An Overview of the $\mathcal{OLI}$ Multiparadigm Programming Language and Its Semantics

J.H.M. Lee
Department of Computer Science & Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong
jlee@cs.cuhk.edu.hk

P.K.C. Pun
Department of Computer Science
University of Manchester
Manchester M13 9PL, United Kingdom
ppun@cs.man.ac.uk

Abstract

Lee and Pun outline the $\mathcal{OLI}$ language as an accompaniment of the $\mathcal{OLI}$ multiparadigm programming design methodology. The $\mathcal{OLI}$ language integrates the object-oriented and the logic programming paradigms with a precise and well-balanced interface so that each paradigm shares an equal and cooperating partnership in problem analysis and problem solving. An important property of $\mathcal{OLI}$ is that programmers can program either in one of the paradigms alone or in a mixed paradigm without sacrificing expressiveness and efficiency. In this paper, we give a formal definition of the $\mathcal{OLI}$ language and study its semantics both from the logical perspective and the object-oriented perspective. By viewing objects as an enrichment of the Herbrand universe, we define the declarative and operational semantics of $\mathcal{OLI}$. We show that $\mathcal{OLI}$'s operational semantics, a generalization of SLD-resolution, is sound and complete. From the object-oriented point of view, the logic part of $\mathcal{OLI}$ is simply an object with logic programs as states and methods for performing logical deduction.

KEY WORDS: *Multiparadigm Programming, Logic Programming, Object-Oriented Programming*

1 Introduction

A programming paradigm is a coherent conceptual framework which enables programmers to look at a domain of interest and analyze problems in this domain. Each programming paradigm offers a set of programming languages as representation facilities so that problem solutions can be expressed and executed on computing machines effectively. Appropriate programming paradigm for a problem domain has great impact on the programming task and the productivity of programmers. A problem of a particular domain can be solved easily using languages in one paradigm, and yet the same problem may require hours of tedious programming using languages in another paradigm. Different paradigms also facilitate different levels of programming abstraction and software design methodologies. In developing large software, we normally encounter many tasks of different domains and very often no single paradigm would be suitable for all these tasks. A multiparadigm programming approach, which support combinations of different paradigms, provides a natural mental model for the construction of complex computer applications. In this paper, we are concerned with the integration of logic programming and object-orientation.

Lee and Pun [12] put forward the $\mathcal{OLI}$ multiparadigm programming design methodology and outline the accompanying $\mathcal{OLI}$ language. The $\mathcal{OLI}$ design methodology consists of a set of sufficient guidelines to assist system designers to characterize the structural and relational characteristics of a complex software system, while the $\mathcal{OLI}$ language provides a language vehicle for natural and easy realization of the design. The major difference between the $\mathcal{OLI}$ scheme and other integration efforts is that the $\mathcal{OLI}$ scheme is based on the concept of maintaining an equal partnership between the two paradigms rather than the concept of enhancing any one paradigm with constructs of the other. An important consequence is that each paradigm in $\mathcal{OLI}$ performs tasks only of the paradigm's natural strength: object for modeling dynamic state changes and logic for modeling relationships in a static world.

An important property of $\mathcal{OLI}$ is that it is, besides a multiparadigm language, also a logic programming language and an object-oriented language. Thus, its semantics can be studied from two perspectives: the object-oriented view and the logical view. In this paper, we present the formal syntax and semantics of the $\mathcal{OLI}$ language. The declarative semantics of $\mathcal{OLI}$ is based on viewing objects as enrichment of the Herbrand universe. A generalization of SLD-resolution is defined to cope with the new Herbrand terms. We show that the generalized resolution principle is both sound and complete. The object-oriented semantics of $\mathcal{OLI}$ is based on the notion of a logical object, which has as states a logic program and has methods for invoking logical deduction.

The rest of the paper is organized as follows. Section 2 reviews related work in multiparadigm programming. In section 3, we use a motivating example to illustrate the need of a multiparadigm language and give an overview of the language constructs of $\mathcal{OLI}$, followed by a description of the $\mathcal{OLI}$ syntax in section 4. The declarative and operational semantics of $\mathcal{OLI}$ and its soundness and completeness results are given in section 5. Section 6 gives $\mathcal{OLI}$'s informal object-oriented semantics. Section 7 summarizes our contribution and sheds light on future direction of research.

2 Related Work

In general, there are two camps in integrating the object-oriented and the logic programming paradigms: logical emulation of objects and object-oriented emulation of predicates. The following review is not intended to be exhaustive; it is limited to what is considered representative. A more detailed treatment of this topic is reported in other literature [6, 4].

In the first camp, objects are either emulated as unconditional clauses or perpetual processes. Languages and systems [19, 11, 22, 8] of this camp force a programmer to subscribe to the clausal view or the process view which do not necessarily agree with the object model [5] as subscribed by the object-oriented programming paradigm in programming design. In addition, extra-logical constructs are employed to model state changes. More recently, a number of vanilla preprocessors, *e.g.* *L&O* [14] and PROLOG++ [17] have been developed to reinstate the object model of object-oriented design. However, problems with the semantics of the translated programs still remain. Moreover, the motivation of *L&O* and PROLOG++ is to extend the logic programming paradigm with features from the object-oriented paradigm in order to make the design and development of large logic programs manageable. Such a motivation does not advocate an equal and co-operating partnership between the object-oriented and the logic programming paradigms, which is a more important objective for their integration.

In the second camp, class methods are allowed to be defined as a set of clauses. Languages and systems [10, 15, 21, 18] of this camp require logical predicates to be specified within classes, *i.e.* predicates do not exist independently. In general, languages and systems of these two camps are extensions of a native programming paradigm by emulating programming constructs of the other paradigm as an enhancement of the respective native paradigm. These multiparadigm languages and systems do not provide an equal and cooperating partnership between the object-oriented and the logic programming paradigms in both problem analysis and problem solving. Moreover, these emulation strategies always result in an interface with little or no semantics. The interface is usually not well-balanced to support communication among constructs of these two paradigms.

The LIFE language [3] subscribes to the logic programming, functional programming, and object-oriented programming paradigms. The language design of LIFE is most similar to the philosophy of our approach. The central component of LIFE is a calculus of ψ-terms [1, 2], an extension of first order terms (Herbrand terms and λ-terms) which provides structural convenience for representing record-like data structures. The ψ-calculus defines a clear notion of subsumption interpretable as a subtype ordering, together with an efficient unification operation interpretable as type intersection. Thus ψ-terms account for a form of structured type inheritance typically found in object-oriented languages. The ψ-calculus, however, does not address dynamic state changes and only accounts for a limited set of object-oriented features in terms of type inheritance.

3 An Overview of $\mathcal{OLI}$

Before we expose the formal syntax and semantics of $\mathcal{OLI}$ in subsequent sections, we use an example scenario to motivate the need of a multiparadigm programming language combining the power of logic and object-oriented programming. Program fragments are used to illustrate the language constructs of the $\mathcal{OLI}$ language. An $\mathcal{OLI}$ program consists of two parts: the *object* part which contains object class definitions (*a la* SMALLTALK) and the *logic* part which contains predicate definitions (*a la* PROLOG). We assume readers basic knowledge of PROLOG and SMALLTALK. Interested readers are referred to [20, 7] for detailed description of the languages.

3.1 A Multiparadigm Scenario

A university is an organization made up of different personnel. Each member of the university personnel has a name and other details such as age, sex, address, *etc*. The members of university personnel relevant in our scenario are lecturers, students and the registrar. Each lecturer is responsible for teaching certain university courses. Each student is identified by a student identification number; he/she is currently studying in a particular year and belonging to a particular major subject stream. The university registrar is responsible for managing information regarding course registration. The university offers a number of different courses in which students can choose to enroll according to the previous courses they have taken. Each university course is identified by a course code; it has such details as course contents and the relevant major subject stream it belongs to. A university registration system is an information system which maintains information of different courses offered by the university and manages the ways these courses are related to the university personnel. In order to prevent students from abusing this system, an access quota is imposed on students in order to monitor their usage of the system.

Object-oriented programming can model university personnels and courses as object classes. Members of university personnels can be further refined into lecturer, student and registrar object classes using superclass/subclass relationships. Structural aspects, such as age and name of a personnel, are identified and represented as state variables of each class. However, relationships among objects, such as the different ways university courses are related to the university personnel, have to be simulated by the use of instance variables and methods which refer to other objects. Such an approach imposes an unidirectional application restriction to relationships which are multi-directional in nature.

By using logic programming, the principle components of the university registration system are rules and facts about the university personnel and university courses. Since relational abstraction is declarative and multi-directional in nature, the same set of facts and rules can be reused in different contexts, *e.g.* the same piece of fact can be used to assert that a student has taken certain courses or to find out all courses taken by him in the past. However, the only way for logic programming languages to describe a data type is to give an abstract, algebraic description of the properties and its operations as a set of facts and related rules. In this way, the structural aspects in the university registration system, *e.g.* name, age, access quota of a student, are represented in an abstract manner which makes them more difficult to understand than they should be. Moreover, update of these object structures, *e.g.* setting and resetting access quota of students, always involves using constructs of destructive effect or complicated semantics. In this way, it seems unnatural to represent history sensitive structures in the logic programming paradigm.

Each paradigm alone does not suffice to model the university registration system adequately. In the following, we describe a multiparadigm solution using the $\mathcal{OLI}$ language.

3.2 The Object Part

Class definitions in $\mathcal{OLI}$ resembles that of SMALLTALK, which consists of state variable declarations, and class and instance method definitions. We differentiate between *variant* and *invariant* methods. The former creates new objects and/or changes states of some existing objects by side-effects; while the latter is used for either state enquiry or function abstraction.

The object part of the university registration system contains definition of the `Personnel`, `Lecturer`, `Student`, `Registrar`, and the `Course` classes. Figure 1 shows code fragments of the `Registrar` class definition. The `Registrar` class is a subclass of `Personnel`. `Registrar` has as instance variable `salary`, in addition to `name`, `age`, and `sex` inherited from `Personnel`. `Registrar` has one variant class method for creating new instances of the class and has no invariant class methods. A `Registrar` object has variant instance methods for adding and dropping courses for a student, advising a student on the kind of courses that (s)he can take, obtaining the class list of a course and the graduation list of a major stream. We examine the methods `add:course:` and `advise:` in more details.

The `add:course:` method takes as arguments a student and a course. The method starts by checking if the student has used up his/her CPU quota for accessing the registration system. If not, the method decrements the CPU quota of the student and inquires the logic part of the university registration program to check if the student is eligible to take the course and the current courses that the student is taking. This is achieved by sending a `query:` message to the built-in object `$logicBase`, which is the interface to the logic part of the program. If the student can take the

```
Personnel subclass: #Registrar
  InstanceVariableNames: 'salary'

Variant Class Methods:
name: aName age: anAge sex: aSex salary: aNumber
  "Create an instance of myself with age, sex, and salary."
  ...

Variant Instance Methods:
add: aStud course: aCourse
  "Add a course for a student."
  (aStud cpuQuota > 0) ifTrue:
    [aStud deductCpuQuota.
     ($logicBase query: [canTake(aStud,aCourse),curCourse(aStud,CList)]) ifTrue:
        [:cList|$logicBase retract: [curCourse(aStud,cList)].
                $logicBase assert: [curCourse(aStud,cList addHead: aCourse)]]].
advise: aStud
  "Advise a student on the kind of courses that (s)he can take."
  |list|
  list := PList new.
  (aStud cpuQuota > 0) ifTrue:
    [aStud deductCpuQuota.
     ($logicBase query: [canTake(aStud,ACourse)]) whileMoreDo:
         [:aCou|list addHead: aCou].
     ^list]
...

Invariant Instance Methods:
salary
    "Answer my salary."
    ^salary
```

Figure 1: Code Fragments of the `Registrar` Class Definition

course under consideration, the method proceeds to update the current-course database encoded in the logic part of the program. A logical query is represented as an instance of the `Block` class [7]. `CList` is a logical variable (using PROLOG's upper case letter convention) and denotes an unknown in the query. If the logical derivation activated by the query is successful, a `PTrue` (PROLOG true) object that contains a set of unary solution tuples for `CList` will be returned. The `PTrue` object returned can respond to the `ifTrue:` message with a block object with one argument. The argument of the block will take on the value of the first solution tuple, which is an instance of `PList` (PROLOG list), a subclass of `PTerm` (PROLOG term). The two statements inside the block serve to update the `curCourse/2` database in the logic part of the program. The `addHead:` message prepends a course object to the head of the `PList` object, held in variable `cList`. If the query fails, a `PFalse` (PROLOG false) object is returned.

The `advise:` method takes as argument a student and advises the student on the courses

```
$tom =:= Student name:"Tom" age:20 sex:"M" year:1 major:"CSC" id:123 quota:10.
$mary =:= Student name:"Mary" age:22 sex:"F" year:2 major:"CSC" id:456 quota:10.
...
$jim =:= Lecturer name:"Jim" age:32 sex:"M" dept:"CSC" fac:"ENG" salary: 1000.
...
$csc4190 =:= Course code:"CSC4190" content:"AI" major:"CSC".
...
```

Figure 2: Code Fragments for Defining Persistent Objects

that (s)he can take. After the method has checked the student's CPU access quota, it enquires `$logicBase` about the set of eligible courses for the student. If the set is non-empty, a set of unary solution tuples will be contained in the `PTrue` object returned from a successful logical derivation. The upper case `ACourse` is a logical variable. What is different in this case is that the `PTrue` object receives a `whileMoreDo:` message, which evaluates its argument block once for each solution tuple contained in the `PTrue` object. The argument block of `whileMoreDo:` collects the eligible course of the student in a `PList` object.

3.3 The Logic Part

In the logic part, persistent objects can be defined and named using the classes defined in the object part of an $\mathcal{OLI}$ program. Example definitions of persistent objects in the university registration program are shown in figure 2. These definitions are made at the beginning of the logic part, followed by program code written in the style of PROLOG rules.

Predicate definitions in the logic part consist of a set of *generalized definite clauses*, which are definite clauses [13] containing persistent objects defined in addition to Herbrand terms [13]. The logic part can only access the invariant methods of the persistent objects created from the object classes defined in the object part. In figure 3, we illustrate the syntax and informal semantics of generalized definite clauses using code fragments from the university registration program. The `coursesTaken/2`, `curCourse/2`, `preReq/2`, and `maxLoad/2` predicates define respectively the database of courses taken by individual students, courses currently enrolled in by individual students, the prerequisite courses of individual courses, and the maximum course load for students in each year of study. The `degreeReq/2` predicate defines the degree requirement of each major stream. The `canTake/2` predicate defines a binary relation on the eligible courses for individual students. It can be used to check whether a student can take a particular course and what course a student can take. A student can take a course if (1) (s)he has not reached the maximum course load, (2) (s)he has taken all the prerequisite of the course, and (3) the student's major stream matches the course's major stream. The definition of the `canTake/2` predicate reflects exactly the said rule. Note that the invariant message `year` and `major` are sent to the student object to inquire its year of study and major stream. Similarly, the invariant message `major` sent to a course object inquires the major stream of the course. The predicates `length/2`, `subset/2` and `delete/3` are those that can be found in standard PROLOG text [20]. A student can graduate with a degree in a particular major if (1) the student's major stream matches with the input major stream and (2) the student has taken all the required courses of the major. Again, this rule is reflected clearly in the definition of the `graduationStudent/2` predicate.

```
coursesTaken($tom, []).                     preReq($csc234, [$csc110,$csc111]).
coursesTaken($mary, [$csc110,art101]).      ...
...
curCourse($tom,[$csc110,$csc111,$his179]).    maxLoad(1,6).  maxLoad(2,6).
curCourse($mary,[$csc234,$csc245,$art201]).   maxLoad(3,5).  maxLoad(4,4).
...
degreeReq(...) :- ...

canTake(Student, Course) :-
    curCourse(Student, CC), length(CC, L), maxLoad(Student year, ML),
    L <= ML, coursesTaken(Student, CT), preReq(Course, PR),
    subset(PR, CT), Student major = Course major.
...
graduationStudent(Student, Major) :-
    Student major = Major, coursesTaken(Student, CT),
    degreeReq(Major, ReqCourses), subset(ReqCourses, CT).
```

Figure 3: Code Fragments from the Logic Part

3.4 Program Execution

Being a multiparadigm language, $\mathcal{OLI}$ allows users to invoke its execution via either the logic part or the object part. In the $\mathcal{OLI}$ programming environment, a workspace similar to that of SMALLTALK is provided. To find out whether Mary can take the course CSC341, users can post the query "`?- canTake($mary,$csc341).`" in the workspace. To add the course CSC267 for Paul, users can try evaluating the message expression "`registrar add: $paul course: $csc267`," where `registrar` is an instance of the `Registrar` class. Mixed paradigm queries and message expressions are also possible. The query "`?- canTake($mary,Course), Course major = csc.`" inquires for computer science courses that Mary may take. The message expression

```
"$logicBase query: [canTake($mary,Course),Course major = csc] ifTrue:
            [:aCourse|register add: $mary course: aCourse]."
```

adds an eligible computer science course for Mary. In the execution of logical queries and message expressions, execution may flow back and forth between logical deduction and message evaluation. Details of the $\mathcal{OLI}$'s operational semantics are recorded in sections 5 and 6.

4 Syntax

The $\mathcal{OLI}$ language, in its current form, resembles a blend of PROLOG and SMALLTALK, which are representational languages from logic programming and object-oriented programming respectively. The framework of $\mathcal{OLI}$, however, by no means depends on any peculiar features of PROLOG and SMALLTALK. Therefore, $\mathcal{OLI}$ is more a language scheme than a specific language. Our subsequent description remains valid if we replace PROLOG and SMALLTALK by any pure logic programming and object-oriented languages, such as Gödel [9] and Eiffel [16].

NameOfSuperclass **subclass:** #NameOfClass
instanceVariableNames: 'instVarName1 ...'
classVariableNames: 'ClassVarName1 ...'

Variant Class Methods:
Method definitions

Invariant Class Methods:
Method definitions

Variant Instance Methods:
Method definitions

Invariant Instance Methods:
Method definitions

message selector and argument names
"comment stating purpose of message"

|temporary variable names|
statements

Figure 4: Templates of the Class and Method Definitions in $\mathcal{OLI}$

4.1 The Object Part

The object part of an $\mathcal{OLI}$ program consists of a set of class definitions, which formulate the behaviour of the classes' object instances. Each class must have state variable declarations and (class and instance) method definitions [7]. Each object is an *instance* of a class. A class defines a set of objects (its instances) that share the same response to a common message but these objects have different states in general. A *message* requests an operation on the part of the receiver object. It is up to the receiver to decide how to respond to the message based on the corresponding method definition. A *message expression* describes a receiver, a selector, and possibly some arguments. The receiver and arguments are described by other expressions, whereas the selector is specified literally. We leave unspecified whether the object part should be typed and whether the object part should support class inheritance. In our current SMALLTALK-based implementation, we support class inheritance but not typing.

SMALLTALK takes the view that everything, including class, is an object. We adopts this viewpoint in $\mathcal{OLI}$. Therefore, each class has *class methods* for the class itself and *instance methods* for instances of the class. For each type of method, we differentiate between *variant* and *invariant* methods. The former, besides computing a function, has the side-effect of creating new objects and/or changing states of some existing objects; while the latter is used for either state enquiry or function abstraction. For example, the `salary` method in the `Registrar` class is for state enquiry and the `+` message in the message expression `1+2` represents function abstraction. The templates of $\mathcal{OLI}$ class and method definitions are shown in figure 4, where keywords are in bold.

4.2 The Logic Part

The logic part of an $\mathcal{OLI}$ program starts with object declaration followed by a set of predicate definitions written in the style of PROLOG. Object declaration allows programmers to create and name different object instances using the classes defined in the object part. It has the form:

Name **::=** Classname ObjectCreationMessage.

The objects created are called *primitive o-terms*, which have a name, a state (consisting of primitive o-terms), and a set of state-preserving invariant methods. Since primitive o-terms are instances of the classes in the object part, it possesses variant and invariant methods. *Only invariant methods are accessible by the logic part*. An important requirement of invariant methods is that they always terminate. An *o-term* is either a primitive o-term or a message expression with an o-term receiver. The result of evaluating a message expression is always a primitive o-term, which is the *value* of the expression. For the purpose of this paper, it suffices to assume a *standard message evaluation procedure* of the SMALLTALK language.

An *alphabet* of a first-order language L with o-terms is composed of a set ν_L of variables, a set σ_L of constants, a set μ_L of o-terms, a set Σ_L of function symbols, a set Π_L of predicate symbols, a set of connectives, a set of quantifiers, and a set of punctuation symbols. O-terms also denote individuals. We are now ready to define the first order language used in the logic part of $\mathcal{OLI}$ given by an alphabet. We need an extended notion of term [13]. A constant is a term. A variable is a term. An o-term is a term. If $\mathtt{f} \in \Sigma_L$ is an n-ary function symbol and $t_1, \ldots, t_n$ are terms, then $\mathtt{f}(t_1, \ldots, t_n)$ is a term. If $\mathtt{p} \in \Pi_L$ is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms, then $\mathtt{p}((t_1, \ldots, t_n)$ is an *atomic formula* or an *atom*. A *generalized definite program* consists of a finite set of *generalized definite clauses* of the form $A \leftarrow B_1, \ldots, B_m$ where $m \geq 0$, A and B_i are atoms for $i = 1, \ldots, m$. A is called the *head* of the clause and $B_1, \ldots, B_m$ is the *body* of the clause. A *generalized definite goal* is a generalized definite clause without a head. The logic part of an $\mathcal{OLI}$ program is a generalized definite program. The word "generalized" can be omitted hereafter when the context is clear.

5 Logical Semantics

The introduction of o-terms into first-order language extends the notion of Herbrand universe [13]. We define the *enriched Herbrand universe* (**eHu**) U_L of a first-order language L to be the set of all ground terms, constructable from σ_L, μ_L, and Σ_L. *Ground terms* are terms not containing variables. Let L be the first-order language used in the logic part of $\mathcal{OLI}$. Thus the **eHu** of L is the result of augmenting the Herbrand universe of L with object instances of the classes defined in the object part. The *enriched Herbrand base* (**eHb**) B_L of a first-order language L is the set of all ground atoms, constructable from Π_L and U_L. The *Herbrand interpretation* for L is given by:

- The domain of the interpretation is U_L.
- Constants $c \in \nu_L$ and o-terms $o \in \mu_L$ are assigned themselves in U_L.
- If $\mathtt{f} \in \Sigma_L$ is an n-ary function symbol, then $\mathtt{f}$ is assigned the mapping from $(U_L)^n$ into U_L defined by $(t_1, \ldots, t_n) \rightarrow \mathtt{f}(t_1, \ldots, t_n)$.
- If $\mathtt{p} \in \Pi_L$ is an n-ary predicate symbol, then $\mathtt{p}$ is assigned a mapping from $(U_L)^n$ into $\{\mathtt{true}, \mathtt{false}\}$.

The meaning of the connectives, quantifiers, Herbrand model, and logical consequences can now be defined as usual. Without loss of generality, we assume that the underlying first order language of a set of generalized definite program P is defined by the constants, o-terms, function symbols and predicate symbols appearing in P. Therefore, it makes sense to refer to the **eHu** U_P and **eHb** B_P of a program P and also refer to the Herbrand interpretations of P as subsets of B_P.

5.1 Declarative Semantics

O-terms denote additional individuals, extending the Herbrand universe. *The restriction of only state-preserving messages to o-terms ensure the static nature of the enriched Herbrand universe throughout a logical computation.* This is an essential property for the following results to hold.

Proposition 5.1: (Model Intersection Property) If P is a generalized definite program and $\{M_i\}$ is a non-empty set of Herbrand models for P, then $M_P = \cap\{M_i\}$, called the *least Herbrand model*, is a Herbrand model for P. ■

Theorem 5.2: If P is a generalized definite program, then $M_P = \{A \in B_P \mid P \models A\}$. ■

5.2 Operational Semantics

Unification is an important step in the execution of logic programs. In unifying two expressions containing o-terms, we first replace each non-primitive o-term in the expressions with its value by evaluating the o-term (a message expression), which, by definition, invokes only state-preserving invariant methods. Therefore, the notion of substitution must be extended to include primitive o-terms. A *substitution* is a finite set of the form $\{v_1/t_1, \ldots, v_n/t_n\}$, where v_i's are variables, each t_i is a term distinct from v_i, each t_i cannot be a non-primitive o-term, and the variables $v_1, \ldots, v_n$ are distinct. The definitions of unification and most general unifier, and the unification algorithm can remain the same by treating o-terms as constants. That is to say a primitive o-term unifies only with itself.

Execution of generalized definite programs is based on a generalization of SLD-resolution [13]. Let P be a generalized definite program and $G_i = \leftarrow A_1, \ldots, A_j, \ldots, A_n$ a definite goal, where $n > 0$. The goal $G_{i+1} = \leftarrow (A_1, \ldots, A_{j-1}, B_1, \ldots, B_m, A_{j+1}, \ldots, A_n)\theta$ is *derived* from G_i using definite clause c and most general unifier θ if

1. if A_j, the selected atom in G_i, contains non-primitive o-terms (i.e. message expression), then each message expressions is replaced by its value,
2. $c = A \leftarrow B_1, \ldots, B_m$ is a suitable variant [13] of a clause in P and θ is the most general unifier of A_j and A.

A *derivation* is a, possibly infinite, sequence of definite goals $G = G_0, G_1, G_2, \ldots$ such that G_{i+1} is derived from G_i generating substitution θ_{i+1}. A derivation is *successful* if it is finite and the last goal contains no atoms. The *computed answer* of a successful derivation of length m is the composition $\theta_1\theta_2 \cdots \theta_m$ of the most general unifiers generated in each step of the derivation. A derivation is *finitely failed* if it is finite, ends in a non-empty goal and condition 2 does not hold.

The evaluation of non-primitive o-terms during a derivation step is deterministic and provides a channel to invoke object computation during logical inference.

5.3 Soundness and Completeness

A generalized definite program asserts what is true about the intended relations among individual in the **eHu**. Since only state-preserving messages can be sent to o-terms during derivation, the **eHu** is static. In addition, evaluation of non-primitive o-terms always terminate. Thus it is easy to check that the derivation procedure for generalized definite program is both sound and complete.

Theorem 5.3: (Soundness) Let P be a generalized definite program. If a definite goal $\leftarrow G$ has a successful derivation with computed answer θ, then $P \models \forall(G\theta)$, where $\forall(F)$ is the universal closure of the formula F. ■

Theorem 5.4: (Completeness) Let P be a generalized definite program and $\leftarrow G$ a definite goal. If there exists a substitution θ such that $P \models \forall(G\theta)$, then there exists a successful derivation for $\leftarrow G$ with computer answer σ and a substitution γ such that $\theta = \sigma\gamma$. ■

6 Object-Oriented Semantics

In the following, we give an informal account of the semantics of the interface between the logic part and the object part as viewed from the object part.

In object-oriented programming, a system consists of a pool of objects created as instances of the defined classes. In $\mathcal{OLI}$, this *object pool* is augmented by four types of objects from the logic part. Herbrand terms are objects. In the $\mathcal{OLI}$ scheme, Herbrand terms are instances of the `PTerm` class, which provides methods to traverse as well as to update the structure of the terms. There are three special built-in classes in $\mathcal{OLI}$: `LogicBase`, `PTrue`, and `PFalse`. `LogicBase` has only one instance `$logicBase`, which represents the logic part of an $\mathcal{OLI}$ program. The state of `$logicBase` is the collection of clauses in the logic part, *i.e.* the program in the logic part, which can be updated using the `assert:` and `retract:` messages. Another important message to `$logicBase` is `query:`, which takes as argument a logical query and returns an instance of either the `PTrue` or `PFalse` class. `PTrue` is a subclass of the `True` class in SMALLTALK. Thus `PTrue` objects respond to the usual logical operations, such as `&` and `ifTrue:`. `PTrue` objects has an additional state variable which contains the set of answer tuples of a successful logical derivation. The answer tuples can be enumerated using the `whileMoreDo:` iterator message. The `PFalse` class is an alias of the `False` class of SMALLTALK, instances of which are returned by failed logical derivations. Instances of these special classes are part of the object pool (thus also the **eHu**). This design has interesting meta-programming implication, which is outside the scope of this paper.

Execution of an $\mathcal{OLI}$ program can be invoked from the object part by sending a message to an object. In this case, the *outermost* computation is an object-oriented computation. Execution follows the usual SMALLTALK style execution until a `query:` message is sent to the `$logicBase` object. Control is then temporarily transferred to the logic part, where generalized SLD-resolution is performed. We call this a *sub-computation*, which is logical in this case. Control is returned to the object part after the sub-computation is finished. Execution continues in an object-oriented style until program termination or another `query:` message is sent to the `$logicBase` object. Objects can change states before and after a logical sub-computation. During a logical sub-computation (in the form of a generalized SLD-derivation), invariant messages may be sent to o-terms, in which case control will be transferred to the object part. This is an object-oriented sub-computation embedded in a logical computation. No state change can take place, which is guaranteed by the fact that only invariant methods can be invoked.

Similar control flow pattern is exhibited when an $\mathcal{OLI}$ program execution is initiated by a logical query. The outermost computation is logical. Control may be transferred to the object part when messages are sent to o-terms. However, these object-oriented sub-computation are all embedded in a logical computation. Therefore, no state updates will take place.

In summary, $\mathcal{OLI}$ is designed in such a way that no state updates occur in a logical computation or any object-oriented sub-computation embedded in a logical computation. In other words, state updates can occur only in an outermost object-oriented computation. When object state changes between two different logical sub-computations during the execution of an outermost object-oriented computation, each logical derivation corresponds to a different static **eHu**.

7 Concluding Remarks

The contribution of this paper is two-fold. First, we present the motivation and careful design of a multiparadigm language combining the power of object-orientation and logical deduction. Programmers can choose to program entirely in any one of the paradigms without loss in expressiveness, execution efficiency, and complication in semantics. Programming in the mixed paradigm, however, provides extra expressiveness not available in either paradigm. Second, we give the formal syntax and semantics of $\mathcal{OLI}$. Soundness and completeness result is established, retaining the simple and elegant semantics of logic programming. An important characteristics of $\mathcal{OLI}$ is that an $\mathcal{OLI}$ computation makes sense in the context of both paradigms. From the object-oriented point of view, an $\mathcal{OLI}$ computation is a sequence of object interactions via messages. Logic computation is just a message to the `$logicBase` built-in object. From the logical point of view, an $\mathcal{OLI}$ computation is a generalized SLD-derivation. Object computation merely inquires the states of o-terms or performs function abstraction. We believe that our work provides a good reference point for foreign language interface designers for either PROLOG or SMALLTALK.

The following open problems remain to be answered. First, we observe that certain meta-programming and arithmetic facilities in PROLOG can now be explained as object computation. For example, the goal "`X=1+2`" can be viewed as unifying variable "`X`" with the non-primitive o-term "`1+2`." It is interesting to study the meta-programming implications in depth. Second, the built-in object `$logicBase` encapsulates the logic part of an $\mathcal{OLI}$ program. By allowing more than one `$logicBase` object, we may get a module system for the logic part of $\mathcal{OLI}$ for free. Third, objects are inherently typed. It is interesting to investigate if typing can be introduced into the logic part of $\mathcal{OLI}$, resulting in an inheritance scheme and generic mechanism for predicates. For this, we need to generalize the notion of o-terms. The ψ-calculus [1, 2] would be a good starting point. Fourth, more work is needed for an efficient implementation of the $\mathcal{OLI}$ language. A programming environment similar to that of SMALLTALK should also be investigated.

Acknowledgement

This research was supported in part by a UK/HK Joint Research Scheme grant (HKG/520/8-6), funded by the British Council.

References

[1] H. Aït-Kaci. *A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially-Ordered Type Structures*. PhD thesis, University fo Pennsylvania, Pennsylvania, Philadephia, PA, USA, 1984.

[2] H. Aït-Kaci. An algebraic semantics approach to the effective resolution of type equations. *Theoretical Computer Science*, 45:293–351, 1986.

[3] H. Aït-Kaci and A. Podelski. Towards a meaning of LIFE. *Journal of Logic Programming*, 16(3 & 4):195–234, 1993.

[4] V. Alexiev. Mutable object state for object-oriented logic programming: A survey. Technical Report TR93-15, Department of Computer Science, University of Alberta, 1993.

[5] G. Booch. *Object-oriented Analysis and Design*. Addison-Wesley, second edition, 1994.

[6] A. Davison. A survey of logic programming-based object oriented languages. Technical Report 92/3, Department of Computer Science, University of Melbourne, 1992.

[7] A. Goldberg and D. Robson. *SMALLTALK-80: The Language*. Addison-Wesley, 1989.

[8] E. Gullichsen. BiggerTalk: Object-oriented Prolog. Technical Report STP-125-85, MCC-STP, Austin, TX, 1985.

[9] P. Hill and J. Lloyd. *The Gödel Programming Language*. The MIT Press, 1994.

[10] Y. Ishikawa and M. Tokoro. Concurrent object-oriented knowledge representation language Orient84/K: Its features and implementation. In *OOPSLA'86*, pages 232–241, 1986.

[11] S. Janson. *AKL - A Multiparadigm Programming Language*. PhD thesis, Computing Science Department, Uppsala University, Uppsala, Sweden, 1994.

[12] J.H.M. Lee and P.K.C. Pun. $\mathcal{O}$bject $\mathcal{L}$ogic $\mathcal{I}$ntegration: a scheme for multiparadigm design and programming. In *Proceedings of the Eighteenth International and Exhibition Technology of Object-Oriented Languages and Systems*, pages 265–278, Melbourne, Australia, November 1995.

[13] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second edition, 1987.

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

[15] F. Mellender. An integration of logic and object-oriented programming. *SIGPLAN Notices*, 23(10):181–185, 1988.

[16] B. Meyer. *Eiffel: the Language*. Prentice Hall, 1992.

[17] C.D.S. Moss. *Prolog++: The Power of Object-Oriented and Logic Programming*. Addison-Wesley, 1994.

[18] P.K.C. Pun. A multiparadigm programming approach for developing knowledge-based systems. In *Proceedings of the 19th IEEE Annual International Computer Software and Applications Conference*, pages 362–368, 1995.

[19] E. Shapiro. A subset of Concurrent Prolog and its interpreter. Technical Report CS83-06, Weizmann Institute of Science, Israel, February 1983.

[20] L. Sterling and E. Shapiro. *The Art of Prolog—Advanced Programming Techniques*. The MIT Press, 1986.

[21] S. Wu. Integrating logic and object-oriented programming. *OOPS Messager*, 2(1):28–37, 1991.

[22] C. Zaniolo. Object-oriented programming in Prolog. In *Proceedings of the IEEE International Symposium on Logic Programming*, pages 265–270, 1984.