a class of programs called "simple programs" which are stratified and unambiguous with respect to multiple inheritance. This class of programs is important because each simple program has a unique r-preferred model which corresponds to the natural semantics for the program. The query evaluation procedure translated the goals and programs to Datalog (with negation) and we showed that for simple programs evaluation of the Datalog translations provides the correct answer with respect to the semantics provided by the r-preferred model.

This work is an extension of the foundation which we provided for unroled object-oriented languages [5]. It also extends the work in [1], extending object-oriented languages with dynamic overriding with monotonic inheritance. The language described in [7] is richer than the language described in this paper, allowing both the modeling of monotonic inheritance and the qualification of methods. We feel that in providing a minimum extension to a language we have already investigated and in working in the same framework, we are able to gain a good understanding of the extension.

This work provides a first step to allowing type information to be used in clauses for purposes other than type checking. The next step would be to extend the language to allow further type information in the program which would enable the dynamic population of classes and the creation of virtual hierarchies. It would be interesting and instructive to provide a simple semantics like the one provided here for languages with more type information in the program. This work provides a firm basis for such an extension.

References

[1] 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.

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

[3] B. J. Cox. *Object-oriented programming; an evolutionary approach.* Addison-Wesley, Reading, Mass., 1987.

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

[5] G. Dobbie and R. W. Topor. A model for sets and multiple inheritance in deductive object-oriented systems. In *Proceedings of the Third International Conference on Deductive and Object-Oriented Databases*, Scottsdale, Arizona, 1993.

[6] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley, 1983.

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

[8] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988.

[9] D. A. Moon. The COMMON LISP object-oriented programming language. In W. Kim and F. H. Lochovsky, editors, *Object-oriented concepts, Databases, and Applications*, pages 49–78. ACM Press Books, 1989.

[10] T. C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, 1988. Further revised as [11].

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

[12] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 2nd edition, 1991.

[13] D. Ungar and R. B. Smith. Self: The power of simplicity. In *1987 Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pages 227–242, 1987.