Implementing Abstract Objects with Inheritance in Datalog$^{\text{neg}}$

Hasan M. Jamil
Department of Computing
School of MPCE, Macquarie University
Sydney, NSW 2109, Australia
e-mail: jamil@mpce.mq.edu.au

Abstract

We present an elegant technique to reduce *inheritance* and *encapsulation* to pure deduction. The reduction technique presented in this paper makes it possible to model object-oriented database features in a purely deductive system. Encapsulation has been given a formal treatment for the first time by introducing the so called *context-resolution* scheme. The *completion* technique presented in this paper elegantly tackles inheritance with overriding and conflict resolution by avoiding non-monotonic reasoning. We show that the completion based reduction technique is robust and appealing compared to any other known rewriting based approaches. We propose an object-oriented front-end language called the *Datalog*$^{++}$, and discuss a rewriting scheme to the acclaimed Datalog$^{\text{neg}}$ for this language that exploits the context resolution and completion techniques presented here. We claim that our approach outperforms other known approaches in the literature in terms of its modeling capabilities and efficiency. Unlike most others, an implementation based on our reduction technique does not require meta-interpretation and consequently readily exploits the rich set of optimization techniques available in Datalog$^{\text{neg}}$.

1 Introduction

Deductive and object-oriented databases are two parallel trends evolved over the past few years in database technology having mutually exclusive features. Following the huge research experience of deductive databases, and motivated by the need for the design of complex applications that demand features characteristic to the object-oriented systems, the need for logic based object-oriented query languages have emerged. This motivation solidified further because of the belief that the next generation database technology will require object-oriented features such as complex objects, inheritance, encapsulation and inference capability.

Recently, there have been intense activities in the research front to develop a deductive object-oriented language. However, there have been no significant development so far. In the last few years, several proposals also addressed the issue of a direct semantics for logic based object-oriented languages, for example [5, 7, 11, 12]. It was readily observed that giving a direct semantics to a language capable of modeling desired object-oriented features is a daunting task. While few experimental systems started to emerge, they are far less likely to be used commercially than their deductive database counterparts. This is simply because these systems either lack a formal foundation, are too complex to be used as modeling tools, or are plainly inadequate for useful applications.

We consider an extension of Datalog, called the Datalog^{++}, in the direction of the so called *object-relational* models. The goal here is to develop a language that has most of the desired object-oriented features well within the setting of an existing system, is adequate to use as a modeling tool and at the same time has a formal foundation. Hence, we try to be practical and instead of trying to develop a direct semantics of our language, we develop a rewriting based semantics. That is, to give a semantics to a program in our language we rewrite every Datalog^{++} program to another language, Datalog$^{\text{neg}}$ to be exact. Similar approaches have been studied in works by Abiteboul et al. [1], Dalal and Gangopadhyay [8], Naish [2], Lawley [14], etc. The advantages of such an approach are as follows. Inheritance with overriding can be modeled in a practical manner. While encapsulation can also be incorporated by developing suitable machinery and semantics, as we demonstrate in this paper, unfortunately none of the works known to us have addressed this important issue so far. However, the lack of a complete logical semantics for both of these features – inheritance with overriding and encapsulation, necessitates a meta-logical treatment of these features and thus motivates the rewriting based approach to object modeling presented in this paper.

Proceedings of the 23rd VLDB Conference
Athens, Greece, 1997

1.1 The Merits and Contributions of Datalog^{++}

We must mention here that a rewriting or translation based approach to object modeling is not altogether new. Several researchers have used this technique to develop semantics for object-oriented logics including [1, 2, 8, 14][1]. But, our proposal should not be dismissed only on the ground that we too take a translational approach. In this section, we try to convince the skeptics why Datalog^{++} should be considered seriously.

Among the languages proposed in [1, 2, 8, 14], NUOO Prolog [2] perhaps is the closest in spirit to Datalog^{++} but have several important differences. The characteristics peculiar to Datalog^{++} makes it possible to be used as a practical database language while, others, specially NUOO Prolog, are not. But more importantly, all these languages fall short of providing a clean class interface facility, namely encapsulation – one of the two most essential features of object-oriented systems. Encapsulation is regarded as the basis for modularity and implementation independence in object-oriented systems. Without encapsulation, an object-oriented language may be regarded as incomplete. Furthermore, in works like [14], clauses implementing methods and predicates are mapped to terms in meta-predicates. This approach defeats the query optimization techniques present in deductive databases and hence can not be regarded as a candidate for database applications. While NUOO Prolog handles inheritance somewhat similarly as in Datalog^{++}, they really do not provide the high level abstractions such as class and instance methods, code and value inheritance, etc. and thus limits itself in terms of modeling flexibility.

Another advantage of Datalog^{++} is that the inheritability can be compiled and stored making rule invocation (during inheritance) equivalent to an index look-up using constants, a feature that no other languages have. In fact, the decision on inheritance of a rule is non-deterministic in the languages like Datalog^{meth} [1], F-logic interpreter[14], NUOO Prolog [2] and OOLP+ [8] and must be deferred until at run time. This is a significant impediment when practical sized databases are considered that employ the so called *dynamic inheritance* a la F-logic and Datalog^{meth}.

We can summarize the contributions and advantages of Datalog^{++} as follows: (i) it provides a clean interface mechanism (encapsulation) for objects and classes in a declarative style, (ii) it provides a superior inheritance mechanism based on the idea of i-completion making it possible to compile inheritability, and (iii) it provides higher level abstractions for capturing both value and code inheritance in a single set up, and facilities to declare class and instance methods. While these are the most important contributions of this paper over the existing languages, we will address other issues as well in section 5. In addition to the above it also provides a mechanism to resolve inheritance conflict in multiple inheritance, allows selection of a preferred inheritance, and utilizes the idea of inheritability which is the key to a partial compilation of the database and to conflict resolution.

1.2 Organization of this Paper

We introduce few not so familiar concepts such as *value* and *code* inheritance, *locality* and *inheritability* of clauses, *accessibility* of methods, etc. These concepts allow us to develop a reduction technique for Datalog^{++} programs to Datalog^{neg} and thus help explain the meta-logical features in a logical way. Furthermore, the reduction technique helps us model inheritance and encapsulation in purely deductive ways without having to deal with non-monotonic reasoning.

The material contained in this paper is based on the data model proposed in a preliminery version in [10]. For the lack of space and brevity, we do not repeat the data model here. Interested readers may find a discussion on the data model, and numerous examples that better clarify the concepts introduced in this paper in [10], and may regard it as a companion paper. In the examples in [10] we step by step reduce the Datalog^{++} program **D** in Example 2.1 to a Datalog^{neg} program $\mathbf{D}_r$. An extended version of this paper that also subsumes [10] discusses the implementation issues of Datalog^{++} and may be found in [9].

The rest of the paper is organized as follows. We first introduce the Datalog^{++} language using an example along with its intended semantics on intuitive grounds in section 2. Then we formally present the syntax in section 3. The rewriting based semantics is presented in section 4 by introducing few new concepts and by giving a translation algorithm to reduce every Datalog^{++} program to Datalog^{neg}. We compare our work with few contemporary research and discuss implementation issues in section 5 and finally give our conclusion in section 6.

2 The Datalog^{++} Language Overview

We now introduce the syntax and the semantics of the Datalog^{++} language with an overview to its salient features. The goal here is to develop a logic based language to represent and query deductive object-oriented databases. Datalog^{++} extends Datalog syntax to capture the concepts such as classes, objects, signatures, is-a, methods, etc., sometimes in a meta-logical way. While the extension in Datalog^{++} is syntactical, the semantic interpretation of every Datalog^{++} program is still given in Datalog^{neg} – a variant of Datalog that incorporates *negation*[2] and hence has a first-order interpretation. Since some of the Datalog^{++} features lack logical interpretation (meta-logical in nature), the semantics relies on a translation function to Datalog^{neg} such that every translated feature is given a relational characterization. In this way, it becomes possible to capture most of the non-standard object-oriented notions in a meta-logical way but yet simultaneously give a logical interpretation to these features. We will show that not only our semantics is richer, stronger and more intuitive than Datalog^{meth} [1], OOLP+ [8], logic language in [3], NUOO Prolog [2] and similar other languages, our syntax is also much more flexible that allows users to model their universe of discourse in an object-oriented way as opposed to relational ways as in some of these proposals [1, 3].

2.1 Informal Semantics of Datalog^{++}

A Datalog^{++} program basically is a set of objects, relationships, properties – methods and attributes, and object hierarchy definitions. Recall that objects are of two types: *class* and *instance* objects. A class definition includes structure description via signatures and a set of

[1]Observe that these languages are the most representative in the literature. Others are either similar or weaker than these. Hence it suffices to compare Datalog^{++} with this set.

[2]Notice that the choice of the negation semantics in Datalog^{neg} does not affect the semantics of Datalog^{++} programs.

method implementations. An instance object is similar to a class object except that instances do not define the structural aspects of objects, and can not have instances or subclasses of their own.

We now use a simple example to explain the intended semantics of Datalog^{++} before we formally discuss its syntax and the semantics. This will expose the motivation behind the language structure and the concepts behind the semantic interpretation of Datalog^{++} programs. Notice that an extended definite clause fragment of predicate logic is used to define methods in objects in the following example.

Example 2.1 Consider the miniature university database $\mathbf{D} = \langle D_\Sigma, D_<, D_\Psi, D_\Pi \rangle$ shown below[3]. The database consists of three class objects – *grad_stud*, *gta*, and *faculty*, and several instance objects – *joe, kelly, sally, sue, john* and *max.*

```
class grad_stud
{
    instance signatures
    {
        pub, val name/1;
        pub, val sid/1;
        priv, val stipend/1;
        pub, code income/1;
        pub, val avg_income/1;
        pub, code meandev/1;
    }
}
```

The signature component of the class definition for *grad_stud* above says that *name* is an attribute/method of arity 1, denoted *name/1*, of every instance of *grad_stud*. Furthermore, instances inherit the value of *name* from *grad_stud* and the method is public and hence is visible to any object in this database[4]. Similarly, *sid* (student id), and *avg_income* both of arity 1 are value inheritable public methods for the instances of this class. However, *stipend* is a private value inheritable method and thus is only accessible by the instances themselves and their class objects. On the other hand, while *income* and *meandev* (mean deviation) are public methods for the instances of *grad_stud*, they are code inheritable.

```
(1) grad_stud:stipend(12K);
(2) grad_stud:income(X) ← stipend(X);
(3) grad_stud:avg_income(avg(<I>)) ← O∈grad_stud,
      O≪income(I);
(4) grad_stud:meandev(X) ← income(I),
      avg_income(A), X=abs(I−A);
(5) joe∈grad_stud;
(6) joe:stipend(15K);
(7) kelly∈grad_stud;
(8) kelly:income(X) ← john≪salary(X);
```

Rules (1) through (4) define *local* methods *stipend/1, income/1, avg_income/1* and *meandev/1* in *grad_stud*. Rules (6) and (8) define local methods *stipend/1* and *income/1* respectively for *joe* and *kelly*. Rules (5) and (7) define class memberships of *joe* and *kelly* and captures the fact that they are instances of *grad_stud*.

The class *faculty* below is similar to *grad_stud* except that it has a class method *total_faculty/1* which is public and code inheritable.

[3]Note that we are using CORAL [18] like syntax for the aggregation functions and grouping in our rules in this example. The components $\langle \Sigma, <, \Psi, \Pi \rangle$ are formally introduced in section 3.

[4]See definition 3.6 for the formal meanings of code and value inheritance.

```
class faculty
{
    class signatures
    {
        pub, code total_faculty/1;
    }
    instance signatures
    {
        pub, val name/1;
        pub, val eid/1;
        priv, val salary/1;
        pub, code income/1;
        pub, val avg_income/1;
        pub, code meandev/1;
    }
}
(9) faculty:total_faculty(count(<O>)) ← O∈faculty;
(10) faculty:salary(60K);
(11) faculty:income(X) ← salary(X);
(12) faculty:avg_income(avg(<I>)) ← O∈faculty,
      O≪income(I);
(13) faculty:meandev(X) ← income(I),
      avg_income(A), X=abs(I−A);
(14) john∈faculty;
(15) max∈faculty;
(16) john:name("John");
(17) max:salary(75K);
class gta subclass of {grad_stud, faculty}
{
    instance signatures
    {
        priv, val taship/1;
    }
    controls
    {
        reject sig avg_income/1 from faculty;
        reject sig salary/1 from faculty;
        reject sig income/1 from faculty;
    }
}

(18) gta:taship(16K);
(19) gta:income(X) ← stipend(S), taship(T), X=S+T;
(20) sally∈gta;
(21) sue∈gta;
(22) sally:taship(20K);
```

Some features of the class *gta* (graduate teaching assistant), defined above, deserves additional clarification. The clause subclass of declares that *gta* is a subclass of both *grad_stud* and *faculty* and thus inherits properties from both the classes due to multiple inheritance. The *gta* class also introduces a specialized value inheritable public method *taship/1*. It also has a *control* clause and we will defer the discussion on this until section 2.1.2.

Queries:

```
(23) ? sally≪income(X);
(24) ? sue≪income(X);
(25) ? joe≪avg_income(X);
(26) ? joe≪meandev(X);
(27) ? john≪total_faculty(X);
(28) ? faculty≪total_faculty(X);
(29) ? gta≪total_faculty(X);
(30) ? joe≪stipend(X);
(31) ? kelly≪income(X);
(32) ? joe≪income(X);
```
□

In the following sections we discuss some of its modeling features by analyzing answers to few representative queries in Datalog^{++}.

2.1.1 Inheritance

Consider the *message* query (23) which says that send a message to *sally* and get her *income*. The income is returned as a (constant) binding for the variable X. Notice that the method *income/1* is not defined in *sally*. The closest superclass *gta* defines a local method for *income/1* and hence overrides the definitions in class *grad_stud* and *faculty* for which *gta* is a subclass. Now that object *sally* has a unique source for the method *income/1*, it must inherit the method definition from *gta*. The mode of inheritance (value or code inheritance), however, must be determined by inheriting a signature for *income/1*. Since "pub, code income/1" is inheritable in *sally* from *grad_stud*[5], we are inclined to conclude that *income/1* is code inheritable in *sally*. Hence, we have the following code for *income/1* in *sally*: *sally:income(X)* $\leftarrow$ *stipend(S), taship(T), X=S+T*; – the clause for *income/1* is brought down to *sally* with proper adjustments in the context (see Definition 3.6). Finally, to compute income, we must find a *stipend/1* and *taship/1* value for *sally*. While *taship/1* is locally defined through rule (20), *stipend/1* must be inherited. The closest superclass of *gta* that defines *stipend/1* is *grad_stud* as a unique source with a value 12K. Since *stipend/1* is value inheritable in *sally*, the *income* value for *sally* is 32K. Following a similar analysis the reader may verify that the query (24) will produce the answer 28K since in this case *sue* inherits the default *taship/1* value from *gta* and *stipend/1* from *grad_stud*.

In contrast, now consider the query (25). Note that the method *avg_income/1* is value inheritable in *joe* from *grad_stud* hence the value for *avg_income/1* will be inherited in *joe* after the computation of the rule (3) in *grad_stud* giving a value 22.5K. Following a similar analysis, it is easy to see that query (26) produces $X = 7.5K$.

Another interesting query is query (27). While *john* is a *faculty*, the query fails since *total_faculty/1* is a class method. Since *john* is an instance of *faculty*, it can not inherit the method definition. But note that the queries (28) and (29) are valid queries that produces $X = 4$ and $X = 2$ respectively. The reader should be able to see why the same method produces two different values at these two classes. The answer lies in the mode of inheritance – namely code inheritance.

2.1.2 Inheritance Conflict and its Resolution

Notice that *gta* is a subclass of both *grad_stud* and *faculty*. Hence the structure and method definitions local to these classes will be inherited in *gta*. But, *income/1* is defined differently in these two classes and *gta* has a choice. As a result an inheritance conflict occurs. By default, *gta* should have rejected both the definitions and the signatures as well (in our model). But due to the control statement "reject sig income/1 from faculty" in the controls clause of the *gta* class definition, it only rejects the signature of *income/1* from *faculty* leaving a unique source for *income* – the *grad_stud* class. This is called *conflict resolution.* Note that signature rejection implies method rejection but the converse is not true. This follows from our notion of well-typing and consistency requirement on the method inheritance in our data model [10]. Method rejection can be defined using meth keyword in place of the sig keyword.

[5]Because the signature in *faculty* has been rejected by *gta* due to conflict resolution and *income/1* is defined as an instance signature in *grad_stud*.

2.1.3 Encapsulation

Now consider the query (30) or (31). Both the queries fail. The reason for the first query to fail is that *stipend/1* is a private method in *joe* and hence is not accessible from outside. But note that (32) is a legitimate query and produces 15K which is nothing but *joe*'s *stipend* value. This is possible because *income/1* is public in *joe* and internally *joe* is sending a message to itself to access it's *stipend* value to compute *income* without any violation of encapsulation. But notice that (31) fails simply because the object *kelly* is sending a message (in rule (8)) to object *john* to access *john*'s private method *salary/1* and hence causing a violation of encapsulation. Also in rule (3), the subgoal $O \ll income(I)$ succeeds without any violation of encapsulation since *grad_stud* is a superclass of the object O. A logical basis for this approach to encapsulation may be found in [6].

3 Syntax

We now formally introduce the syntax of Datalog^{++}. The language $\mathcal{L}$ of Datalog^{++} is a tuple $\langle \mathcal{P}, \mathcal{F}, \mathcal{V} \rangle$, where $\mathcal{P}$ is an infinite set of predicate symbols with associated arities, $\mathcal{F}$ is an infinite set of function symbols, and $\mathcal{V}$ is an uncountable set of variables. The terms $\mathcal{T}$ of $\mathcal{L}$ are constructed as usual from $\mathcal{F} \cup \mathcal{V}$. Let the ground subset of $\mathcal{T}$ be denoted by $\mathcal{T}^*$.

Class definitions in $\mathcal{L}$ can be generated using the production rules below. In these rules, we use sans serif and italic strings to denote terminal symbols and uppercase strings to denote non-terminals. Furthermore, sans serif strings represent reserved words, and italic strings *term*, *pred*, and *arity* represent respectively user defined identifiers from the set $\mathcal{T}^*$, predicate names from the set $\mathcal{P}$ and a natural number (constants of arity zero) from $\mathcal{F}$.

OBJ := class *term* SUB {DEFN}
SUB := ϵ | subclass of {SUPCLASS}
SUPCLASS := *term* NULL
NULL := ϵ | , SUPCLASS
DEFN := CLASS INST CONT
CLASS := ϵ | class signatures {DEFK}
INST := ϵ | instance signatures {DEFK}
DEFK := DEF VOID
VOID := ϵ | ; DEFK
DEF := PV, VC, *pred/arity*
PV := pub | priv
VC := val | code
CONT := ϵ | controls {RESK}
RESK := RES EMPTY
EMPTY := ϵ | ; RESK
RES := reject SM *pred/arity* from *term*;
SM := sig | meth

The methods and instance definitions are an extended set of Horn formulas. We next define the syntax of the Horn formulas.

3.1 Atomic and Complex Formulas

There are five types of atoms in our language: (global) predicates, local predicates, message predicates, instance is-a, and subclass is-a atoms[6].

[6]We follow the convention of using strings with lower case letters and numbers for ground terms (e.g., *sally*), strings starting with upper case letters for variables (e.g., *Total_income*), and lower case bold italic strings to denote first-order terms (e.g., ***o***) in this paper.

- *Global predicate*: Let p be a predicate symbol of arity k – denoted p/k, and $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k$ be terms. Then $p(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k)$ is a global predicate. A predicate assumes a meaning depending on its position in a rule. Usually a predicate represents a relation except when a predicate is in the rule body and a local predicate is in the rule head. In that case the predicate represents a property (method or attribute) of an object.
- *Local predicate*: Let $p(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k)$ be a predicate and $\boldsymbol{o}$ be a term, then $\boldsymbol{o} : p(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k)$ is a local predicate. Intuitively, a local predicate of the form $\boldsymbol{o} : p(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k)$ means that the predicate $p(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k)$ holds in the object $\boldsymbol{o}$. $\boldsymbol{o}$ is called the *context* or the *descriptor* of the local predicate atom.
- *Message predicate*: Let $p(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k)$ be a predicate and $\boldsymbol{o}$ be a term, then $\boldsymbol{o} \ll p(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k)$ is a message predicate. Intuitively, a message predicate of the form $\boldsymbol{o} \ll p(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k)$ means that evaluate the predicate $p(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k)$ in the object $\boldsymbol{o}$.
- *Is-a*: Let $\boldsymbol{o}_c$, $\boldsymbol{o}_i$ and $\boldsymbol{o}_s$ be terms. Then $\boldsymbol{o}_i \in \boldsymbol{o}_c$ and $\boldsymbol{o}_s :: \boldsymbol{o}_c$ are respectively instance and subclass is-a atoms. Intuitively, they say that $\boldsymbol{o}_i$ and $\boldsymbol{o}_s$ are instance and subclass of $\boldsymbol{o}_c$ respectively. When the difference between the two is unimportant, we will write $\boldsymbol{o}\sharp\boldsymbol{p}$ in the remainder of this paper.

Formulas of $\mathcal{L}$ are defined as usual. A literal is either an atom ($\mathcal{A}$) or the negation of an atom ($\neg\mathcal{A}$). Following the custom in logic programming, we only consider the definite (Horn) clause fragment of our language. This is also for a technical reason. In our set up, a general clause does not make sense since we can not then talk about *locality* and *inheritability* of clauses in the object hierarchy, which we will be explaining shortly.

A clause in $\mathcal{L}$ is an expression of the form $\mathcal{A} \leftarrow \mathcal{B}_1 \ldots, \mathcal{B}_m$ such that $\mathcal{A}$ is any atom in $\mathcal{L}$ except a message predicate, and every $\mathcal{B}_i$, $0 \leq i \leq m$, is any literal except a local predicate. A clause $\mathcal{A} \leftarrow \mathcal{B}_1 \ldots, \mathcal{B}_m$ is called a global, local or is-a clause depending upon whether the atom $\mathcal{A}$ is a predicate, local predicate or an is-a atom respectively.

Furthermore, let $\mathcal{A}$ be a predicate atom – global, local or message predicate. Then the notation $pred(\mathcal{A}) = p/k$ denotes the predicate symbol of $\mathcal{A}$. For any local predicate (or a local clause) $\mathcal{A}$ of the form $\boldsymbol{o} : p(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k)$, the function $context(\mathcal{A}) = \boldsymbol{o}$ returns the object $\boldsymbol{o}$ where the predicate is defined. If $\mathcal{A}$ is a head atom, then we call $\boldsymbol{o}$ as the context of the clause $\mathcal{A} \leftarrow \mathcal{B}_1, \ldots, \mathcal{B}_m$ when $\mathcal{A} = \boldsymbol{o} : p(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k)$. Similarly, for any message predicate of the form $\mathcal{A} = \boldsymbol{o} \ll p(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k)$, $recipient(\mathcal{A}) = \boldsymbol{o}$ returns the target object.

Definition 3.1 (Programs and Queries) A *database* Δ, or equivalently a *program* $\mathbf{P}$, in Datalog^{++} is an expression of the form $\langle \Sigma, <, \Psi, \Pi \rangle$, where Σ, $<$, Ψ, and Π are (possibly empty) sets of class structure, is-a hierarchy, relationship and object property definitions respectively. A program $\mathbf{P}$ is *p-stratified* if every subclass is-a clause in $<$ is message predicate free – i.e., the body literals of the subclass is-a clauses do not contain message predicates. A *query* in $\mathbf{P}$ is a clause of the form $\leftarrow \mathcal{B}_1, \ldots, \mathcal{B}_m$ where the head is empty, and $\mathcal{B}_1, \ldots, \mathcal{B}_m$ are literals. □

For the rest of the paper, we only consider p-stratified programs. For technical reasons, we assume a system object $o^\top$ for every Datalog^{++} program not defined as part of the program universe. But to refer to a global predicate $p(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k)$ in a local clause, we use a message predicate syntax of the form $o^\top \ll p(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k)$ which captures the fact that $p(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k)$ is a global predicate since $o^\top$ is really not an object in the usual sense[7]. Furthermore, for any query $\mathcal{Q} = \leftarrow \mathcal{B}_1, \ldots, \mathcal{B}_m$, $context(\mathcal{Q}) = o^\top$.

To be able to capture inheritance with overriding and conflict resolution, and encapsulation we need to introduce several concepts such as locality and inheritability of clauses and signatures, and inherited clauses and types in our language[8]. We proceed as follows. Herbrand instantiation of programs is defined in a way identical to the classical case. Furthermore, let $\preceq$ be the reflexive transitive closure of $<$.

Definition 3.2 (Locality) Let $\mathbf{P} = \langle \Sigma, <, \Psi, \Pi \rangle$ be a program, $|\mathbf{P}|$ be its Herbrand instantiation, o_c be a class object, o be any object, $ty \equiv <\pi, \varphi, p/k>$ be any signature expression for o_c in Σ, and $cl \equiv \mathcal{A} \leftarrow \mathcal{G}$ be any clause in $|\Pi|$ such that $context(cl) = o$. Then cl and ty are local to object o and o_c in $|\mathbf{P}|$. □

Definition 3.3 (Signature Inheritability) Let S be a set of (ground) is-a atoms, p/k be a predicate symbol, and o be an object. Then the inheritability of the signature of p/k in the object o is defined by the *signature inheritability function* ∇_s as follows:

$$\nabla_s(S, p/k, o) = \begin{cases} o_c & \begin{array}{|l} \text{if signature } p/k \text{ is not local to} \\ o \text{ and } [\exists q \text{ such that } o\sharp q \in S, \\ \nabla_s(S, p/k, q) = o_c, \text{ signature of} \\ p/k \text{ is local to } p \text{ and } (\forall r, \text{ such that} \\ o\sharp r \in S, \text{ one of the following holds.} \\ \quad \bullet\ \nabla_s(S, p/k, r) = r, \text{ and signa-} \\ \quad\ \ \text{ture of } p/k \text{ is not local to } r, \\ \quad \bullet\ \nabla_s(S, p/k, r) = o_c, \text{ or } o \text{ rejects} \\ \quad\ \ \text{signature of } p/k \text{ from } r.)] \end{array} \\ o, & \text{in all other cases.} \end{cases}$$ □

Definition 3.4 (Method Inheritability) Let S be a set of (ground) is-a atoms, p/k be a predicate symbol, and o be an object. Then the inheritability of method p/k in the object o is defined by the *method inheritability function* ∇_m as follows:

$$\nabla_m(S, p/k, o) = \begin{cases} o_s & \begin{array}{|l} \text{if method } p/k \text{ not local to } o \\ \text{and } [\exists q \text{ such that } o\sharp q \in S \\ \nabla_m(S, p/k, q) = o_s, \text{ method } p/k \\ \text{is local to } o_s,\ \nabla_s(S, p/k, o) = \\ \nabla_s(S, p/k, o_s) \text{ and } (\forall r, \text{ such that} \\ o\sharp r \in S, \text{ one of the following holds.} \\ \quad \bullet\ \nabla_m(S, p/k, r) = r, \text{ and} \\ \quad\ \ \text{method } p/k \text{ is not local} \\ \quad\ \ \text{to } r, \\ \quad \bullet\ \nabla_m(S, p/k, r) = o_s, \text{ or } o \\ \quad\ \ \text{rejects method or signature} \\ \quad\ \ p/k \text{ from } r.)] \end{array} \\ o, & \text{in all other cases.} \end{cases}$$ □

[7]Notice that without this, the predicate $p(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k)$ will be treated as a method of the object corresponding to the descriptor of the local clause. Hence, a predicate will become inaccessible from a local method defined in an object.

[8]Similar concepts of rule locality and inheritability were also exploited in [7, 11]. But the way they were exploited in [7, 11] have important conceptual differences with our method in Datalog^{++}.

Definition 3.5 (Inherited Signatures) Let $\mathbf{P}$ be a program, $|\mathbf{P}|$ be its Herbrand instantiation, o be a class object, and $< \pi, \varphi, p/k >$ be a signature expression defined in o. The signature expression $< \pi, \varphi, p/k >$ is inheritable in q if $\nabla_s(S, p/k, q) = o$. We then say that the signature $< \pi, \varphi, p/k >$ holds in q. □

Definition 3.6 (Inherited Clauses) Let $\mathbf{P}$ be a program, $|\mathbf{P}|$ be its Herbrand instantiation, $\hat{<}$ be the ground closure of $<$, $cl \equiv \mathcal{A} \leftarrow \mathcal{G}$ in $|\Pi|$, $context(cl) = o$, and $pred(\mathcal{A}) = p/k$. Then cl is *inherited* in q if the signature $< \pi, \varphi, p/k >$ holds in q (i.e., $\nabla_s(\hat{<}, p/k, q) = r$) and $\nabla_m(\hat{<}, p/k, q) = o$. We say, q *code* inherits cl if $\varphi = code$, otherwise it *value* inherits cl, i.e., $\varphi = val$. □

It is important to observe the difference between *code inheritance* and *value inheritance.* The code inherited clause may be obtained by replacing every occurrence of o in cl by q, i.e., $cl' \equiv (\mathcal{A} \leftarrow \mathcal{G})[o /\!\!/ q]$[9]. This means that if q code inherits cl from o, then $\mathcal{A}[o /\!\!/ q]$ holds in q if $\mathcal{G}[o /\!\!/ q]$ holds in q. In contrast, if q value inherits cl from o, then $\mathcal{A}$ holds in q if $\mathcal{G}$ holds in o, i.e., if $\mathcal{A}$ holds in o.

Example 3.1 Interested readers may verify that the method *income/1* in *gta* is indeed *code inheritable* in *sally,* i.e., $\nabla_s(S, income/1, sally) = grad\_stud$, $\nabla_m(S, income/1, sally) = gta$, S={*gta::grad_stud, gta::faculty, joe*∈*grad_stud, kelly*∈*grad_stud, john*∈*faculty, max*∈*faculty, sally*∈*gta, sue*∈*gta*}, and that *grad_stud* defines the signature <*pub,code,income/1*>. Hence, as anticipated from Definition 3.6, we derive *sally:income(X)← stipend(S), taship(T), X=S+T* from clause (19) in database $\mathbf{D}$ of Example 2.1 as a result of *gta:income(X)← stipend(S), taship(T), X=S+T[gta//sally]*. □

4 Semantics Based on Rewriting

We take the rewriting based approach to give semantics to our language. The primary reasons for this are the followings. Inheritance (value inheritance and code inheritance) and encapsulation in our set up necessitate the use of meta-knowledge not available in the source programs. The traditional fix point operator can not handle such knowledge and the standard proof techniques become inapplicable forcing us to develop new operators and proof techniques. While such techniques are not forthcoming, we are ready to accept the cost associated with rewriting based evaluation techniques for the time being. Although the semantics is not direct, it sheds light on the working principles of the language that can be exploited to develop a direct semantics in future.

To be able to give semantics to Datalog^{++} programs in Datalog$^{\text{neg}}$, we must be able to capture inheritance and encapsulation with all their intricacies and complexities solely by using the machineries in Datalog$^{\text{neg}}$. Hence the goal here is to obtain a purely deductive program that encodes the semantics of the source program but does not lend itself to non-monotonic reasoning, which appears to be the principal obstacle in devising a logical characterization of object-oriented languages. Furthermore, we would like to make sure that the two semantics, i.e., Datalog^{++} and Datalog$^{\text{neg}}$, are identical and intended.

[9]Note that, the term replacement $[o/q]$, in general $[\boldsymbol{p}/\boldsymbol{r}]$, is different from usual definition of substitutions where only variables are replaced by terms (e.g., $\theta = \{X/\boldsymbol{o_s}\}$).

In the following sections we first develop a theoretical basis to reduce inheritance and encapsulation to pure deduction. The techniques we develop are called *i-completion* and *context resolution.* We then develop a suitable *translation function* τ that encodes every i-completed and context resolved Datalog^{++} program into a Datalog$^{\text{neg}}$ program which preserves the intended semantics of every source program. Observe that the reduced program in Datalog$^{\text{neg}}$ is computable but the Datalog^{++} program is not.

4.1 Disassembling Class Structures

Recall that signature definitions in Datalog^{++} are compound non-Horn expressions. To be able to use the signature expressions effectively, we break the structure of the class description, i.e., disassemble the class definitions, as follows.

Definition 4.1 Let $\mathbf{P} = \langle \Sigma, <, \Psi, \Pi \rangle$ be a Datalog^{++} program. Then the disassembled program $\mathbf{P}' = \langle \Sigma', <', \Psi', \Pi' \rangle$ of $\mathbf{P}$ is such that $<' = <$, $\Psi' = \Psi$, and $\Pi' = \Pi$, and Σ' is the smallest set of expressions[10] obtained as follows from every class structure definition o_c in Σ:

- add an expression $\alpha[o_c]$ in Σ',
- for every superclass o_s of o_c, add $\beta[o_c, o_s]$ in Σ',
- for every class signature definition of the form $< \pi, \varphi, p/k >$, add $\sigma_c[o_c, \pi, \varphi, p/k]$ in Σ',
- for every instance signature definition of the form $< \pi, \varphi, p/k >$, add $\sigma_i[o_c, \pi, \varphi, p/k]$ in Σ', and
- for every control definition of the form $<$ *reject* γ p/k *from* $o_a >$, add $\rho[\gamma, p/k, o_c, o_a]$ in Σ'.

4.2 Exposing Clause Locality Through L-closures

The notion of clause locality introduced in section 3 is, however, a meta-knowledge and is not explicitly captured in a Datalog^{++} program. L-closure defined below helps syntactically expose this important piece of knowledge implicitly assumed by every Datalog^{++} programmer.

Definition 4.2 Let $\mathbf{P} = \langle \Sigma, <, \Psi, \Pi \rangle$ be a disassembled Datalog^{++} program. Then the l-closure $\mathbf{P}^* = \langle \Sigma^*, <^*, \Psi^*, \Pi^* \rangle$ of $\mathbf{P}$ is the smallest set of expressions such that

- $\Sigma^* = \Sigma$, $<^* = <$, $\Psi^* = \Psi$ and $\Pi \subseteq \Pi^*$,
- whenever a local clause of the form $\boldsymbol{o} : p(\boldsymbol{t}_1, \ldots, \boldsymbol{t}_k) \leftarrow \mathcal{G} \in \Pi^*$, then also the expression $\lambda[\boldsymbol{o}, p/k] \in \Pi^*$.

The expression of the form $\lambda[\boldsymbol{o}, p/k]$ captures the fact that a method predicate p of arity k is locally defined in object $\boldsymbol{o}$. Once we have the knowledge about the locality in this form and the signature expressions in Σ^*, we can readily determine the *inheritability* of signatures and methods

[10]Observe that the expressions in Σ' are not in the language of Datalog^{++}, nor they are in Datalog$^{\text{neg}}$. The goal here is to recognize them as special expressions which will ultimately be converted to Datalog$^{\text{neg}}$ vocabulary. The same remark applies for the expressions introduced in the next few sections. To distinguish between the Datalog^{++} formulas and these expressions, we will call them reduction expressions since they are introduced during the reduction process. For simplicity, we will call Datalog^{++} formulas and reduction expressions as expressions in the rest of this paper.

in Π^* using the ∇_s and ∇_m functions presented in section 3. Let the expressions of the form $\omega_s[p/k, \boldsymbol{o}, \boldsymbol{q}]$[11] and $\omega_m[p/k, \boldsymbol{o}, \boldsymbol{q}, \varphi]$ denote the fact that the object $\boldsymbol{o}$ inherits from $\boldsymbol{q}$ respectively the signature for p/k, and the method p/k with mode φ. We can now use this explicit expressions to capture inheritance of methods with overriding and conflict resolution.

4.3 Inheritance Based on I-completion

Consider rule (2) in Example 2.1. Since this clause is local to *grad_stud*, it computes an *income* value for the class object *grad_stud*. Recall that the language itself does not have any mechanism to inherit this rule to, say, an instance object *joe* of *grad_stud*. Since we allow the context of a local clause to be a term as well, we may rewrite rule (2) as follows:
(2$'$) V:income(X) $\leftarrow$ stipend(X);

and thus make this rule local to every object in the database which, however, is not our intention. To make it meaningful, we must restrict the instantiation of the variable V to only those objects which must inherit this rule. This is accomplished by adding the inheritability subgoal as follows[12]:
(2$''$) V:income(X) $\leftarrow$ stipend(X), ω_m[income/1,V,grad_stud,code];

That is, rule (2$''$) now applies to those objects V such that V can legitimately code inherit *income/1* from *grad_stud* where it is originally defined. Observe that rule (2$''$) (or the Datalog$^{\text{neg}}$ version of it) is a purely deductive rule with a built-in inheritance mechanism. We will revisit this issue again in section 5 and discuss its elegance with respect to, for example, works proposed in [1, 3, 8]. The informal presentation above is the idea behind the so called *i-completion* which is formalized below.

Definition 4.3 Let $\mathbf{P} = \langle \Sigma, <, \Psi, \Pi \rangle$ be an l-closed program. Then the *i-completion* of $\mathbf{P}$, denoted $\mathbf{P}^i = \langle \Sigma^i, <^i, \Psi^i, \Pi^i \rangle$ is the smallest set of expressions such that

- $\Sigma^i = \Sigma$, $<^i = <$, and $\Psi^i = \Psi$,
- for every clause $Cl \equiv \mathcal{A} \leftarrow \mathcal{G} \in \Pi$, add a clause $Cl' \equiv (\mathcal{A})[\boldsymbol{o}/\!\!/V] \leftarrow (\mathcal{G})[\boldsymbol{o}/\!\!/V], \omega_m[p/k, V, \boldsymbol{o}, code] \in \Pi^i$ such that the *context*$(Cl) = \boldsymbol{o}$, *pred*$(\mathcal{A}) = p/k$ and V is a distinct variable not occurring in Cl.

Notice that i-completion can only handle code inheritance, hence value inheritance, which is a much simpler problem, must be handled separately. There are two ways to address this issue – (i) by rewriting every rule in Π, or (ii) by adding an axiom to Π^i. For example, to value inherit *avg_income/1* in *joe* from *grad_stud* (rule (3)), we may write
(3$'$) V:avg_income(X) $\leftarrow$ grad_stud$\ll$avg_income(X),
ω_m[avg_income/1,V,grad_stud,val];

This rewriting for value inheritance, as shown above in rule (3$'$), is costlier than our alternative approach. Observe specially the rewriting of the *avg_income/1* subgoal in rule (3$'$) above that encodes the meaning of value inheritance vis a vis Definition 3.6. We will discuss the second approach to capture this phenomenon later when we discuss the translation algorithm to Datalog$^{\text{neg}}$ because it is possible to axiomatize value inheritance. Observe a subtle issue here. Since the signature definitions are independent of the method definitions, and the signatures may be inherited from any arbitrary superclass (other than the class from where the method definition is being inherited), seemingly there is no easy way to determine whether a local clause is to be value inherited or code inherited. Consequently, we are forced to do both for a single rule. However, since inheritability is a total function and it does not differentiate between the modes, all we know is that the definition corresponding to a method, say, p/k is inheritable. Hence only one of the two rewritten rules will become active at run time.

4.4 Encapsulation Through Context Resolution

Encapsulation is an issue that did not receive proper attention from the logic programming community so far except perhaps in the works by Miller [17], Kifer et al. [12], and by Bugliesi and Jamil [6]. While Miller addresses this issue in a logical way, his semantics is too restrictive and hand coded, and actually is not in the spirit of object-oriented logics. Kifer's approach to encapsulation is meta-logical and does not tackle the issue in a computationally efficient way. The language proposed in [6] addresses the issue in a more direct and intuitive way and provides a basis for a logical analysis of encapsulation in logic based object-oriented languages. While the setting is restricted in [6], we exploit the basic theory and fully extend the idea in our present paper.

Let the expression $\eta[\boldsymbol{o}_s, \boldsymbol{o}_r \ll p(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k)]$ denote the fact that object $\boldsymbol{o}_s$ (sender) is sending a message to object $\boldsymbol{o}_r$ (receiver) to evaluate the method $p(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k)$ in $\boldsymbol{o}_r$. $\boldsymbol{o}_r$ will respond to this message only if p/k is accessible in $\boldsymbol{o}_r$ for $\boldsymbol{o}_s$ (the rules for method accessibility may be found in [10]). The accessibility function presented below formalizes these rules.

Definition 4.4 (Method Accessibility) Let S be a set of (ground) signature expressions, I be a set of (ground) is-a atoms, p/k be a predicate symbol, o_s and o_r be two object symbols. Then the accessibility of method p/k in the object o_r with respect to o_s is defined by the *context resolution function* Υ as follows:

$$\Upsilon(S, p/k, o_s, o_r) = \begin{cases} true & \text{if one of the following conditions holds:} \\ & \bullet\ o_s = o_r, \\ & \bullet\ o_s \neq o_r, \text{ and } o_r \preceq o_s \text{ holds, or} \\ & \bullet\ o_s \neq o_r, \pi = pub, \text{ and } ((\sigma_c[o_c, \pi, \varphi, p/k] \in S \text{ and } \nabla_s(I, p/k, o_r) = o_c) \\ & \quad \text{or } (\sigma_i[o_c, \pi, \varphi, p/k] \in S \text{ and } \nabla_s(I, p/k, o_r) = o_c)) \text{ holds} \\ false, & \text{in all other cases.} \end{cases}$$ □

Let the expression $\varrho[p/k, \boldsymbol{o}_r, \boldsymbol{o}_s]$ state the fact that p/k is accessible in object $\boldsymbol{o}_r$ by object $\boldsymbol{o}_s$. Consider now rule (8) in Example 2.1. We first transform rule (8) as follows to capture the context of the message call.
(8$'$) kelly:income(X) $\leftarrow$ η[kelly,john$\ll$salary(X)];

[11]We, however, do not make use of the expression $\omega_s[p/k, \boldsymbol{o}, \boldsymbol{q}]$ in this paper for i-completion based signature inheritance. The reason for this is that we do not yet allow signature definition using clauses in Datalog^{++}.

[12]Contrast this approach to the negation based approach, for example in [1]. We present a discussion on their differences in section 5.

Then we finally transform the rule as follows to resolve the context.

(8″) kelly:income(X) $\leftarrow$ john$\ll$salary(X), ϱ[salary/1,john,kelly];

Now, the expression ϱ[salary/1,john,kelly] will evaluate to true only if *salary/1* is public or *kelly* is a superclass of *john*. From the database description, we know that none of them are true, and hence the implication fails as expected. The following definition formalizes the intuitive observations above.

Definition 4.5 Let $\mathbf{P} = \langle \Sigma, <, \Psi, \Pi \rangle$ be an i-completed Datalog^{++} program. Then the *context resolution* of **P**, denoted $\mathbf{P}^c = \langle \Sigma^c, <^c, \Psi^c, \Pi^c \rangle$ is the smallest set of expressions such that

- $\Sigma^c = \Sigma$, and $<^c = <$,
- for every global clause $Cl \equiv \mathcal{A} \leftarrow \mathcal{G} \in \Psi$, add Cl' to Ψ^c such that $Cl' \equiv o^\top : \mathcal{A} \leftarrow \mathcal{G}$,
- for every query clause $Cl \equiv\ \leftarrow \mathcal{B}_1, \ldots, \mathcal{B}_n \in \Psi$, add Cl' to Ψ^c obtained from Cl as follows:
  - $Cl = Cl'$,
  - for every $\mathcal{B}_i \in Cl'$, such that $0 \leq i \leq n$ and $\mathcal{B}_i$ is a message predicate, $pred(\mathcal{B}_i) = p/k$, $recipient(\mathcal{B}_i) = \boldsymbol{o}_r$, add $\varrho[p/k, \boldsymbol{o}_r, o^\top]$ as $\mathcal{B}_{n+1}$ in Cl',
- for every local clause $Cl \equiv \mathcal{A} \leftarrow \mathcal{B}_1, \ldots, \mathcal{B}_n \in \Pi$, add Cl' to Π^c obtained from Cl as follows:
  - $Cl = Cl'$,
  - for every $\mathcal{B}_i \in Cl'$, such that $0 \leq i \leq n$ and $context(Cl) = \boldsymbol{o}_s$ do the following:
    - * if $\mathcal{B}_i$ is a message predicate add $\varrho[p/k, \boldsymbol{o}_r, \boldsymbol{o}_s]$ as $\mathcal{B}_{n+1}$ in Cl' where $context(Cl) = \boldsymbol{o}_s$, $recipient(\mathcal{B}_i) = \boldsymbol{o}_r$, $pred(\mathcal{B}_i) = p/k$ and $\boldsymbol{o}_s \neq \boldsymbol{o}_r$.
    - * if $\mathcal{B}_i$ is a (self) predicate then replace $\mathcal{B}_i$ as $\boldsymbol{o}_s \ll \mathcal{B}_i$.

Notice that an accessibility expression is added only when it is necessary to do so – that is the addition is avoided when the terms representing the sender and receiver objects are identical, implying a self invocation. Also notice that the global clauses are assigned to the system object (doing so does not disrupt the semantics of the programs) and queries are evaluated in the context of system object $o^\top$ the reason for which is described in [6] and should be easy to see.

4.5 Datalog$^{\text{neg}}$ Rewriting of Datalog^{++} Programs

We are now ready to define an algorithm to reduce every Datalog^{++} program to Datalog$^{\text{neg}}$. This requires us to develop a translation function τ as stipulated in Definition 4.6 that will map every Datalog^{++} expression to Datalog$^{\text{neg}}$ expressions. We proceed as follows.

Given any Datalog^{++} expression ϕ, its encoding into Datalog$^{\text{neg}}$, denoted ϕ^*, is given by the following recursive transformation rules. In the following, τ is an identity function on terms and symbols in Datalog^{++}.

- Encoding of complex formulas:
  - $\tau(\mathcal{A} \leftarrow \mathcal{B}_1, \ldots, \mathcal{B}_m) = \tau(\mathcal{A}) \leftarrow \tau(\mathcal{B}_1), \ldots, \tau(\mathcal{B}_m)$
- Encoding of atomic Datalog^{++} formulas (given case by case):
  - $\tau(p(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k)) = \mathsf{rel}(p, arg(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k))$[13].
  - $\tau(o^\top \ll p(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k)) = \mathsf{rel}(p, arg(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k))$.
  - $\tau(o^\top : p(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k)) = \mathsf{rel}(p, arg(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k))$.
  - $\tau(\boldsymbol{o}{:}p(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k)) = \mathsf{meth}(\boldsymbol{o}, p, k, arg(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k))$ when $\boldsymbol{o} \neq o^\top$.
  - $\tau(\boldsymbol{o} \ll p(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k)) = \mathsf{meth}(\boldsymbol{o}, p, k, arg(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k))$ when $\boldsymbol{o} \neq o^\top$.
  - $\tau(\boldsymbol{o} \in \boldsymbol{q}) = \mathsf{ins}(\boldsymbol{o}, \boldsymbol{q})$.
  - $\tau(\boldsymbol{o} :: \boldsymbol{q}) = \mathsf{sub}(\boldsymbol{o}, \boldsymbol{q})$.
- Encoding of reduction expressions (given case by case):
  - $\tau(\alpha[o_c]) = \mathsf{class}(o_c)$.
  - $\tau(\beta[o_c, o_s]) = \mathsf{sub}(o_c, o_s)$.
  - $\tau(\sigma_c[o_c, \pi, \varphi, p/k]) = \mathsf{sig}(o_c, \pi, \varphi, p, k, class)$.
  - $\tau(\sigma_i[o_c, \pi, \varphi, p/k]) = \mathsf{sig}(o_c, \pi, \varphi, p, k, ins)$.
  - $\tau(\rho[\gamma, p/k, o_c, o_a]) = \mathsf{rej}(\gamma, p, k, o_c, o_a)$.
  - $\tau(\lambda[\boldsymbol{o}, p/k]) = \mathsf{loc}(\boldsymbol{o}, p, k)$.
  - $\tau(\omega_m[p/k, \boldsymbol{o}, \boldsymbol{q}, \varphi]) = \mathsf{meth\_inh}(p, k, \boldsymbol{o}, \boldsymbol{q}, \varphi)$.
  - $\tau(\varrho[p/k, \boldsymbol{o}_r, \boldsymbol{o}_s]) = \mathsf{vis}(p, k, \boldsymbol{o}_r, \boldsymbol{o}_s)$.

4.6 Reduction

Finally, we have the following definition for inheritance and encapsulation *reduced* first-order Datalog^{++} programs. Let the reduction expressions denote the set of meta-expressions introduced in a Datalog^{++} program during, disassembling, l-closure, i-completion and context resolution.

Definition 4.6 Let **P** be a Datalog^{++} program, and τ be a translation function for every expressions in **P** Let P_{∇_s}, P_{∇_m}, and P_{Υ} be Datalog$^{\text{neg}}$ programs that implement respectively the ∇_s, ∇_m, and Υ functions. Also let P_{isa} and P_{val} be Datalog$^{\text{neg}}$ programs for computing the reflexive transitive closure $\preceq$ from $<$ and the value inheritance axiom respectively. If $\mathbf{P}^\downarrow$ is a disassembled, l-closed, i-completed and context resolved program of **P** then $\mathbf{P}_r \equiv \tau(\mathbf{P}^\downarrow) \cup P_{\nabla_s} \cup P_{\nabla_m} \cup P_{\Upsilon} \cup P_{isa} \cup P_{val}$ is the inheritance and encapsulation *reduced* Datalog$^{\text{neg}}$ program of the Datalog^{++} program **P**.

The reduction algorithm can be given as follows as suggested by Definition 4.6. Note that the programs P_{∇_s}, P_{∇_m}, P_{Υ}, P_{isa} and P_{val} are already in Datalog$^{\text{neg}}$.

```
Input: A Datalog++ program P.
Output: A reduced program P_r of program P
    in Datalog^neg.
begin
    - $\mathbf{P}' =$ Apply disassembling to program $\mathbf{P}$.
    - $\mathbf{P}^* =$ Apply l-closure to program $\mathbf{P}'$.
    - $\mathbf{P}^i =$ Apply i-completion to program $\mathbf{P}^*$.
    - $\mathbf{P}^c =$ Apply context resolution to program $\mathbf{P}^i$.
    - $\mathbf{P}_r = \tau(\mathbf{P}^c) \cup P_{\nabla_s} \cup P_{\nabla_m} \cup P_{\Upsilon} \cup P_{isa} \cup P_{val}$.
end.
```

Note that, so long **P** remains a definite program, the reduction $\mathbf{P}_r$ is always stratified. The inheritability axioms we introduce as part of the implementation for P_{∇_s} and P_{∇_m} contain negative literals and thus necessitate Datalog$^{\text{neg}}$. However, the rules in P_{∇_s} and P_{∇_m} are locally stratified as shown in [10].

[13]Note that $\tau(p(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k)) = p(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k)$ is also possible.

5 Related Research and Implementation Issues

It is easy to see that a one pass compiler for Datalog^{++} can be developed which can reduce the program in a single scan. This is possible because of the techniques presented in this paper. Recall that none of the techniques presented, for example disassembling, l-closure, i-completion and context resolution, require to inspect an expression other than which is being reduced. This is not true for most other proposals including [1, 3, 8]. In their cases, to disallow application of clauses, at least a hierarchy computation is essential. If the hierarchy depend on inherited properties, then a rewriting is not really possible. We do not have such serious limitations. In fact, every and any Datalog^{++} database is Datalog^{neg} reducible. We now discuss some of the implementation and system related issues that influenced our design in comparison to few representative proposals.

5.1 Compile Time Inheritability

It is possible to compute the inheritability expressions at compile time. This is possible only if the class hierarchy is static – no rules of the form $\boldsymbol{o} :: \boldsymbol{p} \leftarrow \mathcal{G}$ exists in **P** such that $\mathcal{G}$ contains any message atom. A dynamic computation of inheritability is always possible. Only difference is that, we may have to accept multiple minimal models as opposed to a least model. Note that, for every reduced Datalog^{++} program, we currently have a least model. Observe that by design, our class hierarchy is static while we allow instance membership to be dynamic (rules of the form $\boldsymbol{o} \in \boldsymbol{p} \leftarrow \mathcal{G}$) since it does not affect the inheritability. The programs P_{∇_s} and P_{∇_m} added during reduction computes inheritability of signatures and methods at run time which involves (only) negation computation and thus slowing down the execution of queries. It appears that if we accept programs with static class hierarchies, we can avoid negation computation all together by procedurally computing inheritability at compile time and improve performance. Another advantage of a compile time inheritability computation is that we can now throw away most of the reduction expressions and make the target program more neat and compact.

5.2 Efficiency of Static Overriding

We took the so called static overriding [1] approach to inheritance[14]. While it is possible to adopt dynamic inheritance a la Datalog^{meth} [1] and F-logic [12], we think it has serious practical drawbacks. For example, consider the following code fragment adapted from [1].

r_1 - employee:socins(Y) $\leftarrow$ salary(Z), Y=0.1*Z;
r_2 - wstudent:socins(50) $\leftarrow$ dept$\ll$paystax(wstudent,X), salary(S), S$\leq$500;
r_3 - wstudent::employee;

In this case, if we are to verify the applicability of r_2 in *wstudent*, we must also compute the subgoal *dept$\ll$paystax(wstudent,X)* which in turn may require us to compute another message subgoal in a chain reaction fashion – virtually forcing us to compute a huge portion of the database only to discover later that *salary(S), S$\leq$500* is false or *dept$\ll$paystax(wstudent,X)* is false. In contrast we believe that overriding based on definition makes better sense from a practical point of view. We regard the above program as ill-conceived and a case of poorly defined method, i.e., if we have to apply r_1 in *wstudent* if r_2 should fail. In our framework we would include the following rule to alleviate the problem and to complete the definition for *socins/1* in *wstudent.*

r_4 - wstudent:socins(Y) $\leftarrow$ salary(Z), Z>500, Y=0.1*Z;

As another example, consider the following database.

r_1 - employee:income(60K);
r_2 - wstudent:income(15K);
r_3 - wstudent::employee;

In [1], *wstudents* will inherit both 60K and 15K if static inheritance is adopted which may not make sense for many applications. Similar remark applies for [8, 2]. Dynamic inheritance also does not come to rescue us from this unwanted situation. But in our case, we will inherit neither considering it as an inheritance conflict and can emit an error message if desired. We also are able to choose one of the inheritance if it seems appropriate. This is possible because of our inheritability function and i-completion. Static overriding in OOLP+ [8] is even more restrictive. In OOLP+, users have to override methods using keywords in the class definition practically making it hand coded and defeating the spirit of inheritance to a large extent.

5.3 Completion Based Rewriting

It is easy to notice the superiority of our completion based rewriting of local clauses as opposed to negation based blocking of rule application in [1] or the Prolog cut operator in [8]. In our case, we add an inheritability expression of the form $\omega_m[p/k, V, o, code]$ to every local clause with appropriate term replacement which does not require inspection of any other rule. Also the expression $\omega_m[p/k, V, o, code]$ is computable either statically at compile time or dynamically at run time. But in the case of [1], for every class c that has a local definition, we are forced to add a subgoal of the form $\neg c(X)$ to the rule. In fact, this forces us to hand code the inheritance for every class defeating the purpose of declarative programming. Besides, if all the classes, at the worst, overrides a method, we will have to include each one of the classes as $\neg c(X)$ in the rule.

Furthermore, the approach in [1] assumes that only classes may override a method (or state variable) but not instances. This is a quite restricted view of the databases. Consider, for example, a football player database where we would like to define that all instances of a player class have a default height of 6 feet while the individual instances are allowed to override this value with their own. This very common phenomenon can not be captured in [1] while can very easily be captured in ours. While with proper tuning it is possible to capture this in [1], it readily becomes awkward and computationally infeasible.

5.4 Encapsulation and Context Resolution

Probably for the first time, we have introduced a sound and effective semantics for encapsulation in deductive systems with inheritance. A theoretical basis for our approach may be found in [6]. The rewriting based on context resolution and the visibility function Υ together form the basis for encapsulation in Datalog^{++}. None of the works, to our knowledge, so far addressed this issue. Notice that the function Υ can not be computed at compile time since it

[14]Recall that the database systems such as O_2 [15], Orion [13], and Gemstone [16], and programming languages such as C^{++} [19], and Smalltalk [4] take our approach.

depends on inheritance and involves network of visibility relationships.

6 Conclusion and Future Works

In this paper, we presented a language called the Datalog^{++}. The semantics of Datalog^{++} is given in Datalog^{neg} by reducing inheritance and encapsulation to pure deduction. Several meta-logical constructs enriched the features of our language and the reduction was necessary to develop a translational semantics of these features in the language. We also provided several computable functions to compute the so called inheritability and visibility of methods, and demonstrated that our completion based technique allows flexible modeling of the applications and supports an object-oriented perception of the world.

We believe that the semantics developed in this paper has a far reaching influence on the design of the deductive object-oriented languages. It shows that certain features that are believed to be difficult to address logically, can indeed be captured logically. Since a logic program (the reduced Datalog^{neg} program) can be developed to model object-oriented features (as demonstrated in this work) in an indirect way, perhaps the working principle of these features can be studied and isolated, and given a full logical characterization. We thus believe that a direct semantics for every feature we introduced in this paper can be developed.

Our work can be extended in several different ways. It seems appropriate for some applications to have a choice to override methods dynamically or statically based on the need of the application. For some applications, a method may call for monotonic inheritance and not override at all. We think, giving such a choice to application designers will result in a flexible design environment. While Datalog^{neg} and CORAL has built-in optimization mechanisms, specific Datalog^{++} optimization techniques may be possible that utilizes knowledge specific to object-oriented paradigm. Finally, update can be accommodated as an orthogonal feature as it was addressed in [3]. These are some of the issues we plan to investigate in our future research.

Acknowledgement: This research was supported in part by the Macquarie University Research Grant. The author would like to thank Mohammad Ashrafuzzaman of the University of Saskatchewan, Canada for reading the initial version of this paper and giving helpful comments.

References

[1] S. Abiteboul, G. Lausen, H. Uphoff, and E. Waller. Methods and Rules. In *ACM SIGMOD Conference on Management of Data*, pages 32–41, 1993.

[2] F. Belli, O. Jack, and L. Naish. Object-oriented programming in Prolog: Rationale and a case study. Technical Report 92/2, Department of Electrical and Electronics Engineering, University of Paderborn, Paderborn, Germany, 1992.

[3] E. Bertino and D. Montesi. Towards a logical object-oriented programming language for databases. In A Pirotte, C. Delobel, and G. Gottlob, editors, *Proc. of the 3rd Intl. Conf. on EDBT*, pages 168–183. Springer-Verlag, 1992. LNCS 580.

[4] A. H. Borning and D. H. Ingalls. A type declaration and inference system for Smalltalk. In *Proc. of the ACM Symposium on Principles of Programming Languages*, pages 133–141, 1982.

[5] M. Bugliesi. A declarative view of inheritance in logic programming. In K. Apt, editor, *Proc. Joint Int. Conference and Symposium on Logic Programming*, pages 113–130. The MIT Press, 1992.

[6] M. Bugliesi and H. M. Jamil. A logic for encapsulation in object oriented languages. In M. Hermenegildo and J. Penjam, editors, *Proceedings of the 6th International Symposium on Programming Language Implementation and Logic Programming (PLILP)*, pages 213–229, Madrid, Spain, 1994. Springer-Verlag. LNCS 844.

[7] M. Bugliesi and H. M. Jamil. A stable model semantics for behavioral inheritance in deductive object oriented languages. In G. Gottlob and M. Y. Vardi, editors, *Proceedings of the 5th International Conference on Database Theory (ICDT)*, pages 222–237, Prague, Czech Republic, 1995. Springer-Verlag. LNCS 893.

[8] M. Dalal and D. Gangopadhyay. OOLP: A translation approach to object-oriented logic programming. In *Proceedings of the First DOOD Conference*, pages 593–606, 1990.

[9] H. M. Jamil. Architecture and implementation of the Visual Datalog^{++} system. Technical report, Department of Computing, Macquarie University, Sydney, Australia, February 1997. Submitted for publication.

[10] H. M. Jamil. Inheritance with overriding without non-monotonic reasoning in Datalog^{++}. In *Proceedings of the 5th International ICLP Workshop on Deductive Databases and Logic Programming (DDLP)*, Leuven, Belgium, July 1997.

[11] H. M. Jamil and L. V. S. Lakshmanan. A declarative semantics for behavioral inheritance and conflict resolution. In John Lloyd, editor, *Proceedings of the 12th International Logic Programming Symposium*, pages 130–144, Portland, Oregon, December 1995. MIT Press.

[12] M. Kifer, G. Lausen, and J. Wu. Logical Foundations for Object-Oriented and Frame-Based Languages. *Journal of the Association of Computing Machinery*, 42(3):741–843, July 1995.

[13] W. Kim. A model of queries for object-oriented databases. Technical Report ACA-ST-365-88, MCC, 1988.

[14] M. J. Lawley. A Prolog interpreter for F-logic. Unpublished Manuscript. Griffith University, Brisbane, Australia, 1993.

[15] C. Lecluse, P. Richard, and F. Velez. *O_2, An Object-Oriented Data Model.* ACM Press, 1987.

[16] D. Maier and J. Stein. Development and implementation of object-oriented DBMS. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 355–392, Cambridge, MA, 1987. MIT Press.

[17] D. Miller. A Logical Analysis of Modules in Logic Programming. *Journal of Logic Programming*, 6(1/2):79–108, January/March 1989.

[18] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL : Control, Relations and Logic. In *Proc. of 18th VLDB Conference*, pages 238-250, 1992.

[19] B. Stroustroup. *The C^{++} Programming Language.* Addison-Wesley, 1986.