I^+: A Multiparadigm Language for Object-Oriented Declarative Programming

K.W.Ng and C.K.Luk
Department of Computer Science
The Chinese University of Hong Kong
Shatin, Hong Kong
Fax: 852-6035024 Email: kwng@cs.cuhk.hk

Abstract

This paper presents a multiparadigm language I^+ which is an integration of the three major programming paradigms: object-oriented, logic and functional. I^+ has an object-oriented framework in which the notions of classes, objects, methods, inheritance and message passing are supported. Methods may be specified as clauses or functions, thus the two declarative paradigms are incorporated at the method level of the object-oriented paradigm. In addition, two levels of parallelism may be exploited in I^+ programming. Therefore I^+ is a multiparadigm language for object-oriented declarative programming as well as parallel programming.

Keywords: Multiparadigm, Object-oriented paradigm , Logic paradigm , Functional paradigm

1 Introduction

A *multiparadigm language* is a language that supports more than one programming paradigm. Multiparadigm languages are desirable for the following reasons:

- A programmer can choose the most appropriate paradigm for a particular problem at hand so that the gap between the design phase and the programming phase can be minimized.
- Existing software (e.g. library subroutines) written in languages of a component paradigm can be reusable in a multiparadigm environment.
- It is natural to model real-world objects using multiparadigm languages since they can be viewed as loosely coupled distributed systems with multiparadigm cooperating subsystems.

Among the existing paradigms, we are especially interested in integrating three of them, namely *object-oriented*, *logic* and *functional* because of their individual salient features. The object-oriented paradigm[37] is distinguished by its capability to organize and reuse programs through encapsulation, polymorphism, and inheritance. On the other hand, both the logic[24] and the functional[16] paradigms are distinguished for their expressiveness in writing programs and are thus called *declarative paradigms*. Logic programs are represented by relations and functional programs are represented by functions. A language that integrates these three paradigms in an appropriate manner could support most of their merits in a single framework and is appropriate for *object-oriented declarative programming*.

This paper presents a multiparadigm language I^+ which integrates these three paradigms. Section 2 presents the elements of I^+. Section 3 is a discussion of the various applications of the language. Section 4 outlines the implementation of a prototype of I^+. Section 5 concludes our work and Section 6 considers some future directions for our language.

2 The multiparadigm language I^+

2.1 Overview

I^+ is a multiparadigm language which integrates the three major programming paradigms: object-oriented, logic and functional in a single environment. It is an enhanced descendent from another multiparadigm language *I* [29, 30]. I^+ has an object-oriented framework in which the notions of classes, objects, methods, inheritance and message passing are supported. Nevertheless, methods in I^+ are different from those in imperative object-oriented languages in the sense that they are not specified as procedures but as *clauses* or *functions*. Thus the two declarative paradigms are incorporated at the method level of the object-oriented paradigm. In addition, two levels of parallelism are exploited in I^+ programming. Therefore I^+ is a multiparadigm language for object-oriented declarative programming as well as parallel programming. In the following sections, we examine the elements of I^+ in details.

2.2 An Object-Oriented Framework

An I^+ program is a collection of *class definitions* and *queries*. An outline of the syntax of I^+ programs is shown in Fig. 1. A complete description of the syntax can be found in [25].

program → *classes queries*
classes → ε
classes → *class classes*
class → *logic_class* | *functional_class*
queries → ε
queries → *query queries*
query → *logic_query* | *functional_query*
...

Fig. 1 The structure of an I^+ program

There are two types of classes, namely *logic_class* and *functional_class*. They are different in the way that their methods are defined. Methods of *logic_class* (often referred as *logic methods*) are clauses while those of *functional_class* (often referred as *functional methods*) are functions. Similarly, there are two types of queries: *logic_query* and *functional_query*. A *logic_query* is for goal satisfaction and is

prefixed by the symbol `?-`. A *functional_query* is for evaluating expressions and is prefixed by the symbol `>`. For example, `?-foo(X)` is a query to satisfy the goal `foo(X)` while `>1+2*3` is a query to evaluate the arithmetic expression `1+2*3`.

2.2.1 The Declarative Meaning of Classes and Methods

In the sense of imperative object-oriented programming, a *method* is defined as a procedure designated for accomplishing a particular task in a class of objects which share some common behaviors. This is the *procedural* view of methods. There is another view of methods in I^+ and some other integrated languages like [11, 26]. We can regard a class as a collection of things that we know to be true of objects of this class. The truths are represented as relations or functions (strictly speaking, functions are a kind of relations also). Such a collection of truths is called a *theory*. So *a class denotes a theory and methods are vehicles to know about the truths of a theory*. An I^+ program, which consists of a set of class definitions, allows us to describe more than one theory within a single system.

2.2.2 Class Structure

The general structure of all I^+ classes, either *logic_classes* or *functional_classes*, is written in BNF as follows:

```
<class type> <class signature>
[inherit
      <specification of variables and methods to be inherited from
      a class>+
]
[world
      <declaration of methods which are accessible to public>
]
[family
      <declaration of variables and methods which are accessible
      to the descendants of this class>
]
[personal
      <declaration of variables and methods which are local to
      this class>
```

```
]
implementation
      <definitions of all declared methods of this class>
class_end
```

Fig. 2 The general class structure

The words in bold are I^+ keywords, the phases in angled brackets are descriptions of the text that should be placed there, square brackets denote that the things inside are optional, and the + stands for one or more occurrence.

A class definition begins by specifying its *type* which should be one of the keywords **`logic`** or **`functional`**. Then the *class signature* is defined. A class signature has the form:

$$\texttt{classname(\&p}_1\texttt{,...,\&p}_n\texttt{)}$$

`classname` is the name of the class being defined. $\texttt{\&p}_1\texttt{,...,\&}_n$ are called *class parameters*. The symbol `&` distinguishes a class parameter from other identifiers. Class parameters provide a flexible and dynamic way to define methods. For example, consider the following class definition:

```
logic person(&NAME, &ID, &GENDER)
world
      name/1,id/1,gender/1
implementation
      name(&NAME).
      id(&ID).
      gender(&GENDER).
class_end
```

Program 1 The definition of class `person`

The information of a person is represented by three methods `name/1`, `id/1` and `gender/1` in the form of Prolog clauses (the notation `m/k` denotes a method of name `m` and arity `k`). We can create instances of `person` with different values of `&NAME`, `&ID`, and `&GENDER` for different persons. For instance, `person(mary,476,female)` denotes a female whose name is Mary and her identity number (id) is

476. Without class parameters, we need to define separately a class for each person. Thus class parametrization is a means to achieve data abstraction. Values of the class parameters are supplied when a subclass or an object of that class is made, and are substituted into the right places through the call-by-name mechanism.

Before explaining the **inherit** part, let us first consider the other components in Fig.2. The **world**, **family** and **personal** parts are constructs for describing the *access control* of methods and state variables defined in the class. The optional **world** part declares a group of methods that are accessible to public. For instance, the three methods in Program 1 are such methods. The optional **family** part declares a group of *state variables* and methods that are accessible to all descendants of the class being defined and the class itself. The optional **personal** part declares a group of state variables and methods that can only be used locally in the class being defined. For instance, in the following parts of a class definition:

```
logic c
...
world m/1, n/2
family x, y, p/3, q/4
personal z, r/5, s/6
...
class_end
```

Program 2 Parts of a class definition

Suppose that the class `c` has a subclass `d` and there is another class `e` which is not in the family containing `c` and `d`. Then the above declarations tell us that methods `m/1` and `n/2` are accessible to classes `c`, `d` and `e`. Methods `r/5` and `s/6` and the state variable `z` are only accessible to class `c`. Methods `p/3` and `q/4` and state variables `x` and `y` are accessible to classes `c` and `d` but not `e`.

Our way of specifying access control of methods and state variables is more general than those in some other integrated languages [8, 11, 12, 26]. In these languages, there are only two levels of access control: public and private.

The **implementation** part contains the definitions of all methods declared previously in the **world** part, the **family** part and the **personal** part. If the class being defined is of type **logic**, then the methods are implemented by a set of clauses. Otherwise, the methods are implemented by a set of functions.

The class definition finishes with the keyword **class_end**.

2.2.3 Inheritance

Inheritance is a mechanism for defining a class by exploiting common behaviors among other existing classes. If a class `C` inherits *something* from another class `P`, then `C` is said to be a *subclass* or a *child* of `P` and `P` is said to be a *superclass* or a *parent* of `C`. The term *something* usually refers to methods and state variables. In I^+, as in many other object-oriented languages [13, 33], a class is allowed to arbitrarily redefine inherited methods or state variables and to add new methods or state variables. Thus a class may hold more data and provide more methods than its superclass does.

The **inherit** *construct*

It is a two-way process to specify what things a subclass should inherit from its superclass. The superclass specifies a method or a state variable as *inheritable* by putting it in the **world** part or the **family** part of the class definition. In addition, all methods and state variables that the superclass has inherited from other classes are also *inheritable*. Nevertheless, only a subset of all inheritable methods and state variables are really inherited by the subclass. This subset is determined by the **inherit** part of the subclass's definition. The **inherit** part consists of a group of *inheritance-specifications*. Each such inheritance-specification is for declaring which methods and state variables are to be inherited from a particular class. An inheritance-specification has the following BNF:

```
<methods_and_variables> from <class signature>
```

which declares that the methods and state variables represented by `<methods_and_variables>` are to be inherited from the class denoted by `<class signature>`. There is a number of ways to specify such methods and state variables. The simplest way is to list them out. For instance, the following is an inheritance-specification:

```
[name/1,id/1,gender/1] from person(john,759,male)
```

which inherits all methods from the class `person(john,759,male)`. A short form of the above inheritance-specification is:

```
all from person(john,759,male)
```

The keyword **`all`** denotes all inheritable methods and state variables from the class listed after the keyword **`from`**. In the above example, all the three methods are inherited from `person(john,759,male)`.

Another way is to state what should *not* be inherited. For instance, the following inheritance-specification:

```
all except gender/1 from person(john,759,male)
```

declares that all inheritable methods except `gender/1` should be inherited from `person(john,759,male)`.

The ability to inherit only part of inheritable methods (state variables) is useful for modeling a kind of inheritance relationship in which the subclass shares most but not all properties of the superclass. For instance, a homeless person is a person who has no home. Thus it is natural to model homeless persons as a

subclass of persons by discarding the properties related to home (e.g. address, neighbors) in the inheritance process.

So far, we have not talked about the *access control* of the inherited methods and state variables: Are they **world** accessible or **family** accessible or **personal**? If it is not specified, an inherited method (state variable) is **world** accessible. We can specify the access mode of an inherited method (state variable) using the keyword **as**. For instance, in the following inheritance-specification:

```
all except gender/1 as family, gender/1 as personal
        from person(john,759,male)
```

The method `gender/1` is inherited as a **personal** method in the subclass. The methods `name/1` and `id/1` are inherited as **family** methods.

Multiple inheritance

It is common for a class to have multiple superclasses. A class should declare all of its superclasses in its **inherit** part in the form of inheritance-specifications. For instance, consider the following program fragment:

```
logic c1
inherit
      m/2, n/4, p/1 from c2
      m/2, q/3 from c3
      m/2, n/3 from c4
 ...
class_end
```

Program 3 A multiple inherited class `c1`

The class `c1` has three superclasses: `c2`, `c3` and `c4`.

In the above program fragment, the method `m/2` is inherited from three superclasses. It is necessary to choose a definition of `m/2` from the three superclasses when `m/2` of `c1` is called (There is no such need for the methods `n/3` and `n/4` since their arities are different). In I^+, such *conflicting methods* are sorted according to the three precedence rules used by CLOS[38] for linearizing inheritance hierarchies: *depth-first rule*, *left-to-right rule* and *up-to-join* rule. For the method `m/2`, the order should be: `c2 > c3 > c4` which means that the `m/2` of `c2` has the highest precedence. Thus the `m/2` of `c2` is chosen first. If `m/2` is a functional method (not our case), then the remaining method definitions on the precedence list (i.e. `m/2` of `c3` and `c4` in our case) are discarded since a function must be *deterministic*. On the contrary, since `m/2` is a logic method which can be *nondeterministics*, the remaining method definitions may be used one by one in the precedence order when `m/2` is backtracked. This ability to return alternative solutions to a method invocation is a contribution of the logic paradigm to the object-oriented paradigm.

2.2.4 Objects and messages

The term *object* has two different interpretations. Conceptually, an object is a particular instance of a class (theory). It models a real world entity (logical or physical). From the implementation view, however, an object is an execution environment which contains both codes and storage. Due to these two different views, there are two ways to create an object in I^+. The first way is to simply write the class name followed by a list of actual class parameters. For example, refer to Program 1, both `person(john,1756,male)` and `person(mary,3042,female)` are objects of the class `person` with different actual class parameters. They correspond to two persons in the real world. Another way is to use the system-call **`create/2`** to create an execution environment (an object) on a particular processor:

`create(`*`instance`*`,`*`location`*`)`

where *instance* is a class name followed by a list of actual class parameters and *location* is an expression for calculating the processor's position on a network. Some examples of *location* follow:

***location* expression**	**Allocated processor's position**
3	The processor numbered 3
2+5*12	The processor numbered 62
Left	The processor on the immediately left of the processor executing create/2
(7,9)	The processor on the row 7 and column 9 of a two-dimensional grid

Table 1 Some examples of *location* expressions

The following expression:

```
O = create(person(ann,213,female),19)
```

allocates the processor 19 for the object `person(ann,213,female)` and this environment is denoted by `O`.

An object is activated by invoking its methods through *message calls*. A message call is of the form:

$$O::m(X_1,...,X_n)$$

where `O` is the identity of an object, `m` is the name of a method, and $X_1,...,X_n$ are the actual parameters of `m`. For instance, both

```
... person(john,1756,male)::name(X) ...
```

and

```
...O=create(person(john,1756,male),15),O::name(X)...
```

are message calls to invoke the method `name/1` in a `person` object. They succeed with `X` instantiated to `john`.

2.3 Logic Methods

2.3.1 Augmented Horn clauses

A *logic method* $\texttt{p(x}_1\texttt{,...,x}_n\texttt{)}$ is a predicate `p` with arguments $\texttt{x}_1\texttt{,...,x}_n$ in the sense of logic programming. Such a predicate is defined by a set of *augmented Horn clauses*. An augmented Horn clause is a Horn clause [24] with the capability to use methods of other objects. Formally, an augmented Horn clause has the form:

$$Head \text{ :- } Body_1,..., Body_n. \qquad n \geq 0$$

where *Head* is a predicate $p(t_1,...,t_m)$ with $m \geq 0$ and t_i is an *augmented term*. An augmented term is defined recursively as follows:

i A variable is an augmented term

ii A constant is an augmented term

iii If c is a k-ary constructor and $y_1,...,y_k$ are augmented terms, then $c(y_1,...,y_k)$ is an augmented term

iv If O' is an object and E' is an expression, then $O'::E'$ is an augmented term

The original definition of a *term* in logic programs consists of (i-iii) only. (iv) denotes the value of an expression. The usage of (iv) is discussed in section 2.3.3.

$Body_i$ has two possible forms:

i $p'(t'_1,..., t'_r)$ a locally defined predicate

ii $O::p'(t'_1,..., t'_r)$ a predicate defined in a logic object O

where $r \geq 0$ and t'_i is an augmented term. The (ii) form is for the purpose of *distributed inference*.

2.3.2 Distributed Inference

In logic programming, computation is identified with inference. In conventional logic programming systems like Prolog, all facts and rules are stored in a centralized database and any one of them may be used in an inference step. On the contrary, in our system, since we are handling more than one theory, we must be able to do inference across different theories. This is achieved by the message call $O::p(x_1,...,x_n)$ which causes the system to deduce the predicate $p(x_1,...,x_n)$ in the context of object O. The following example demonstrates how inference can be done in multiple objects. Suppose we define a class `student` as follows:

```
logic student(&NAME, &AGE, &SCHOOL, &LEVEL)
world name/1, school/1, level/1, classmate/1
implementation
      name(&NAME).
      school(&SCHOOL).
      level(&LEVEL).
      classmate(X) :-
            object_of(student,L),
            member(S,L), not(myself(S)),
            school(MY_SCHOOL), S::school(MY_SCHOOL),
            level(MY_LEVEL), S::level(MY_LEVEL),
            S::name(X).
class_end
```

Program 4 The definition of class `student`

The method `classmate(X)` states the relation "one of my classmate is named `X`". It is implemented by first finding the list of all `student` objects through `object_of/2`, and then checking them one by one to see if a `student` object has the same school and level of the asking object by sending messages to

other `student` objects. Thus, a deduction of the `classmate(X)` requires doing inference in multiple objects. Both `object/2` and `myself/1` are system predicates. `myself(S)` instantiates `S` to the identify of the calling object . It is necessary here to avoid a `student` object to regard itself as its classmate.

2.3.3 Using Expressions in Logic Methods

A common style of Prolog programming is

```
..., X1 is X+3 p(X1), ...(*)
```

The `is/2` predicate is used to assign a value to the argument of p. This looks like the assignment statement `X1 := X+3` in Pascal and so is not logic. Also, it is quite clumsy to introduce a new variable for every use of `is/2`. A better way is to use an expression directly. For example, (*) may be rewritten as:

```
...,p(X+3), ...
```

I+ supports the use of expressions in logic methods. We can write an expression of the form *O*::*E* whenever a value is expected in a Horn clause. This causes the evaluation of the expression *E* in the context of the object *O*. For example, a class of objects for finding the greatest common divisor of two natural numbers can be defined as:

```
logic gcdclass
world gcd(N1, N2, N)
implementation
      gcd(N1,0,N1).
      gcd(N1,N2,N):-
            gcd(N2,functional::mod(N1,N2),N).
class_end
```

Program 5 The definition of class `gcdclass`

`gcd(N1,N2,N)` succeeds if the greatest common divisor of `N1` and `N2` is `N`. The function `mod/2` is predefined in the system class **`functional`**.

Higher-order functions can be used in logic methods. A function can be passed as an argument to another function in the expression *E* of *O*::*E* or can be returned by *O*::*E*. For instance, suppose that the function *g* is defined as follows in **`functional`**:

$$g\ x\ y = x * y$$

and a logic method `p(X,Y,Z)` is defined by:

```
p(X,Y,Z) :- G=functional::(g X), Z=functional::(G Y).
```

If we call `p(2,3,Z)` `G` is instantiated to an anonymous function $\lambda\ t.2t$ and then used for evaluating `(G 3)`. `Z` is instantiated to the result of 2*3, i.e. 6.

Our way of incorporating functions and expressions in logic methods can be classified as a *compilational* approach [35] to combining the functional and the logic paradigms. In the compilational approach, a program containing expression is said to be *isomorphic* to another program in which the expressions are replaced by new variables together with additional goals that supply the values of the new variables. An advantage of the compilational approach is that there is no need to modify the conventional unification algorithm to evaluate expressions in logic programs. What we need is a *preprocessor* that is able to transform a logic program embedded with expressions to another isomorphic logic program which is expression free.

2.3.4 State modeling

The capability of state modeling is essential to any practical language. I^+ supports state variables and the assignment statement in **logic** classes (but not **functional** classes) for modeling mutable state. As we have seen, state variables are declared in the **family** or **personal** parts of **logic** classes. State variables can be manipulated in much the same way as variables in imperative languages. For example, in the following program fragment,

```
logic city...
      ...
world born/1, dead/1, total/1...
family population, ...
      ...
implementation
      ...
total(population).
born(X) :- population is population + 1, ...
dead(X) :- population is population - 1, ...
      ...
class_end
```

Program 6 The definition of class `city`

`population` is a state variable. Its value is changed by the two assignment statements: `population is population+1` and `population is population-1`. It can also be accessed, as for logical variables, in the fact `total(population)`.

The main difference between state variables in I^+ and variables in imperative languages is that state variables can take any term as their value. Though state variables are powerful, they should be used only when it is necessary in order to minimize their damage to the declarative semantics of logic methods.

2.4 Functional Methods

2.4.1 The syntax of functions

Methods of **`functional`** classes are applicative functions. I^+ adopts the syntax of functions in Lazy ML [1]. A function f is defined by a *function declarator* which gives different return values for different patterns of input parameters. The general form of a function declarator is:

$$\begin{array}{llllll} & f & P_{11} & \ldots & P_{1n} = & E_1 \\ \| & f & P_{21} & \ldots & P_{2n} = & E_2 \\ & \ldots & & & & \\ \| & \mathrm{f} & P_{m1} & \ldots & P_{mn} = & E_m \end{array}$$

where f is the name of the function being declared. P_{ij}'s are patterns and E_i's are the expressions for different cases. The symbol ‖ means "or". It separates different patterns of input parameters. This declarator corresponds to the mathematical definition of a function:

$$f(x_1,\ldots,x_n) = \begin{cases} E_1 & x_1 = p_{11} and \ldots and \ldots x_n = p_{1n} \\ E_2 & x_1 = p_{21} and \ldots and \ldots x_n = p_{2n} \\ & \mathrm{M} \\ E_m & x_1 = p_{m1} and \ldots and \ldots x_n = p_{mn} \end{cases}$$

For example, we can define a group of boolean operations in a class boolean_ops as follows:

```
functional boolean_ops
world NOT/1,OR/2,AND/2...
      ...
implementation
            NOT   true        =     false
      ||    NOT   false       =     true
      and
            OR    true  true  =     true
      ||    OR    true  false =     true
      ||    OR    false true  =     true
      ||    OR    false false =     false
      and
            AND   true  true  =     true
      ||    AND   true  false =     false
      ||    AND   false true  =     false
```

```
        ||    AND   false false =     false
        and
              ...
class_end
```

Program 7 The definition of class boolean_ops

The definitions of three boolean operations: `NOT`, `OR` and `AND` are shown in the above program. `true` and `false` are the two predefined boolean values. The word `and` is used to combine two declarators to form a single declarator which defines both the identifiers from the declarators.

2.4.2 Abstract Data Types

A favorite usage of functional methods is to construct Abstract Data Types (ADTs). For example, we may define a class `list` for the list ADT as follows:

```
functional list
world empty(),insert/2,append/2,first/1,last/1
implementation
rec type List *a  = mt + elem (List *a) *a (List *a)
and   new                              = mt
and   insert (x, l)                    = elem mt x l
and   append (x,l)                     = elem l x mt
and   first (elem mt x l)              = x
    ||first (elem l1 x l2)             = first l1
and   last (elem mt x l)               = l
    ||last (elem l1 x l2)              = elem last l1 x l2
class_end
```

The statement `rec type List *a = mt + elem (List *a) *a (List *a)` defines three identifiers: the type `List`, the constructors `mt` and `elem`. The identifier `*a` is called type variable. For instance, `List Int` is the type of lists of integers. The type `List` is hidden from anywhere outside `list`. `new` is a function to return an empty list. `insert(x l)` returns a list with `x` as the first element and `l` the tail sublist (i.e. `[x|l]`). `append(x l)` returns the result of appending `[x]` to `l`. `first(l)` returns the first element of the list `l`. `last(l)` returns the tail sublist of the list `l`.

Since stacks and queues can be regarded as two kinds of lists, the classes for stack and queue can be derived from `list` as follows:

```
functional stack
inherit
      new as world,
      [insert/2,last/1,first/1] as personal from list
world push/2, pop/1, top/1
implementation
      push  e s   =     insert e s
and   pop   s     =     (first s, last s)
class_end
functional queue
inherit
      new as world,
      [append/2,first/1,last/1] as personal from list
world put/2, get/1
implementation
      put   e q   =     append e q
and   get   q     =     (first q, last q)
class_end
```

Program 8 Implementation of list, stack and queue ADTs by **`functional`** classes

Both `stack` and `queue` only inherit some useful methods from `list`. Except `new`, all inherited methods are treated as **`personal`** in order to avoid direct reference of such methods from users of `stack` and `queue`. `pop s` returns an ordered pair `(first s,last s)`. The first member of the pair is the element at the top of the stack `s`. The second member is the resultant stack of the pop operation. Similarly, `get q` also returns an ordered pair `(first q, last q)`.

2.4.3 Augmented List Comprehensions

List comprehensions are a distinguished feature of several functional languages which can increase the expressiveness of functional programs. In Lazy ML, a list comprehension is of the form,

[*Expression* ;; *qualifier*;...; *qualifier*]

where each *qualifier* is either a *generator* or a *filter*. A *generator* takes the form,

pattern <- *list-valued expression*

The effect of a generator is to draw values for the names in the *pattern* from the list one by one in turn. Elements of the list which do not match the pattern are ignored. A filter is a boolean expression; only the bindings which make a filter take the value true are considered. For instance, consider the following list comprehension,

```
y = [3 * X ;; X <- A; X > 3]
```

if `A` is `[1,2,3,4,5,6]` then `y` is `[12,15,18]`.

List comprehensions are analogous to set comprehensions in mathematics. The above list comprehension is essentially equivalent to the set comprehension:

$$y = \{3 * X \mid X \in A \text{ and } X > 3\}$$

Furthermore, it can be rewritten as:

```
y = {3 * X | member(X, A) and greater(X, 3)}
```

where `member/2` and `greater/2` are predicates defined as usual in Prolog. Thus, it makes sense to allow predicates as qualifiers in list comprehensions. This leads to the *augmented list comprehensions* in I^+. Augmented list comprehensions are an extension to list comprehensions by which the qualifiers may be predicates defined in some **`logic`** classes.

The advantages of augmented list comprehensions are two-fold. First, it is sometimes more natural to describe a qualifier as a relation instead of a function. For example, suppose we want to write a function

`separate(L)` which returns a list of tuples `(L1,L2,L3)` such that the concatenation of `L1`, `L2` and `L3` is `L`. That is,

```
separate([1,2])=
[([],[1,2],[]),([],[1],[2]),([1],[2],[]),([1],[],[2]),[1,2],[],[])]
```

It is convenient to define `separate(L)` using augmented list comprehensions as:

```
separate(L)=
[(L1, L2, L3);; logic::append(L12, L3, L); logic::append(L1, L2, L12)]
```

where `append/3` is defined in **`logic`** in the same way as in Prolog. Note that the order of the two `append`s is important here. If they are swapped, `separate(L)` cannot terminate since the length of `L12` will grow infinitely.

The second advantage is that augmented list comprehensions provide a functional interface to access databases represented by Prolog clauses. For example, suppose that there is a number of objects of the class `person` defined in Program 1, we can construct a function `information` to display the information of a person `P` as:

```
information(P)=[(N, ID, G);;P::name(N); P::id(ID); P::gender(G)]
```

`information(P)` returns a tuple `(N, ID, G)` in a list where `N`, `ID` and `G` are the name, identity number and gender of the person `P` respectively. Carrying on this fashion, we can write a function `all_persons` to show the information of every person in a list:

```
all_persons=[Info;;logic::object_of(person,P);[Info] <- information(P)]
```

2.5 Exploiting Parallelism in *I*+ Programs

2.5.1 Inter-Object Parallelism

The kind of message passing we introduced previously implements a strict caller/callee semantics of ordinary procedure calls. For instance, in the following message call,

```
...,O::p, G,...
```

The subgoal `G` cannot be tested until the result of testing `p` is returned from the object `O`. Such kind of message call is called the *synchronous-mode* message passing in which the sender object must wait for the reply from the receiver object before continuing execution. There is another mode of message passing called *asynchronous-mode*. In asynchronous-mode, the sender object sends a request to the receiver object without waiting for the reply. When the reply is available, it is stored in a data structure internal to the sender. At a future time, the sender can collect the reply by accessing the data structure. If the reply is still not available, the sender waits for it. This mode implements a lazy strategy called *wait by need* for invoking methods.

To distinguish between these two modes of message passing, *I*+ uses `O>>p` to denote the initialization of an asynchronous request `p` in the object `O`. To collect the reply of `O>>p`, use `O??p`. For example,

```
...,O>>m(X),A,B,O??m(X),c(X),...
```

Subgoals `A` and `B` are executed in parallel with the subgoal `m(X)` in the object `O`. The result of `O>>m(X)` is collected by asking `O??m(X)`. If the result is ready, then `c(X)` will be tested; otherwise the process

executing `O>>m(X)` is suspended until the result is available. Such a decomposition of a method call into a requesting phase and an answering phase enables the parallel execution of activities in multiple objects. This kind of parallelism is often referred to as the *inter-object parallelism* in some parallel object-oriented languages [5].

A good example to demonstrate the use of inter-object parallelism is the parallel version of quicksort. A class of objects called `sorter` is defined as:

```
logic sorter(&THRESHOLD)
world sort/2
family seq_qsort/2, par_qsort/2, split/4
implementation
    sort([],[]).
    sort(X,Y):- length(X)>&THRESHOLD, par_qsort(X,Y).
    sort(X,Y):- length(X)=<&THRESHOLD, seq_qsort(X,Y).
    /* sequential quicksort */
    seq_qsort([H|T],Y):-split(H,T,A,B), seq_qsort(A,A1),
                        seq_qsort(B,B1),
                        append(A1,[H|B1],Y).
    /* parallel quicksort */
    par_qsort([H|T],Y):-split(H,T,A,B),
     /*create a new sorter on the right hand side machine */
                O=create(sorter(&THRESHOLD, Right)),
                O>>sort(B, B1), sort(A,A1),
                O??sort(B,B1), append(A1,[H|B1],Y).
    /* a method for splitting a list into two sublists */
    split(_,[],[],[]).
    split(H,[A|X],[A|Y],Z) :- A =< H, split(H,X,Y,Z).
    split(H,[A|X],Y,[A|Z]) :- A > H, split(H,X,Y,Z).
class_end
```

Program 9 A parallel quicksort program in I^+

The **world** accessible method `sort(X,Y)` checks whether `length(X)>&THRESHOLD`. If it is, `sort(X,Y)` calls a parallel quicksort procedure `par_qsort(X,Y)` which exploits parallelism by invoking another "sorter" to sort the sublist `B`; otherwise `sort(X,Y)` calls the sequential quicksort procedure `seq_qsort(X,Y)`.

Asynchronous-mode message passing is an useful communication primitive. Synchronous-mode message passing can be implemented by asynchronous-mode message passing:

```
O::P :- O>>P, O??P.
```

Asynchronous-mode message passing is not limited in calling logic methods. It is also applicable to functional methods. The interpretation of `O>>g` in the context of functional methods is that it is a boolean function. `O>>g` returns true if the function `g` is successfully invoked in the object `O`; otherwise it returns false. The result of evaluating `g` is obtained by evaluating `O??g`. Thus, a proper use of asynchronous-mode message passing in function evaluation is of the form:

```
... if O>>g then ...O??g...else...
```

2.5.2 Intra-Object Parallelism

Intra-object parallelism corresponds to the capability of executing more than one method concurrently in a single object. This kind of parallelism has two purposes: (i) to ensure that each activity will be run in turn so that no one needs to wait too long for serving, and (ii) to avoid deadlocks between method calls. Fig. 3 illustrates a deadlock situation.

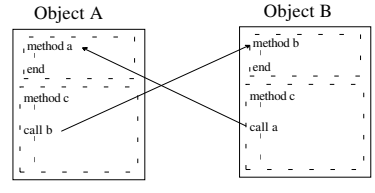
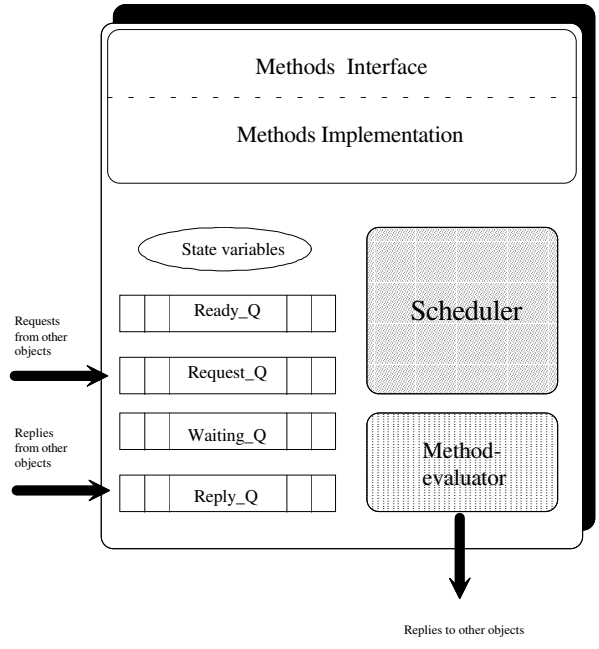


Fig. 3 A deadlock situation

In Fig. 3, suppose that both object A and B execute the `call` statement at about the same time. Without intra-object parallelism, the single activity of each object will be hanged at the `call` statements. On the

contrary, with intra-object parallelism, method `a` can be invoked at the same time that `call b` is being executed and so is true for method `b` at `call a`. A similar situation arises when the execution of a method needs to call (through message passing) another method of the same object.

In order to support intra-object parallelism, objects should behave like processors in multiprocessing systems. We call the execution of a method an *activity* inside the object. The activities/objects relationship here is analogous to the processes/processors relationship. Thus, an object can be treated as a virtual processor and has the components shown in Fig. 4 in order to schedule multiple activities inside the object.



The *static* part of an object contains the `interface` and the `implementation` of the methods. The *dynamic* part comprises the following components:

`State variables`:	They are present only in **`logic`** objects but not **`functional`** objects.
`Scheduler`:	It schedules the activities of the object using the round-robin policy.
`Ready_Q`:	A queue to store the activities that are ready to be executed.
`Waiting_Q`:	A queue to store the activities that are waiting for some conditions. Such a condition may be a reply from another object or a semaphore.
`Request_Q`:	A queue to hold requests (i.e. method invocations) from other objects.
`Reply_Q`:	A queue to hold replies (i.e. results of method invocations) from other objects.
`Method-evaluator`:	It is for evaluating methods. Two types of method-evaluator exists: one for logic methods and the other for functional methods.

Fig. 4. The configuration of an object with intra-object parallelism

Normally, I^+ maximizes intra-object parallelism by executing all ready activities of an object concurrently. Sometimes, however, it is necessary to execute a method (an activity) sequentially. For instance, to achieve mutually exclusive access to some state variables. One can specify a method to be executed sequentially by putting a `#` in front of the method's name in its declaration. For example, in the following program fragment:

```
...
world p/2, q/4, #r/3, #s/1 ...
...
```

Methods `p/2` and `q/4` are executed concurrently but methods `r/3` and `s/1` are executed sequentially.

2.5.3 Active Objects

Until now, objects are *passive*. That is, an object is activated only when its methods are invoked by other objects. The degree of concurrency can be further improved by incorporating *active objects*. An active object is an object that has its own activity once it is created. It does not rely on the activation by other objects. Objects of a particular class is made active by including an initialization method with the same name as that of the class in the class's definition. For instance, in the following class definition,

```
logic menu(...&LAYOUT...)
      ...
family menu() ...
      ...
implementation
menu :- display(&LAYOUT).
      ...
class_end
```

Program 10 The definition of class `menu`

The method `menu()` will be executed automatically once an object of class `menu` is created. Here, the initializing work is to display the menu according to the given layout specification `&LAYOUT`. Also since `menu()` is declared as a **`family`** method, all descendants of `menu` will have `menu()` as their initializing method.

In fact, supporting active objects is reasonable even without considering the issue of concurrency because autonomous objects are common in real life.

3 Applications

One of the well-known applications of logic programming and functional programming is in the area of artificial intelligence (A.I.), mainly due to the declarative nature of these two programming paradigms. I^+ incorporates them by providing an object-oriented framework so that, in addition to the advantages offered by these two declarative paradigms, we can model and solve problems in an object-

oriented approach to attain natural models and solutions to problems. Furthermore, the use of inheritance allows programmers to define the behaviors of objects in an incremental way.

On the other hand, I^+ is a distributed programming language since it supports computation by multiple objects which communicate by message passing rather than shared variables and is implemented by geographically separated networks of communicating processes. Thus, I^+ is a language for distributed A.I. programming. A number of applications have been written in I^+[25]. They include:

- Modeling of a state space search
- Solutions to the N-queen problem
- Object-oriented modeling of a university database
- A simple medical expert system

4 Implementation

To examine the correctness and practicality of our design, a prototype of I^+ has been built on a network of DEC/ULTRIX[10] workstations using the C programming language[22], Quintus Prolog[32] and Lazy ML (LML) [1].

The implementation model has two major components: an I^+-to-Prolog translator and an I^+-to-LML translator. The input to the two translators is an I^+ program. The function of the I^+-to-Prolog translator is to convert definitions of **`logic`** classes in the program into a group of Prolog modules. Similarly, the I^+-to-LML translator transforms definitions of **`functional`** classes into a group of LML modules. Both the translators have to perform the following steps in the translation process:

- Lexical and syntax analysis
- Check for semantic errors such as "referring to undefined parent classes"
- Construct a class table to hold all relevant information of class definitions

- For each class, determine all inherited state variables and methods from its parent classes and solve the conflicts between them

Message passing is implemented by sockets which are a kind of inter-process communication provided by 4.3BSD that provides communication between processes on a single system and between processes on different systems. Among the three types of communication protocols provided, the TCP/IP protocol is used because a TCP/IP based network communication package is supported in Quintus Prolog. This package implements a stream socket which provides for bi-directional, reliable, sequenced and unduplicated flow of data without message boundaries. Communication between processes is modeled by the client/server relationship.

5 Conclusion

We have designed a multiparadigm language I^+ which aims to integrate three programming paradigms: logic, functional, and object-oriented. Our integration techniques are based on classifying an object as an instance of either a **`logic`** class or a **`functional`** class. Methods of an object are defined and computed according to which language class the object belongs to. In our view, such kind of integration has an obvious advantage for building a multiparadigm language: it is conceptually simple to extend the system by a new paradigm, by adding a new language class for that paradigm to the class hierarchy. Thus, our approach favors the implementation of a multiparadigm system using an object-oriented language.

As pointed out in [37], harmonious integration of multiple paradigms has often failed at the language level because of the unavoidable divergence between paradigms. Integration of paradigms in I^+ is done at the class level with looser coupling among paradigms. More precisely, I^+ comprises an interface of the logic and the functional paradigms in an object-oriented framework. Compared with some other

multiparadigm approaches, I^+ is characterized by integrating the three paradigms with the capability of exploiting multi-level parallelism. See Table 2 for a brief summary.

	Logic	Functional	Object-Oriented	Single-level Parallelism	Multi-level Parallelism
LogiC++[39]	√		√		
Intermission[19]	√		√		
OOPP[41]	√		√		
CPU[27]	√		√	√	
DLP[11]	√		√	√	
OLPSC[15]	√		√		
KSL/Logic[17]	√		√	√	
Orient84/K[18]	√		√	√	
Vulcan[23]	√		√	√	
Bridge[21]	√		√		
PROOF[40]		√	√		√
FLC[4]		√	√		
CLOS[14]		√	√		
HOPE[9]	√	√			
FUNLOG[34]	√	√			
F*[28]	√	√			
LEAF[3]	√	√			
Applog[6]	√	√			
LIFE[2]	√	√	√	√	
UNIFORM[20]	√	√	√		
G[31]	√	√	√		
L&O[26]	√	√	√		
I^+	√	√	√		√

Table 2 A summary on the paradigms involved and the level of parallelism of some multiparadigm languages

In addition to the design, we have implemented a prototype of I^+ on an UNIX based network. This prototype allows us to test our ideas by actual examples and to discover the bugs in our design. We find that it is feasible and also worthy to integrate the three paradigms in a single environment, since such integrated language has improvements in many aspects. Fig. 5 shows how each paradigm contributes its distinguished properties to another paradigm in I^+.

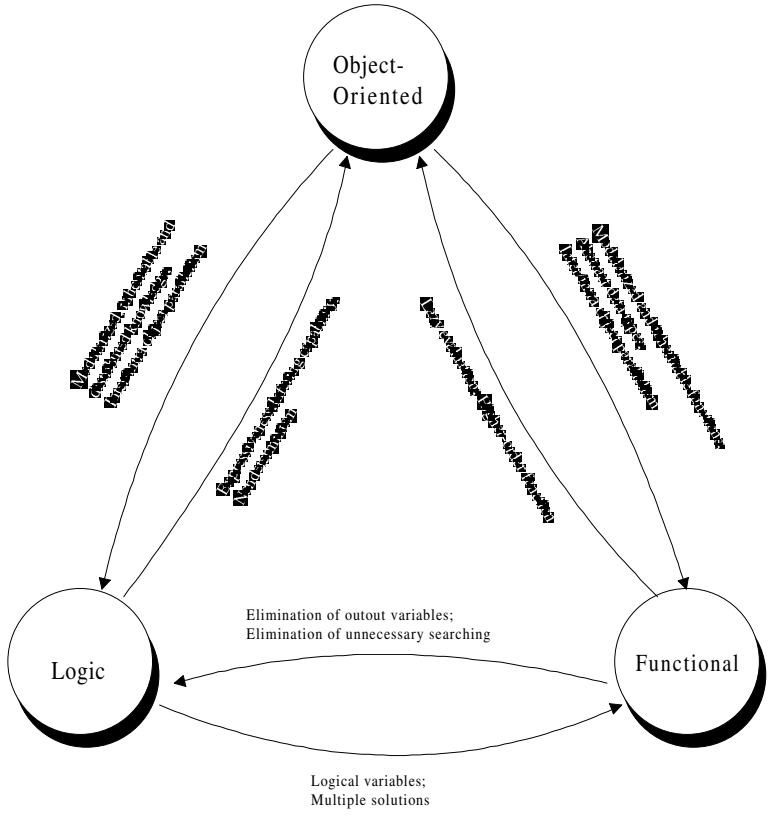


Fig. 5 Cooperation between the three paradigms in I^+

6 Future Work

Future development of I^+ can be in the following directions:

An I^+-to-C++ Compiler

A common practice to implement a new language is to construct a preprocessor which translates programs written in the new language to equivalent programs written in another well developed language. In our implementation, I^+ is translated to two languages Prolog and LML for the purpose of prototyping. The resultant system is not efficient enough and not quite portable. A better approach is to translate I^+ to a single language which is efficient and popular. In addition, the target language is preferably an object-oriented language such that the two language classes can be realized naturally. C++ is our possible choice.

Programming Environment

An integrated programming environment for I^+ is desired. Our intended environment should include a file system, an editor, a debugger/tracer, a system set-up manager, and of course a compiler/interpreter. In addition, special facilities for monitoring the execution of objects on multiple machines are required to support dynamic adjustment of the granularity of parallelism and load balance.

MIMD Computers

An interesting and challenging work is to implement I^+ on a MIMD machine. The most fundamental problem facing parallel computing at present is to find desirable programming models [36]. Such models should be as high a level as possible, with maximum abstraction from hardware. I^+ is a feasible programming model for MIMD machines since (i) it is natural to distribute autonomous objects to different processors with independent instruction streams, and (ii) the declarative nature of the logic and the functional components provides a convenient way to exploit fine-grain parallelism.

References

1. Lennart Augustsson and Thomas Johnsson. *Lazy ML user's manual*. August 1992.

2. H. Ait-Kaci. An overview of LIFE. In J.W.Schmidt and A.A.Stogny, editors, *Next Generation Information System Technology*, pp.42-58. Springer-Verlag, 1991.
3. Barbuti, R., Bellia, M., Levi, G., and Martelli, M. Logic, Equations and Functions. In DeGroot, D., and Lindstrom, G., editors, *Logic Programming : Relations, Functions, and Equations*. Prentice-Hall, Englewood Cliffs, N.J., 1985, pp. 201-238.
4. M. Beaven, R. Stansifer, and D. Wetklow. A functional langauge with classes. *LNCS 507*, 1991, pp. 364-370.
5. Antonio Corradi and Letizia Leonardi. Parallelism in Object-Oriented Programming Language. In *Proceedings of the 1990 International Conference on Computer Languages*, pp. 271-280.
6. Shimon Cohen. The Applog Language. In DeGroot, D., and Lindstrom, G., editors, *Logic Programming : Relations, Functions, and Equations*. Prentice-Hall, Englewood Cliffs, N.J., 1985, pp. 239-261.
7. M.H.M. Cheng, M.H. van Emden, and B.E. Richards. On Warren's Method for Functional Programming in Logic. University of Victoria.
8. A. Davison. Polka: A Parlog object oriented language. Internal report, Dept. of Computing, Imperial College, London 1988.
9. Darlington, J., Field, A.J., and Pull, H. The unification of functional and logic languages. In DeGroot, D., and Lindstrom, G., editors, *Logic Programming : Relations, Functions, and Equations*. Prentice-Hall, Englewood Cliffs, N.J., 1985, pp. 37-70.
10. Digitial Equipments Corp. *ULTRIX Supplementary Document : Vol. 2 Programmer*. Digitial Equipments Corporation, 1990.
11. Anton Eliens. *DLP A Language for Distributed Logic Programming Design, Semantics and Implementation*. John Wiley & Sons, 1992.
12. Goguen, J.A. and Mesegner, J. Unifying Functional, Object-Oriented, and Relational Programming with Logical Semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*. MIT Press, 1987, pp. 417-478.
13. A. Goldberg and D. Robson. *SMALLTALK-80 - The Language and Its Implementation*. Addison-Wesley. Reading, Massachusetts. 1990.
14. R.P. Gabriel, J.L. White, and D.G. Bobrow. CLOS: Integrating Object-Oriented and Functional Programming. *Comm. ACM*, Vol.34, No.9, Sept.1991, pp. 28-38.
15. J.S. Hodas and D. Miller. Representing Objects in a Logic Programming Language with Scoping Constructs. In *Proceedings of the Seventh International Conference of Logic Programming*, MIT Press, 1990, pp. 511-528.
16. Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, Vol. 21, No.3, pp.359-411, Sept. 1989.
17. Ibrahin, M.H. and Cummins, F.A. KSL/Logic: Integration of Logic with Objects. In *Proc. of International Symposium on Logic Programming*, IEEE, 1990, pp. 228-235.
18. Ishikawa, Y. and Tokoro M. Orient84/K : An Object-Oriented Concurrent Programming Language for Knowledge Representation. In Yonezawa, A., and Tokoro, M., editors, *Object-Oriented Concurrent Programming*. MIT Press, 1987, pp. 129-158.
19. Kahn, K.M. Intermission-Actors in Prolog. In Clark, K.L. and Tarnuland, S. A. *Logic Programming*. Academic Press, 1982, pp. 213-230.
20. Kahn, K.M. UNIFORM : A language based upon unification which unifies (much of) Lisp, Prolog, and Act 1. In DeGroot, D., and Lindstrom, G., editors, *Logic Programming : Relations, Functions, and Equations*. Prentice-Hall, Englewood Cliffs, N.J., 1985, pp. 411-438.
21. Koschmann, T. and Evers, M.W. Bridge the Gap Between Object-Oriented and Logic Programming. *IEEE Software*, Vol. 5, No.5 ,1988, pp. 36-42.

22. Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language 2nd. Edition*. Prentice-Hall, 1989.
23. Kahn, K., Tribble, E., Miller, M., and Bobrow, D. Vulcan: Logical Concurrent Objects. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*. MIT Press, 1987, pp.75-112.
24. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1984.
25. K.W. Ng and C.K. Luk. *The I+ Programming Language*. Technical Report, in preparation, Dept. of Computer Science, The Chinese University of Hong Kong.
26. Francis G. McCabe. *Logic and Objects*. Prentice Hall, 1992.
27. P. Mello and A. Natali. Programs as collections of communicating Prolog units. *LNCS 213*, Springer-Verlag, 1986, pp. 274-288.
28. Narain, S. Lazy evaluation in logic programming. In *Proc. of International Symposium on Logic Programming*, IEEE, 1990, pp. 218-227.
29. K.W. Ng and C.K. Luk. The Design of a Multiparadigm Programming Language: *I*. *Microprocessing and Microprogramming*, 37:171-174, 1993.
30. K.W. Ng and C.K. Luk. *I* : An Integrated Programming Language. In *Proceedings of IEEE TENCON'93*, pp. 382-385, International Academic Publishers, 1993.
31. Placer, J. The multiparadigm language G. *Computer. Lang.* Vol.16, No.3/4 (1991), 235-258.
32. Quintus Corporation. *Quintus Prolog 3.1 Reference Manual*. Quintus Corporation, 1991.
33. Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley. 1986.
34. Subrahmanyan, P.A., and You, J. -H. FUNLOG : A computational model integrating logic programming and functional programming. In DeGroot, D., and Lindstrom, G., editors, *Logic Programming : Relations, Functions, and Equations*. Prentice-Hall, Englewood Cliffs, N.J., 1985, pp. 157-197.
35. M. van Emden and K. Yukawa. Logic programming with equations. *J. of Logic Programming*, 4(4), Dec. 1987.
36. D.J.Wallace. Massively Parallel Computing: Status and Prospects. *Technical Report, Edinburgh Parallel Computing Centre*. University of Edinburgh, 1992.
37. Peter Weger. Concept and paradigms of object-oriented programming. *OOPS Messager*, Vol. 1, Number 1, pp. 7-87, Aug.90.
38. P.H. Winston and B.K.P. Horn. *LISP 3rd edition*. Addison-Wesley, 1989.
39. Shaun-inn Wu. Integrating Logic and Object-Oriented Programming. *OOPS Messager*, Vol. 2, No.1, pp.28-37, Jan. 91.
40. Yau, S.S., Jia, X., and Bae, D.-H. PROOF : A Parallel Object-Oriented Functional Computation Model. *J.Parallel Distrib. Comput.* 12 (1991), 202-212.
41. C. Zaniolo. Object-Oriented Programming in Prolog. In *Proceedings of 1984 IEEE Symposium on Logic Programming*, pp. 265-270.

Summary

I^+ is a multiparadigm language which integrates the three major programming paradigms: object-oriented, logic and functional in a single environment. It is an enhanced descendent from another multiparadigm language *I*. I^+ has an object-oriented framework in which the notions of classes, objects, methods, inheritance and message passing are supported. Nevertheless, methods in I^+ are different from those in imperative object-oriented languages in the sense that they are not specified as procedures but as *clauses* or *functions*. Thus the two declarative paradigms are incorporated at the method level of the object-oriented paradigm. In our view, such kind of integration has an obvious advantage for building a multiparadigm language: it is conceptually simple to extend the system by a new paradigm, by adding a new language class for that paradigm to the class hierarchy. Thus, our approach favors the implementation of a multiparadigm system using an object-oriented language. In addition, two levels of parallelism, *inter-object* and *intra-object* parallelism, are exploited in I^+ programming. Therefore I^+ is a multiparadigm language for object-oriented declarative programming as well as parallel programming.

Besides the design, we have implemented a prototype of I^+ on an UNIX based network. This prototype allows us to test our ideas by actual examples and to discover the bugs in our design. We find that it is feasible and also worthy to integrate the three paradigms in a single environment, since such integrated language has improvements in a number of aspects.