A Survey of Languages Integrating Functional, Object-oriented and Logic Programming

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

Abstract

Functional, object-oriented and logic programming are widely regarded as the three most dominant programming paradigms nowadays. For the past decade, many attempts have been made to integrate these three paradigms into a single language. This paper is a survey of some of this new breed of multiparadigm languages. First we give a succinct introduction to the three paradigms. Then we discuss a variety of approaches to the integration of the three paradigms through an overview of 24 multiparadigm languages. All possible combinations of the three paradigms, namely logic + object-oriented, functional + logic, functional + object-oriented, and object-oriented + logic + functional, are considered separately. For the purpose of classification, we have proposed a design space of programming languages called the FOOL-space.

Keywords: Functional, object-oriented, logic, multiparadigm, FOOL-space

1. INTRODUCTION

The *functional*, *object-oriented*, and *logic* paradigms are probably the three most representative programming paradigms nowadays. Briefly, the object-oriented paradigm is distinguished by its suitability to construct complex software by modeling the behavior and interaction of objects in the problem domain. The logic and functional paradigms are distinguished by their abilities to provide a framework in which declarative programs can be written, in the forms of relations and functions respectively.

Multiparadigm languages are a kind of languages that allow programmers to design and implement their programs based on two or more paradigms. They are considered desirable in because of the following factors:

- Programmers can choose the most appropriate paradigm for a particular problem 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 is reusable in the multiparadigm language.
- It is natural to model real-world objects using multiparadigm languages since the former can be viewed as loosely coupled distributed systems with multiparadigm cooperating subsystems.

In the past decade, there have been many attempts at constructing multiparadigm languages with the intention of capturing the merits of the three paradigms. For instance, [Ale93] reports that up to March, 1993 there are at least 50 systems or languages concerning the combination of object-oriented and logic paradigms. However, there is a lack of surveys of the integration of the three paradigms.

This paper is a survey of some of the existing multiparadigm languages and is organized as follows. Section 2 summarizes the elements of the three paradigms. Section 3 discusses a variety of approaches to the integration of the three paradigms through an overview of a selection of 24 representative multiparadigm languages. Section 4 is a conclusion.

2. A REVIEW OF THE OBJECT-ORIENTED, LOGIC AND FUNCTIONAL PARADIGMS

This section serves as a review of the essential characteristics of the three paradigms. We recommend the readers to refer to the references mentioned accordingly for an in-depth study of the three paradigms.

2.1 The Object-Oriented Paradigm

2.1.1 Basic Components

The three basic components of object-oriented programming are *Objects*, *Classes* and *Inheritance*:

Objects: An object is a collection of *operations* and a *state* that remembers the effect of the operations. Communication between objects is done by *message passing*. The state is hidden from the outside world and is accessible only to the object's operations which are invoked by message calls from some objects (an object may send a message call to itself). The internal state of an object is represented by a set of local variables called *instance variables*. The operations of an object are usually called *methods*.

Classes: Classes serve as templates from which objects may be created. A class has two parts: *interface* and *body*. The interface part specifies what operations are available in objects created from the class. The body part specifies the implementation of the operations in the interface part and the representation of the state. We may treat a class as defining the behavior common to all objects of the class.

Inheritance: Inheritance is a mechanism for composing the behavior of a class based on that of its parent classes. Subclasses of a class inherit the operations of their parent classes and may add new operations and new instance variables. With inheritance, classes can be organized into a hierarchy of subclasses and superclasses.

2.1.2 Motivations

A fundamental shortcoming of the imperative paradigm is the unprohibited access of global variables from any part of a program. When a program is large, management of such global variables becomes difficult. Object-oriented programming is a successful solution to this problem by encapsulating each global variable in a module with a group of operations (i.e. an object) that have exclusive direct access to the variable. The scope rules of many object-oriented languages such as C++ [Str91], Smalltalk [Gol84] and ABCL [Yon90] ensure that the hidden variables of an object are only accessible by calling the operations exported in the interface part of the object. Thus *information hiding* and *data encapsulation* are achieved.

Furthermore, object-oriented programming offers *high reusability* of software. Traditional procedure-oriented libraries are actually collections of unrelated software components. It is the responsibility of library users to combine procedures from the library into programs. In a more systematic way, object-oriented libraries are collections of hierarchically organized classes from which objects may be created. Much of the work of component composition is inherent in the class hierarchy. The behavior of a class can be reused by creating instances (objects) and by defining subclasses that modify its behavior.

2.1.3 Concurrent Object-Oriented Programming

Objects of object-oriented programs are like members of a society. Objects execute independently of each other and communicate by message passing. This is called *inter-object parallelism* [CL90]. The degree of inter-object parallelism depends on the mode of communication between objects. Basically, there are two modes of communication: *synchronous* and *asynchronous*. Within each object, several threads of execution can be run concurrently by CPU time-multiplexing. These threads are restricted to be executed within a particular object. Such kind of parallelism is called *intra-object parallelism* [CL90]. Introducing inter-object and intra-object parallelism into a single environment provides various degree of parallelism.

[YT87] is a collection of papers introducing some representative concurrent object-oriented languages.

2.1.4 Other Issues

[Weg90] is an excellent survey paper on the object-oriented paradigm. Besides the basic concepts we have presented here, the following topics are also discussed in that paper:

- The relationship between *types* and classes in object-oriented programming
- Different inheritance mechanisms in object-oriented programming

- A family of object-based languages
- Computational models and the semantics for object-oriented programs

2.2 The Functional Paradigm

2.2.1 Basic concepts

A program can be viewed as implementing a *mapping* from some input values to output values. In imperative programming, this mapping is implicit. Commands read input values, manipulate them and write output values. The commands of a program affect one another through the modification of variables accessible by them. Thus the relationships between commands may be very complicated or perhaps unmanageable even in a moderate size program.

The main idea of functional programming is to explicate the mapping of input values to output values. Basically, a functional program itself is a function (or a group of functions) which accepts input values and produces output values according to the definition of the function. Concepts like commands and global variables are no longer present in functional programs. Let us look at a simple example `fac(n)` which calculates the factorial of an integer `n` to distinguish the styles of imperative and functional programming. Using a Pascal-like language, `fac(n)` can be written as :

```
function fac(n : integer) : integer;
var
   result : integer;
begin
   result := 1;
   while n > 0 do
      begin
            result := result * n;
            n := n - 1;
      end;
   fac := result;
end;
```

We can catch a glimpse of the essence of imperative programming: explicit looping over variable with changing value and storing of useful values in variables. On the other hand, the definition of `fac` written in a functional language ML [Mil90] is:

```
   fac 0 = 1
|| fac n = n * fac (n - 1)
```

where `||` means "or". It specifies different cases of input parameters of the function.

Instead of using explicit looping, recursion is used. Also, we do not need extra variables to hold temporary values. In addition, the program appears very much like the mathematical definition of factorial.

Due to the side-effect free nature of functional programming, *referential transparency* is preserved in functional programs. A program is referentially transparent if every reference to a value (may be in the form of a function or an expression) is equivalent to the value itself no matter where or when the reference occurs. Referential transparency guarantees that "equals can be replaced by equals". For example, consider the following ML expression:

```
let
      y = f x
in y + y * y
```

The function application `f x` may be substituted for any free occurrence of `y` in the scope of the `let` expression. Thus `y+y*y` can be replaced by `f x + f x * f x` in the above expression. We cannot do this in imperative programming. For instance, consider the following Pascal-like program:

```
program changed;
var flag : boolean;
function f(n:integer):integer;
begin
   if flag then
      f := n
   else
      f := n * 3;
   flag := not flag
end;
begin
   flag := true;
   writeln(f(1)+f(2)); (* output 7 *)
   writeln(f(2)+f(1)); (* output 5 *)
end.
```

We get `7` for expression `f(1)+f(2)` but `5` for `f(2)+f(1)`. The problem is that the value of `flag` is modified by the destructive assignment statement `flag := not flag`.

Lambda calculus is the most fundamental and powerful mathematical system to capture the computational aspects of functions. Even though it is simple in syntax and semantics, it is expressive

enough to describe all functional programs. Therefore it always acts as an intermediate language between high-level functional languages and their low-level implementations. The readers can find a brief introduction to the lambda calculus in Chapter Five of [Mey90].

2.2.2 The characteristics of functional programs

Are functional programs just collections of functions and expressions? If the answer is "yes", then we can simply drop the assignment statement and any other side-effecting primitives in an imperative programming language to derive a functional language. If we do so, we will find that the result of such a derivation is not satisfactory due to the poor functional subset of most imperative languages. Most of the modern functional languages including ML [Mil90], Hope [DFP85] and Miranda [Tur90] have the following distinguishing features to support practical programming in a pure functional style:

- Higher order functions
- Lazy evaluation
- Pattern matching
- Strong typing
- Abstract data types

We recommend [Hud89], an excellent survey paper on functional programming, for a clear and concise description of the above terms. [FH88] is a comprehensive text on basic concepts and techniques in functional programming. [Jon87] is a definite reference for the implementation of functional languages.

2.2.3 The opportunity for parallelism

The possibility of parallel execution is an attractive feature of functional programming languages. Functional programs have great potential for parallel programming because of their side-effect free nature. By removing the assignment statement, functional programs need not be executed sequentially. At any moment, there are a number of reducible expressions in the program, and in principle they could all be reduced simultaneously. Usually, parallelism in functional programs is implicit in contrast to that in imperative programs in which additional language constructs like `cobegin...coend` and `fork...join` are required to coordinate the concurrent activities. No extra language construct is needed to write parallel functional programs. Functional programmers do not have to design a static task partition, guarantee mutual exclusion and synchronization or construct communication protocols between tasks. Nevertheless, as optimal mapping of processes to processors is a nontrivial task, a kind of meta languages called *parafunctional* programming languages has been developed. In a parafunctional programming language, annotations are included to explicitly control priorities of tasks and mapping of processes to processors. [Jon89] is a good introduction to parallel functional programming. [Szy91, Har91, Gla92] describe more recent work.

2.3 The Logic Paradigm

2.3.1 Relations and Logic Programs

As we mentioned above, imperative and functional programming are about implementing mappings from input to output. Logic programming is essentially about implementing *relations*. A mapping is a many-to-one relationship, whereas a relation is in general a many-to-many relationship. Therefore relations are more general than mappings and logic programming is potentially higher level than imperative and functional programming.

A logic program is a collection of *Horn clauses.* A Horn clause has the general form:

$$H \leftarrow B_1, \ldots, B_n$$

where H is commonly called the *head* of the clause and B_i's the *body*. Both H and B_i are simple assertions of the form `p(...)` where p is a relation name. Informally such a clause may be read as stating that H is true if $B_1, \ldots, B_n$ are all true. In the special case where $n=0$, the Horn clause states a fact that H is unconditionally true.

Computation consists of testing a given query $\leftarrow Q$. If we can infer from the clauses of the program that Q is true, then we say that Q succeeds. If we cannot infer that Q is true, then we say that Q fails. The testing of queries can be implemented by a technique for inference called *resolution* which is based on a variable binding mechanism named *unification* [Llo84].

Prolog [CM84] is the most popular logic programming language that exists today. Basically a Prolog program is a collection of Horn clauses, but in addition contains a number of features that are convenient for computer programming. In Prolog convention, $H \leftarrow B_1,...,B_n$ is written as `H:-`B_1`,...,`B_n`.`, $H \leftarrow$ is written as `H.` and $\leftarrow B_1,...,B_n$ is written as `:-` B_1`,...,`B_n`.`. For example, consider the following Prolog program:

```
append([], L, L).
append([H|T], L, [H|K]) :-
      append(T, L, K).
```

The notation `[H|T]` denotes the list with the first element `H` followed by a tail list `T`. The notation `[]` denotes the empty list. `append(X,Y,Z)` implements the relation: `Z` is the list resulting from the concatenation of lists `X` and `Y`. Two cases are considered in `append`. If the first list is empty then the result is the second list. Otherwise, the front element of the first list is made the front element of the list that results from appending the second list to the rest of the first list. Besides the purpose of list concatenation, `append` can be used to remove a given list from the front (or back) of another, or to find all ways of deconcatenating a given list since `append` implements a relation instead of a function.

[CM84] is widely selected as the fundamental text for learning Prolog. [Ste87] gives more advanced materials on Prolog and logic programming. For the theoretic foundations of logic programs, the readers should refer to [Llo84].

2.3.2 The opportunity for parallelism

Two kinds of parallelism (OR and AND parallelism) [Con87] can be exploited in logic programs. OR-parallelism refers to the parallel execution of different clauses of a procedure. For example, consider the following logic program :

```
g ← p, q, r
p ← a
p ← b
p ← c
r ← d
```

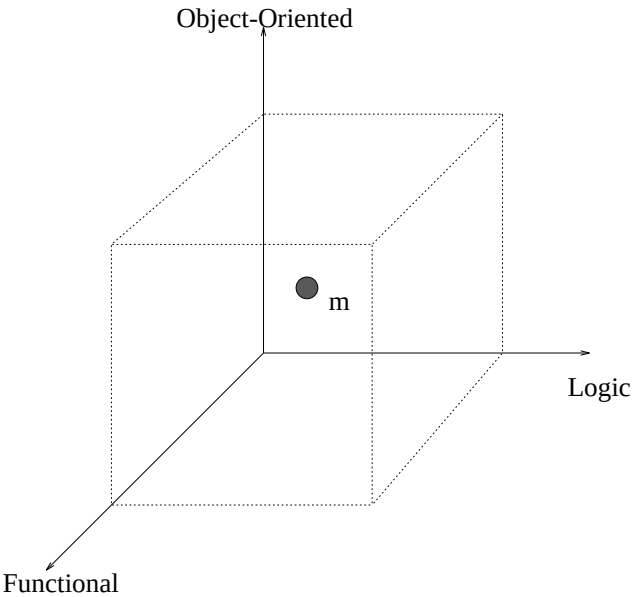
To solve the goal `g`, the three clauses with head `p` may be searched in parallel. When there is more than one way to solve a goal, multiple results can be obtained by OR-parallelism.

AND-parallelism corresponds to the parallel execution of goals in the body of a clause. For example, to solve the goal `g`, subgoals `p`, `q`, `r` may be solved in parallel. So, AND-parallelism aims at faster generation of a single result.

However, there are some problems associated with the AND/OR parallelism model. Unlimited OR-parallelism may result in an uncontrollable number of processes in order to find all solutions to a goal. Unrestricted AND-parallelism may result in a large amount of incompatible bindings of the logical variables involved. To solve these problems, the two major representatives of concurrent logic programming: Concurrent Prolog [Sha87] and Parlog [Gre87] have based their approaches on *committed choice non-determinism* and *restricted unification*. See [Sha89, Tak92] for a comprehensive survey of concurrent logic programming languages.

2.4 Summary

At first glance, it seems difficult to compare these three paradigms on a common base. However, we have endeavoured to make a comparison of them in Table 1 based on some issues related to every programming paradigm or language.



m is a multiparadigm language in the FOOL-space

Fig. 1 The FOOL-space

	Logic	Functional	Object-oriented
Basic representations	Relations represented by Horn clauses	Mathematical functions	Objects, classes, inheritance
Theoretical frameworks	First-order logic	Lambda calculus	Automata theory, equivalence relation, reflexive systems, fixed point theory
Problem-solving mechanisms	Resolution	Reduction of expressions	Method invocation by message passing
Values and types	Primitive data types; type-checking at run-time	Primitive + composite data types; strong typing is encouraged	Primitive + composite data types; strong typing is encouraged
Parameter-binding method	Through unification	Through pattern-matching	Many alternatives: call-by-value, call-by-name, call-by-reference
Evaluation policy	Eager evaluation	Lazy evaluation	Eager evaluation
Program expressiveness	Declarative	Declarative	Procedural
State	Reflected in the parameters of predicates; explicit modeling of state	Reflected in the parameters of functions; explicit modeling of state	Reflected in the values of global variables; implicit modeling of state
Higher-order capability	First-order	Higher order	Language dependent
Determinism	In theory: nondeterministic; sequential implementations: deterministic; parallel implementations: nondeterministic	Deterministic for both sequential and parallel implementations	Sequential : deterministic; parallel: may be nondeterministic
Efficiency of implementations	Poor due to the searching process	Poor due to lazy evaluation and the use of many intermediate composite value	Efficient

Table 1 A summary of characteristics of the logic, the functional and the object-oriented paradigms

	Subclasses	Subtypes
PROOF	√	
CLOS	√	
FLC		√
FOOP	√	√

Table 2 The support of subclasses and subtypes in the four functional + object-oriented approaches

3. AN OVERVIEW OF SOME MULTIPARADIGM LANGUAGES

In this section, we examine some existing ways to integrate the three paradigms. We consider the four possible combinations of them, namely logic + object-oriented, object-oriented + functional, logic + functional, and object-oriented + logic + functional, separately. For each combination of paradigms, we give descriptions of some representative approaches to the combination and a discussion. Each one of the multiparadigm approaches can be treated as an element in a design space of programming languages, called the **FOOL**-space (**F**unctional, **O**bject-**O**riented and **L**ogic). The position of a multiparadigm language in the FOOL-space indicates the *dominant* paradigm of the language. See Fig. 1. We say that a multiparadigm language M has a dominant paradigm D if M is formed by injecting a number of concepts or elements of some paradigms other than D to a D *induced* language. A paradigm induced language is a language that is designed and implemented based upon the notions of a paradigm [Hai86]. Sometimes, it may not be possible to identify the dominant paradigm of a multiparadigm language if all the component paradigms have equal significance in the integration.

3.1 Logic + Object-Oriented

3.1.1 LogiC++

As indicated by its name, LogiC++ [Wu91] is based on C++. The structure of a LogiC++ program is very similar to that of a C++ program except that methods in a LogiC++ program can be represented by Prolog clauses. The following is a sample program in LogiC++,

```
class manager;
class employee{
   private:
      manager *his_manager;
   public:
      employee(void)
         { his_manager = 0; };
   employee(manager &m)
         { his_manager = &m; };
   methods
      request(Subject) :-
his_manager.give_me_approval(Subject)
.}
class manager : public employee{
   private:
      methods
      unimportant(domestis_trip).
      unimportant(presentation).
      unreasonable(double_salary).
   public:
      manager(void) : employee(){};
      manager(manager &m) :
                  employee(m){};
      methods
      give_me_approval(Subject) :-
         unimportant(Subject).
      give_me_approval(Subject):-
         not unreasonable(Subject).
}
```

Two classes `manager` and `employee` are defined in the above program. `employee` is a super class of `manager`. For the class `employee`, a public method `request/1` is defined. For the class `manager`, two private methods `unimportant/1` and `unreasonable/1` and a public method `give_me_approval/1` are defined as a local knowledge base. The keyword `methods` signifies that the following operations would be defined by Prolog clauses.

Programs written in LogiC++ are translated by a preprocessor to some C++ programs which can then be compiled by a C++ compiler. Each clause of a method is compiled into a function in C++. The C++ functions for all the clauses with the same head predicate are grouped into a single function in which each clause is denoted by an alternative of a branch statement. In the body of a clause, subgoal calls are translated into function calls. The advantages of this approach are that implementation can be efficient and portable to many platforms. However, since all the methods are declared and checked at compile-time, dynamic modification of knowledge is impossible in LogiC++.

3.1.2 Intermission

Prolog is weak in data-abstraction. For example, a Prolog program which is written for sorting integers cannot be used to sort elements of other data types. Intermission [Kah82] solves this problem by embedding an object-based entity

called *actor* into Prolog. An actor is a computational entity that combines in a single unit both program and data. Computation is performed only by sending messages. An actor consists of a *script* and *acquaintances*. The script determines the action to be performed in responding to an incoming message. Acquaintances are the set of other actors that the actor knows. An actor can only send messages to another actor which is either its acquaintance or is named in the incoming message. In Prolog, an actor is represented by a list whose first element is its script and the rest of the list are its acquaintances. For example, consider the following implementation of lists as actors:

```
(list, FIRST, REST)
```

`list` is the script, `FIRST` and `REST` are acquaintances. The message passing mechanism is implemented by a relation sent:

```
sent(Target, Message, Result)
```

`Target` is the name of the actor where the `Message` should be sent. `Result` is the outcome of sending `Message` to `Target`. The following example demonstrates how to abstract a list of integers by an actor with three acquaintances: the first element, the last element and the difference between successive elements. The messages `first` and `rest` can be defined as follows:

```
sent((nlist, BEGIN, END, INCREMENT),
first, BEGIN).
sent((nlist, BEGIN, END, INCREMENT),
rest, (nlist, NEW_BEGIN, END,
INCREMENT)) :-
 sent(BEGIN,(+,INCREMENT),NEW_BEGIN).
```

Sending the `first` message to a `nlist` actor causes it to return its first element `BEGIN`. The `rest` message causes a `nlist` actor to return another `nlist` actor whose first element is its first element plus the `INCREMENT` as its tail sublist.
The addition is performed by actors as follows:

```
sent(NUMBER, (+, ANOTHER), RESULT) :-
   integer(NUMBER),
   integer(ANOTHER),
   RESULT is ANOTHER + NUMBER.
```

The predicate `integer(N)` succeeds if `N` is of the integer data type.

There are some problems with Intermission. The syntax is awkward and verbose mainly due to the use of explicit constructors and selectors instead of pattern matching. Also, Intermission has no constructs for structuring programs which is one of the primary purposes of object-oriented programming. The most serious problem of Intermission is that the implementation of actors on top of Prolog is slow and inefficient. To achieve better performance, they should be incorporated at a lower level of implementation (possibly at the same level as functors, symbols and numbers).

3.1.3 Object-Oriented Programming in Prolog (OOPP)

In [Zan84], syntactic notations are introduced in Prolog to allow the specifications of objects, methods, messages and inheritance. An object declaration has the form:

object with *method-list*

where *object* is an arbitrary predicate and each method in *method-list* is a Prolog clauses. For example,

```
reg_poly(N, L) with [(perimeter(P) :-
P is N * L);
what_is_it(a_reg_polygon)]
```

declares `reg_poly` to be an object with `perimeter/1` and `what_is_it/1` as its associated methods. The application of methods to objects is specified by messages using the infix operator ":". For example,

```
reg_poly(5, 10) : perimeter(X)?
```

returns `X=50`.

Inheritance is declared by the operator `isa`. For example, to declare that both squares and pentagons are regular polygons, we write:

```
square(L) isa reg_poly(4, L).
pentagon(L) isa reg_poly(5, L).
```

The implementation is built on top of UNSW Prolog, with no modification to the interpreter or compiler. Though OOPP offers object-oriented concepts such as objects, methods and messages, it

does not support objects with state. Also, the `isa` relation does not fully exploit the power of inheritance in two senses. First, if more than one superclass has a definition of a method only one of them will be used. This scheme discards all but one solutions to a problem. Second, we can only define a subclass by refining a class's behavior rather than modifying a class's behavior in general. For example, definitions of methods in a class cannot override those in its superclasses.

3.1.4 Communicating Prolog Unit (CPU)

In CPU [MN86], a Prolog program is organized as a collection of objects, which are referred to as Prolog-units (P-units). Each P-unit represents a chunk of knowledge about a particular domain in the form of Prolog clauses. Communication between units is done by the predicate

```
send(Destination, Goal, Answer)
```

where `Destination` is the unit by which `Goal` must be evaluated. The result is collected in `Answer`. A kind of units, called meta-units is introduced to define and handle the interaction between P-units. The main task of a meta-unit is to decide how to solve predicates in a P-unit.

A P-unit may have multiple instances, called processes for serving requests from other P-units. Synchronization among processes is achieved by the synchronization clause

```
entry(...), accept(...) :-
        body(...).
```

When a goal in a send statement unifies with the `entry` part of the clause's head, the evaluation of the `body` is postponed until the `accept` part of the head is satisfied.

The main idea of this approach is to build complex software systems by combining a set of different Prolog modules or P-units through a meta-level specification with the possibility of parallel execution of the P-units. Nonetheless, CPU only focuses on the process part of object-oriented programming, inheritance is not supported.

3.1.5 Distributed Logic Programming (DLP)

The language DLP [Eli92] is an extension of Prolog with object declarations and statements for the dynamic creation of objects, communication between objects ,and the assignment of values to non-logical instance variables of objects. In general, an object declaration in DLP has the form:

```
object name{
      use objects
      var variables
      clauses
}
```

The effect of the `use` declaration is to include the clauses of `objects` into the object `name`. The variables declared by `var` are non-logical variables that may be modified by the assignment statement. For example, the definition of an object `travel` is:

```
object travel{
 use library
 var city[Tokyo,Peking,HongKong]
 reachable(X) :- member(X, city).
 add(X) :- append([X], city R),
           city := R.
}
```

Inheritance of the non-logical variables of objects is accomplished by including the declaration `isa` in the object declarations. The declared object then contains a copy of all the non-logical variables of the objects mentioned in the `isa` list. Clauses are inherited from objects by including the declaration `use`. The following kinds of statements are provided in DLP:

`v := t` /* to assign the value of the term `t` to the non-logical variable `v`*/
`O = new(c(t))` /* to create an active instance `O` of the object `c`*/
`O!m(t)` /* to call the method `m(t)` in the object `O`*/

To be a distributed language, DLP supports the following features:

- Active and passive objects
- Synchronous and asynchronous communication by rendezvous
- Distributed backtracking

Though non-logical variables and assignment statements are useful in state modeling, they cause difficulties in the formulation of a declarative semantics for DLP.

3.1.6 Representing Objects in a Logic Programming Language with Scoping Constructs (OLPSC)

A logic for scoping clauses and constants is introduced in OLPSC [HM90]. Recall that a Horn clause has the general form

$$A \leftarrow B_1,\ldots,B_n \qquad (n \geq 0)$$

In the newly introduced logic, B_i can be replaced with more complex formulas of the form

$$H_1 \wedge \ldots \wedge H_m \Rightarrow B_i \qquad (m \geq 0)$$

where $H_1,\ldots,H_m$ are Horn clauses and $\Rightarrow$ is the converse of $\leftarrow$. To prove B_i, the clauses $H_1,\ldots,H_m$ are first loaded into the current program (i.e. the database in Prolog) and only then is B_i attempted. After B_i succeeds or fails, these clauses are removed from the current program. For example, consider the clause

$$p(X,G) \leftarrow \text{all } y\backslash(assoc(y,x) \Rightarrow G)$$

where `all y\` means the universal quantifier y, `assoc` is a binary predicate symbol. To prove the goal `p(a,G)`, a new constant, say `c`, and a new clause, `assoc(c,a)` are added to the current program and then the goal `G` is called. In other words, `G` is carried from one context to another augmented context. This programming style is known as *goal-continuation passing*. A syntactic construct for modules is added to denote (possibly parametric) collections of program clauses. The module declaration

$$\begin{array}{l} \texttt{MODULE mod}(x_1,\ldots,x_n). \\ \texttt{LOCAL } y_1,\ldots,y_m. \\ H_1(x_1,\ldots,x_n,\ y_1,\ldots,y_m). \\ \qquad\qquad \vdots \\ \qquad\qquad \vdots \\ H_p(x_1,\ldots,x_n,\ y_1,\ldots,y_m). \end{array}$$

associates to the name `mod` the parameters $x_1,\ldots,x_n$, the local constants $y_1,\ldots,y_m$, and the clauses $H_1,\ldots,H_p$. The syntax $\texttt{mod}(t_1,\ldots,t_n) \Rightarrow B$ is equivalent to

$$\text{all } y_1,\ldots,y_m \setminus [(H_1(t_1,\ldots,t_n,\ y_1,\ldots,y_m),\ldots,H_p(t_1,\ldots,t_n,\ y_1,\ldots,y_m)) \Rightarrow B].$$

Objects are represented by modules. For example, a module for an object denoting a locomotive could be given by

```
MODULE locomotive.
LOCAL train.
make_train(train(S, Cl, Co), S, Cl,
Co).
color(train(S, Cl, Co), Cl).
speed(train(S, Cl, Co), S).
country(train(S, Cl, Co), Co).
```

State of an object can be modeled by the query,

$$\texttt{?- object(state)} \Rightarrow \texttt{goal}$$

During the testing of `goal`, the `state` of the `object` is hence accessible. To modify the state, use the query,

$$\begin{array}{l} \texttt{?- object}(state_1) \Rightarrow \\ \quad (\texttt{object}(state_2) \Rightarrow \texttt{goal}) \end{array}$$

The content of the current program looks like:

$$\begin{array}{l} \vdots \\ \texttt{object}(state_2). \\ \texttt{object}(state_1). \\ \vdots \end{array}$$

Thus, if the depth-first searching rule is followed, then $\texttt{object}(state_2)$ will always be selected first. Multiple inheritance is allowed in this approach. However, how a class definition might redefine an inherited method is still in investigation and the possibility of concurrent execution is not mentioned in the paper.

3.1.7 KSL/Logic

KSL [IC90] is a class-based object-oriented system in which everything including class, behavior(method and slot) and expression is an object. Domain object is a kind of object that controls the object's behavior according to the problem domain. Operations are invoked by message passing. When a message is sent to a domain object, the message selector and the class of the object identify the behavior object that

defines the operation required. An evaluation message is then sent to the behavior object causing its functionality to be applied to the domain object.

There are several classes of behavior objects. Method object is a class of behavior objects that implements procedural programming in KSL. When a method object receives an evaluation message, further evaluation messages are sent to each KSL expression object in the method's expression list. The evaluation behaviors of these expression objects implement the functionality of KSL.

Logic programming is incorporated into KSL (the hybrid language is called KSL/Logic) by introducing two behavior classes called PredicateBehavior and BuiltInPredicateBehavior. A PredicateBehavior object is associated with a class of domain object and contains all of the Horn clauses for that predicate as defined in the associated domain class. The specialized evaluation behavior for the PredicateBehavior class will invoke the predicate resolution. So a domain object determines the predicate that will be applied when reasoning occurs in its domain. Reasoning in KSL/Logic is thus domain-specific rather than reasoning under the single domain (the centralized database) in conventional logic programs. Also, dynamic domain switching during inference is allowed. BuiltInPredicates like *member*, *is*, *var* and *nonvar* are defined in BuiltInPredicateBehavior objects.

In KSL/Logic, arguments of predicate expressions can be procedural expressions. In addition, argument values can be complex object structures to be matched.

3.1.8 Orient84/K

Orient84/K [IT87] is an object-oriented knowledge-base system. Knowledge is modularized as objects called knowledge objects. A knowledge object consists of three parts, namely the knowledge-base part, the behavior part and the monitor part. It has the metaclass-class-instance hierarchy with multiple inheritance.

The behavior part is the collection of methods for the object. It receives requests for inference and action from other objects and sends such requests to other objects. A few methods for inferring from the knowledge in its knowledge-base part are predefined. Also, there are some predefined methods to assert/retract rules and facts into/from its knowledge-base part. It may request the monitor part to control incoming messages. The syntax and semantics of this part are similar to those of Smalltalk-80 [Gol84].

The knowledge-base part contains the local knowledge which is stored in the form of rules and facts. The syntax and semantics of this part are closely related to those of Prolog.

The monitor part acts as the demon, guardian, and supervisor of the knowledge object. As the demon, it checks the consistency of the knowledge-base, provides default reasoning and handles exceptions. As the guardian, it checks the authorization of an incoming message and decides whether the sender of the message has the right to use the corresponding method or not. As a supervisor, it decides which incoming message is processed first according to the priority and mode of its methods. Mutual exclusion among methods in an object is accomplished by the mode of the method. In addition, Orient84/K supports concurrent programming by allowing every object to run concurrently.

3.1.9 Vulcan

Vulcan [KTMB87] is a higher-lever language for object-oriented programming based on Concurrent Prolog [Sha87]. To incorporate objects in logic programs, Concurrent Prolog represents states by means of logical variables. For example, consider the following Concurrent Prolog program:

```
account([deposit(Amount)|
NewAccount],Balance,Name) :-
Plus(Balance,Amount,NewBalance),
account(NewAccount?,NewBalance?,
      Name).
```

In the notion of object-oriented programming, the above clause is a method of the object `account` for handling the incoming message: `deposit(Amount)`. `NewAccount` is a new uninstantiated logical variable for further messages. State changes are accomplished by replacing old logical variables with new logical variables. Thus, a logical variable can be treated as a pointer to the object.

In Vulcan, programs are translated into Concurrent Prolog for execution. Classes and methods can be defined explicitly in Vulcan. For example,

```
Class(window,[X,Y,Width,Height,
      Contents])
```

declares a class called `window` with attributes `(X, Y, Width, Height, Contents)`. A method for getting the position of a `window` can be defined as :

```
      window :: position(X,Y)
```

which is then translated into Concurrent Prolog as:

```
window([position(X,Y) | NewWindow],
X, Y, Width, Height, Contents) :-
   window(NewWindow?, X, Y,
      Width, Height, Contents).
```

Verbosity in writing object-oriented program using Concurrent Prolog can thus be alleviated in Vulcan. The object-oriented features supported include message passing, class inheritance and inheritance by delegation. However, backtracking over method calls is not possible due to the committed choice character of Concurrent Prolog.

3.1.10 The Bridge approach

An interface (a "bridge") of Loops (an object-oriented programming environment) [Xer88] and Quintus Prolog [Qui91] is developed in [KE88]. The interface is divided into three layers. At the lowest layer, a Prolog predicate can access Loops objects through three Prolog procedures: `get_value/3`, `put_value/3`, and `send_message/4`. For example, one should use the following procedure to perform a slot-value substitution in a Loops object:

```
substitute_slot_value(Object,Slot,
Old_value,New_value):-
get_value(Object,Slot,Old_value) ,
put_value(Object,Slot,New_value).
```

At the intermediate layer, methods of Loops objects can be defined as Prolog procedures. This is done by specializing the class Method in Loops to define a new class called PrologMethod. PrologMethod has four components: a method object, a functional definition (i.e. the object's interface part), the text representing the Prolog source code, and a set of clauses that have been interpreted or compiled into the Prolog database (i.e. as the object's implementation part). When a PrologMethod object is sent a message, it determines which method definition should be used. If the appropriate method definition is written in Prolog, a function called `PROLOG` will be invoked instead of applying a Lisp [WH89] function to the arguments list (originally, Loops methods are Lisp functions). The `PROLOG` function is an interface between Lisp functions and Prolog procedures. For example, if one wants to invoke the Prolog procedure `foo/3`, using the function call

```
(PROLOG 'foo(LIST 'arg1 *VALUE*
'arg3))
```

would have the same effect as the Prolog query

```
      foo(arg1, X, arg3).
```

The `PROLOG` function will always return only the first answer for `X`, i.e. it is deterministic.

At the highest layer, Prolog clauses can be treated as Loops objects, called PrologClause objects. PrologClause objects have slots to store things like the functor name, number of arguments and the list of arguments. If different sets of clauses represent different views of a problem, then we can alter views simply by asserting or retracting sets of PrologClause objects.

3.1.11 Discussion

As we have seen, there are many approaches to combining the logic and the object-oriented paradigms. We may classify them according to the dominant paradigm of each approach. LogiC++, DLP, KSL/Logic and Orient84/K incorporate the logic paradigm into an object-oriented framework by replacing procedural methods by Prolog clauses. On the other hand, Intermission, Vulcan, CPU, OOPP, and OLPSC incorporate some object-oriented components like objects, classes and inheritance into a logic framework. Usually, the logic based approaches use preprocessors to translate the object-oriented programs into ordinary logic programs. So, less implementation effort is required in these approaches. However, such kind of implementations is unavoidably inefficient since they are built on top of other languages. Also, since most of the logic programming languages only support unification of simple syntactic representations, user-defined complex objects are not unifiable. The Bridge is an interface approach between the two paradigms and

so no paradigm dominates the others. Fig. 2 shows the positions of the approaches in the FOOL-space.

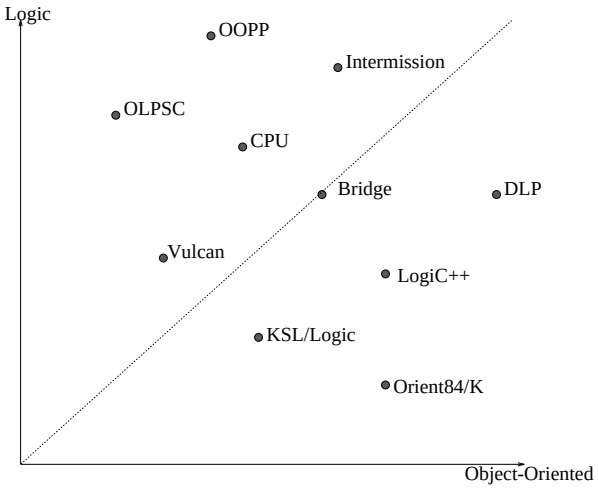


Fig. 2 Some logic+object-oriented approaches in the FOOL-space

How to represent state is an important issue in the integration of the object-oriented and the logic paradigms. Basically, there are three ways to represent state:

- The state is associated with a few arguments of some predicates. The current state is reflected by the values of such arguments. Tail-recursive calling of the same predicate with different argument values denotes changing the current state to a new one. Examples are Vulcan and Intermission.
- The state is recorded as clauses (usually facts) in the database. To alter the current state, one should modify the database by the predicates `assert` or `retract`. Examples are Orient84/K and OLPSC.
- Adding some imperative constructs like instance variables and the assignment statement to manipulate the state. DLP is an example.

The first way is simple and clear but of limited use in practice. It is tedious to carry the state around a program by passing it as arguments to every method accessible to the state. Moreover, persistent storing of the state is impossible. On the contrary, the second and third ways are more convenient in accessing the state. However, this may be harmful if the state is accessible by "everyone". So a better policy is to protect the state from accessing by methods without direct access permission.

It is possible for a language to employ more than one of the above three ways to represent states. For instance, both the first and the third ways are used in a Parlog based object-oriented language called Polka [Dav88]. A Polka class may define state variables and alter their values using an assignment operator *becomes*. The Polka class is then compiled into a set of Parlog clauses. State variables are compiled as part of the arguments of some predicates. The *becomes* operation is implemented by argument replacement between tail-recursive calls of the same predicate. Thus, Polka supports the third way of state representation to programmers but implements it by the first way.

3.2 Functional + Object-Oriented

3.2.1 A PaRallel Object-Oriented Functional computation model (PROOF)

A PROOF [YJB91] program comprises a set of objects which can be executed in parallel. Each object has its own local data and methods and is an instance of a class. The methods in PROOF are pure applicative functions. That is, the functional paradigm is incorporated at the method definition level.

In PROOF, a number of objects can be active simultaneously. To synchronize among such objects, an optional precondition called *guard* is attached to each of the methods in a class. When an object invokes a method, if the attached guard evaluates to true, the method will be executed; otherwise the object will be suspended until the guard becomes true. Since the guards depend only on the local data of objects and do not depend on the definitions of methods, they can be inherited with the methods they are attached to. The following example is a definition of a class `Bounded_Buffer` with guards in PROOF:

```
class definition
   Bounded_Buffer(itemtype, size)
   compositionstore:list(itemtype)×
               count:int
   method put b x
      guard (b.count < size)
      expression
      β[(append_right x), inc] b
```

```
   method get b
      guard (b.count > 0)
      expression
      [β[tail, dec] n, head(b.store)]
   method is_empty b
      expression
            b.count = 0
   method length
      expression
            b.count
end class
```

In the above program, β is the distributed apply function. $\beta[f_1, f_2, \ldots, f_n][x_1, x_2, \ldots, x_n] \equiv [f_1(x_1), f_2(x_2), \ldots, f_n(x_n)]$ where f_i is a function. The symbol = is the logical operation "equal". The guard attached to the method `get`, `(b.count>0)` states that `get` can be executed only when the buffer is not empty.

PROOF is history sensitive by making the objects persistent and allowing the assignment of values to objects through a pseudo function ℜ, called the reception function. ℜ[[o]](e) denotes the assignment of the result of evaluating the expression e to the object O. e can be evaluated in parallel and the local data of o can be modified only as a whole entity. Simultaneous modification to the same object must be serialized. The serialization is done by a two-phase multimode locking protocol.

A prototype programming language PROOF/L is under development. A program in PROOF/L is translated into an equivalent program with explicit parallel constructs.

3.2.2 A Functional Language with Classes (FLC)

FLC [BSW91] is a typed functional language that supports objects, classes, multiple inheritance and parametric polymorphism. In this language, an object is modeled as a *typed record*. The methods of the object are structured as fields with functional types. A class is a mapping from a set of instance variables to an object. Inheritance is modeled by subtyping. The following class declaration defines a class of objects with an instance variable `x`, methods `#a` and `#b`:

```
   val C = class x methods #a = x and
       #b = (fn z ⇒ x+z) end;
```

This defines a class value `C` which has the following type:

```
C : int → object {#a :Pres(int), #b
= Pres (int→int)} Empty
```

where `C` is a function. It maps from an integer value (the instance variable `x`) to an object value. Method `#a` is a constant function which always returns the value of `x`. Method `#b` is a function of single argument `z`. The keyword `Empty` represents the infinite sequence:

$$l_1\text{:Absent}, l_2\text{:Absent}, \ldots$$

which means that the object has only two methods `#a` and `#b`. All other methods are absent.

Instantiation of classes is done through functional application. For example,

```
val O = C 3
```

creates an object `O` of class `C` with 3 as the value of its instance variable `x`.

Inheritance is implemented by expanding the fields of a record. For example, a new class `C'` that inherits method `#a` from `C` may be defined as:

```
val C' = class x
   inherits # a from C with x
   methods #c = self.#a + 1
 end;
```

A computationally equivalent definition of `C'` is

```
val C' = class x
   methods #a = x
   and     #c = self.#a + 1
end;
```

Since methods are modeled as typed record fields, we can determine which fields must be present in a record and so determine the set of messages supported by an object. As a result, the "method not found" run-time error can be eliminated. The treating of classes as functions make FLC essentially a strongly-typed functional language. For this reason, it is appropriate to implement FLC using another strongly-typed functional language. FLC has been implemented in ML.

3.2.3 Common Lisp Object System (CLOS)

CLOS [GWB91, WH89] is a system designed for writing object-oriented programs using Lisp. CLOS primitives make it possible to define *generic functions*. A generic function is a collection of methods that share the same method name. In Lisp, an operation is defined by a single piece of code. Any argument-type conditionality is expressed as code explicitly programmed by the user (e.g. using the `(cond...)` construct). In contrast, generic function in CLOS supports automatic selection of the most appropriate method according to the argument types of the methods involved. For example, consider the following CLOS program:

```
(defclass figure () () )
(defclass triangle(figure)
((base :accessor triangle-base :initarg :base)
(altitude :accessor triangle-altitude :initarg
:altitude)))
(defclass rectangle (figure)
((width :accessor rectangle-width :initarg
:width)
(height :accessor rectangle-height :initarg
:height)))
(defclass circle (figure)
((radius :accessor circle-radius :initarg
:radius)))
(defmethod area ((figure rectangle))
       (*  (rectangle-width figure)
              (rectangle-height figure)))
(defmethod area ((figure circle))
       (*  pi
           (expt   (circle-radius figure) 2)))
```

The class `figure` has three subclasses: `triangle`, `rectangle` and `circle`. The subclass `triangle` has two slots: `base` and `altitude`; `rectangle` has two slots `width` and `height`; `circle` has a slot `radius`. Each such slot is exclusively accessible by the procedure named after the keyword `:accessor`. The method `area` is a generic function composed of three different implementations. When the method `area` is called with an object which is an instance of one of the three subclasses of figure as the argument, the most appropriate definition will be used. For instance,

```
*(area (make-instance 'rectangle
     :width 10 :height 5))
```

will print 50.

3.2.4 FOOPS

FOOPS [GM87] unifies functional and object-oriented programming by providing abstract data types (ADTs) for object attribute values. For example, consider the following definition of class `ACCT` in FOOPS:

```
omod ACCT is
      :
 class Acct
 attrs bal : Acct→Money
      :
 methods credit, debit : Acct →Acct
 ok-axioms
      :
 bal(credit(A,M)) = bal(A) + M
 bal(debit(A,M) = bal(A) - M if
 bal(A) >= M
      :
endo ACCT
```

An object of class `ACCT` has an attribute `bal`. The value of `bal` is the amount of money currently present in the account. It is the state of the object. All state changes are axiomatized by equations that describe the effects of methods `credit` and `debit` on `bal`. The attribute `bal` is of an ADT `Money` which is defined by a functional model `MONEY`:

```
fmod MONEY is
   protecting DECRAT
      sorts
   Money < DecRat
      *** Money is dollars & cents;
      *** may be negative
   PMoney < Money
      *** PMoney is positive Money
   Pate < DecRat
      *** for interest rate
   fn roundoff : DecRat → Money
   ok-axioms
   (D : DecRat) as Money if
      int-part(100*D) == 100*D.
   (M : Money) as PMoney if M > 0.
   D : DecRat) as Rate if D > 0.
   roundoff(D) = (int-part(100*D))
      / 100.
endf MONEY
```

This module imports the built-in module `DECRAT`, which provides a type `DecRat` for finite decimal

rational notation (i.e. an integer, a decimal point, and a natural); `int-part(R)` gives the part of `R` before the decimal point. `PMoney` is a subtype of `Money` for positive interest rates. `Rate` is a subtype of `Money` for positive interest rates. `Money`, `PMoney` and `Rate` are defined through a type constraint mechanism which is an equational condition that defines when data elements belong to a given subtype.

There are two kinds of modules in FOOPS: function-level modules (e.g. `MONEY`) and object-level modules (e.g. `ACCT`). While function-level modules define ADTs, object-level modules define classes. Multiple inheritance for both types and classes, object-valued attributes, generic modules (both function-level and object-level) are supported in FOOPS.

3.2.5 Discussion

Some concepts like data-abstraction, overloading and polymorphism are shared by the two paradigms. The basic difference between them is that the notation of global variables is present in the (imperative) object-oriented paradigm but absent in the functional paradigm. This difference makes many people feel that the integration of these paradigms is impossible and this is the reason why few researchers work on this kind of integration. The four approaches we have seen fall into two categories (see Fig. 3). In FOOP and FLC, the main effort is to fit some of the concepts of object-oriented programming into the functional paradigm without compromising its purity. On the other hand, in PROOF and CLOS, some object-oriented features, in particular local state and assignment, are present at the object level. The purity of the functional paradigm is preserved at the method level (this may not be true in CLOS since Lisp is not a pure functional language). As a result, referential transparency is not preserved at the object level and complex mechanisms are required to serialize the simultaneous access of the instance variables (In PROOF, the use of the two-phase locking protocol in the serialization of the updating of objects may cause deadlock). Thus, PROOF and CLOS are not pure in the view of functional programmers.

[Weg90] points out that types and classes are similar but actually different concepts. Type is a mechanism for specifying the structure of expressions. Class is a mechanism for classifying values by their properties and behavior. The primary purpose of typing is to provide constraints on expressions for type checking by compilers. The primary purpose of classifying is to specify the behavior of classes for program development, enhancement and execution. Types are described by predicates over expressions used for type checking, whereas classes are specified by templates used for creating and managing objects with the same behavior.

Due to these differences in purpose and specification, *subtypes* and *subclasses* are derived from their parents in different ways. Subclasses are defined by template modification mechanisms that are able to modify their parents' behavior arbitrarily. Subtypes, on the other hand, are defined by additional predicates for constraining the structure of expressions. Thus subtyping constrains behavior modification while subclassing facilitates flexible system evolution. However, among the four approaches we introduced, only FOOP bolsters both subtypes and subclasses; the other three approaches support either subtypes or subclasses but not both (See Table 2).

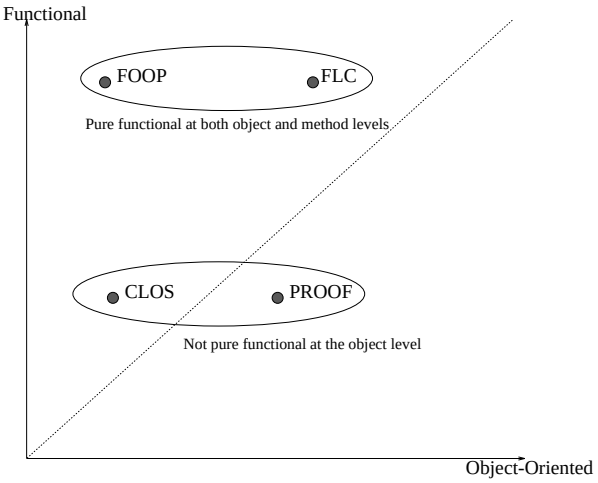


Fig. 3 Positions of some functional + object-oriented approaches in the FOOL-space

3.3 Logic + Functional

3.3.1 HOPE

The approach of HOPE [DFP85] is based on the concept of *absolute set abstraction* where the set members are defined implicitly by a set of conditions which are equations involving functional expressions. For example, given the `append` function:

```
append(nil,l) ⇐ l;
append(x::l1,l2) ⇐ x::append(l1,l2);
```

Then, the `split` function can be defined by absolute set abstraction form as:

```
Split(l) ⇐ {l1,l2| append(l1,l2)=l};
```

where l_1 and l_2 are logical variables. `Split` is a set-valued function since all list pairs satisfying the condition `append(`l_1`,`l_2`) = l` will be returned by `Split`. For example, `Split([1,2])` returns `{([],[1,2]), ([1],[2]), ([1,2],[])}`. Thus, non-determinism is introduced by set-valued functions.

Function evaluation is done by unifying the equations forming the conditions against the equations of the program. In the evaluation of `split([1,2])`, the condition `append(`l_1`,`l_2`)=[1,2]` is unified against the two `append` equations. In this case, the second `append` will be matched and the unification continues recursively until no unresolved conditions remain.

Since a set may be defined using more than one condition and a function may be defined by more than one equation in a program, alternative evaluation strategies exist. Determining of what condition to elaborate next and what equation to be matched corresponds to the AND and OR parallelism in logic programs respectively.

HOPE is a higher-order functional language. Logical variables are able to bind objects of function type, and through unification to produce function values for them. For example, given the definition of `map` :

```
map(nil,f) ⇐ nil;
map(x::l,f) ⇐ f(x)::map(l,f);
```

and the definition of `to` :

```
to(l1,l2) ⇐ {f | map(l1,f)=l2}
```

where `f` is a function in both definitions. Then `to([1,2],[3,4])` should produce the definition of `f : {f | f(1)=3 and f(2)=4}`.

Although unification and non-determinism are provided at the programmers specification level, they are not directly supported at run time. Indeed, a HOPE program is transformed into a pure functional program in compilation time.

A function in HOPE can be used in several modes while a normal function has only one mode of use. A multimode function is accomplished by analyzing the modes of use of the function used in implicit definitions and converting the original function definition into one whose normal mode of use is the one required to produce functional values for the output variables within the set expression. Logical variables are eliminated and only pattern matching and deterministic computation are used at run time.

3.3.2 FUNLOG

FUNLOG [SY84,SY85] employs *semantic unification* as the basic problem solving mechanism. Functions in FUNLOG are defined by sets of equations. For example, the function `if_then_else` can be defined as :

```
if_then_else(true,X,Y) = X
if_then_else(false,X,Y) = Y
```

A FUNLOG program consists of a set of function definitions and a collection of Horn clauses with equality. A Horn clause with equality is of the form:

$$G :- P_1, P_2, \ldots, P_n$$

where P_i can be of the equation form `M=N`. For example, the clause :

```
r(X,Y) :- f(X)=A, A^h(5)=f(3^X),
p(A,X,Y,Z)
```

is a valid relation in FUNLOG.

In solving the goal `G`, all P_i will be solved as in pure logic programs. However, if P_i is of the form `M=N`, semantic unification instead of conventional unification will be invoked to unify `M` and `N`. Semantic unification is indeed conventional unification with pattern-driven reduction. When two forms say `f(`t_1`,...,`t_n`)` and `f(`s_1`,...,`s_n`)` are being unified and suppose t_i and s_i are not unifiable in conventional sense. However, if s_i(or t_i) is reducible (by terms rewriting which is based on function definitions) and a reduced version of it is unifiable with t_i (or s_i), then `f(`t_1`,...,`t_n`)` and `f(`s_1`,...,`s_n`)` are semantic unifiable.

The pattern-driven reduction is in lazy manner. That is, reduction is performed only when it is

necessary to make two terms semantically unifiable. For example, the reduction of the function term `if_then_else(f(a),g(b),h(c))` will reduce `f(a)` in order to match `true` or `false` but will not reduce `g(b)` and `h(c)` since the values of them are unnecessary in this stage.

With lazy reduction, the model is capable of computing infinite data structures. Also, since a single function can be defined via multiple equations, non-determinism is allowed in function evaluation.

3.3.3 F*

Lazy evaluation is the key concept in F*[Nar90]. Besides the capability of handling infinite data structures, lazy evaluation has been shown to be optimal for the reduction process. That is, values of terms can be determined in a minimum number of reduction steps.

The concept of lazy evaluation is brought to logic programs in the following way: Originally, an expressive, first-order and nondeterministic rewriting system F*(A F* program is a set of reduction rules) and an abstract lazy interpreter for it are defined. F* is then transformed into Horn clauses which will be interpreted by a resolution interpreter. The behavior of the lazy F* interpreter is preserved during the transformation so that it can be simulated by resolution. For example, the F* program :

```
infinite_ones ⇒ [1|infinite_ones].
head([U|V]) ⇒ U.
```

is transformed to the Horn clauses:

```
(a)   reduce(1,1).
(b)   reduce([U|V],[U|V]).
(c)   reduce(infinite_one,Z):-
      reduce([1|infinite_ones],Z.
(d)   reduce(head(X),Z) :-
      reduce(X,[U|V]), reduce(U,Z).
```

Evaluation of the function call `head(infinite_ones)` by the lazy-interpreter results in the value 1. The corresponding execution by the resolution interpreter on the goal `reduce(head(infinite_ones),Z)` is:

```
      reduce(head(infinite_ones,Z))
             |
By (d) reduce(infinite_ones, [U|V]),
      reduce(U, Z)
             |
By (c) reduce([1|infinite_ones],[U|V]),
      reduce(U,Z)
             |
By (b) reduce(1,Z)
             |
By (a) reduce(1,1)
```

`Z` is bound to 1 and the resolution steps exactly simulate the lazy-rewriting process.

3.3.4 LEAF

LEAF [BBLM84, BBLM85] has a declarative and a procedural component. Procedures in the procedural component are defined by sets of rewrite rules, which can be either directed equations or annotated Horn clauses. Each term is annotated either as input or output mode. The rewriting system is deterministic and obeying the lazy-evaluation strategy.

The declarative component is composed of Horn clauses and first-order equational theories with constructors. A clause in the declarative component is :

$$H \leftarrow B_1, \ldots, B_n$$

where H is a header and $B_1, \ldots, B_n$ is a collection of atoms. Both header and atom have function form $f(t_1, \ldots, t_n)=t$ and relation form $p(t_1, \ldots, t_n)$ where f is a function symbol, p is a predicate symbol, and $t_1, \ldots, t_n$ are terms. The declarative component uses unification and is nondeterministic. Communication in LEAF is through channels which correspond to variable symbols and are directed from producer(s) to consumer(s). Procedure and declarative components can be integrated in two ways. One is to invoke a procedure within a declarative rule. For example, consider the following clauses :

```
Family_income(x,y)←Not_married(x),
Income(x,y)
Family_income(x,y)←Wife_of(x,z),
Income(x,u),Income(z,v),+(In:u,v;
Out:u)
```

The predicates `Not_married/1`, `Income/2` and `Wife_of/2` are defined declaratively while `+` is a procedure.

The only restriction in invoking a procedure in a clause is that all input variables should be bound before the procedure is invoked. Such restriction can be achieved by ensuring that there is no occurrence of input variables in the clause head.

When procedures are legally invoked in clause bodies, the procedural component evaluates the procedures in the standard way and puts the output in the output channels which are consumed by the declarative component. So, even though a procedure is deterministic, the overall result may be nondeterministic. For example, the goal ← `Family_income(x,10000)` returns multiple values of `x`.

Another way of integration is to invoke declarative rules within a procedure. Such kind of combination poses a problem because a procedure whose body invokes a declarative process will in general be nondeterministic and so is illegal. The integration requires an interface between declarative and procedural processes such that a set of results is returned explicitly by the declarative process. The proposed solution is to treat declarative programs as data for the procedural component. The procedural component works as a meta-language which acts on meta-objects such as declarative rules and (possible infinite) sets of solutions of declarative programs.

3.3.5 Applog

Applog [Coh85] is a combination of Lisp and Prolog. Basically, it is an applicative language. For example, the factorial function `fact` is defined in Applog as:

```
def(fact, lambda([X],
if (X=1, 1, X * fact(X-1)))).
```

Applog is embedded within Prolog. Facilities of Prolog are callable from Applog using the goal function:

```
goal(Goal, Form)
```

where the `Goal` is a Prolog goal and the `Form` is an Applog form. If the `Goal` is satisfied by Prolog, then the `Form` will be executed by Applog. If the Form fails, it backtracks into the `Goal`. For instance, suppose the database contains `a(1)`, `a(2)` and goal `(a(X), if (X>1, X+2))` is tested. Evaluation of `a(X)` will instantiate `X` to 1 and so the `if` fails. Then it backtracks to `a(X)`, finds `X` to 2, passes the test `X>1`, and returns 4 as the result. There is another way to use goal:

```
goal(Goal)
```

which calls Prolog to satisfy the `Goal` and returns the instantiated `Goal` as the result. Thus `goal(a(X))` returns `a(1)`. To call Applog from Prolog, use

```
eval(Form, Result)
```

where the `Form` is an Applog form. Variables in Applog are compatible with Prolog variables.

3.3.6 Discussion

Since the logic and the functional languages are the two most promising declarative languages, much effort has been devoted to their integration. The two essential distinctions between logic and functional programs are:

- Determinism: Logic programs are nondeterministic (search-based computation) while functional programs are deterministic (evaluation-based computation).
- Logical variables: Logic programs use unification, a bi-directional parameter passing mechanism. Functional programs use pattern-matching, a uni-directional parameter passing mechanism. Directionality in functional programs makes them less expressive but more efficient than logic programs.

Much work for combining these two paradigms is based on adding unification and logical variables to a functional language. FUNLOG is an example.

There are several directions in the integration of these two paradigms. The first one is to provide an interface between a functional language (usually Lisp) and a logic language (usually Prolog). Examples are Applog and LEAF. Two issues need to be considered in designing the interface between the two languages:

- How to return the multiple solutions found by a logic procedure to a function?

  In LogLisp [RS82], a Lisp-expression (ALL ($X_1,...,X_t$) $C_1,...,C_n$) may be used to query a logic program. $C_1,...,C_n$ is the Lisp representation of the query, and $X_1,...,X_t$ are the variables to be instantiated in the query. The function ALL returns the list of all solution lists of $X_1,...,X_t$.

- What are the common data structures of the two language components?

  It would be better if the two languages share the same data structures. LEAF achieves this by removing logical variables and nondeterminism from the logic language. However, in some systems like LogLisp, only the Lisp data structures can be manipulated in both languages.

  Some advantages of the interfacing approach are:

- Both paradigms coexist syntactically and semantically in a programming environment such that programmers may choose the most suitable paradigm to solve problems.
- High comparability with programs written either in logic or functional languages. Existing software is reusable.

However, the semantics of the combination is rather complex in these approaches.

Another direction of integration is to add equality to logic programs. Through the equality, function symbols are not only representing data constructors but also computable functions. FUNLOG and the declarative component of LEAF are logic languages with equality. The operational semantics of such kind of languages is essentially based on *narrowing*. Narrowing a functional expression is to find the minimum substitution for the variables in the expression such that the resultant expression is reducible, and then reducing it. The substitution is obtained by unifying the expression with the left-hand sides of equations. Narrowing is implemented by replacing pattern-matching in reduction by unification. For example, suppose we have the following equalities for reverse_list and append:

```
(1) reverse_list([]) = []
(2) reverse_list([H|T]) =
    append(reverse_list(T), [H])
(3) append([], Y) = Y
(4) append([A|X], Y]) =
    [A| append(X, Y)]
```

The expression reverse_list(X) is narrowed as follows:

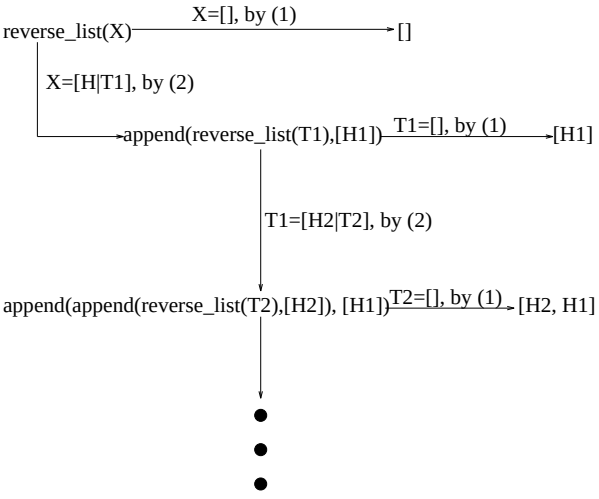


Fig. 4 An example of narrowing

There are other inference methods besides narrowing for logic programs with equality. Examples are the semantic unification in FUNLOG, and the SLD-resolution on the canonical form in LEAF. Nevertheless, it is pointed out in [BL86] that all these methods are essentially equivalent to narrowing.

The third direction of integration is to augment a functional language with set abstraction, as HOPE does. Nondeterminism is replaced by set union. For instance, a definition of append using set abstraction is:

```
append(L1, L2, L) =
{L1, L2 | (L1 = [] ∧ L2 = X ∧ L = X)
∨(L1 = [X|Y] ∧ L2 = Z ∧  L3 = [X|W] ∧
append(Y, Z, W))}
```

L_1, L_2 and L are logical variables. The predicate append is treated as a deterministic set-valued function which returns the set of all (L_1, L_2, L) such that append(L_1, L_2, L) is true. The objective of these approaches is to use reduction to execute logic programs. In contrast, in some systems like F^*, resolution is used to execute functional programs. The classification of the

approaches according to the dominant paradigms is shown in Fig. 5.

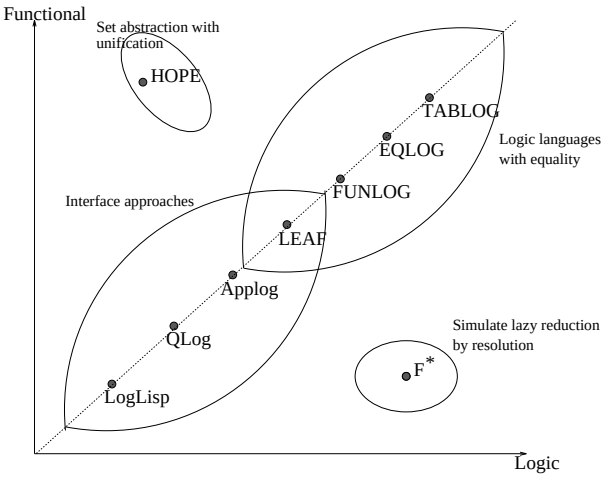


Fig. 5 The classification of some logic + functional approaches

3.4 Logic + Functional + Object-Oriented

3.4.1 Paradise

The elements of an ideal language called Paradise are proposed in [AK91a]. Paradise is a programming system combining the constraint logic paradigm, the object-oriented paradigm and the typed functional paradigm. It intends to provide three kinds of abstraction to programmers:

- *Abstract data structures*: Data is uniformly represented as attributed objects. Types are organized as an inheritance hierarchy and all operations are bound to types. There are a few built-in polymorphic types such as tuples, lists and arrays. User-defined data structures can be implemented by means of generic modules. Static type-checking is used.
- *Abstract constraint solving*: The language will support unification with respect to its data structures. Built-in constraint solving methods are provided for some useful applications such as numerical problems and propositional logic.
- *Abstract control structures*: It offers abstraction of control by introducing two control primitives: generalized exception handling and event suspension. Generalized exception handling is required for the searching over the values of variables in arbitrary constraint-solving. Event suspension allows adaptive control and automatic propagation of constraints.

Other features of Paradise are:

- Programs are structured as a graph of parametric modules. The consistency of modules is checked automatically.
- The programming system will be interactive and incremental.
- Operators are first-class objects, and may be functions, relations or procedures.

3.4.2 LIFE

LIFE (stands for Logic, Inheritance, Functions and Equations) [AK91b, AKMR92] is a language that combines logic programming and functional programming with a facility for structured type inheritance. The most essential element in LIFE is an uniform data structure, called ψ-terms, which represent all data structures, including lists, clauses, functions and sorts (types). ψ-terms may be constructed as records.

A ψ-term has a *sort* which can be regarded as a set of elements having some common properties. The interpretation of the term *sort* in LIFE is essentially equivalent to that of the term *type* in other programming languages like Pascal and ML. The subsort relation can be denoted by the set inclusion. The partially ordered set of all sorts can then be viewed as a multiple inheritance. Also, there is no distinction between values and sorts since the former can be treated as singleton sorts (i.e. a sort has only one element). For example, the value `3.5` is equivalent to `{3.5}` which is a subsort (subset) of the built-in sort `real`. Some examples of ψ-term are:

```
10    /* a particular integer */
string /*built-in types for a
sequence of characters*/
person (name⇒N:string, age⇒int,
friend⇒X:person)
```

The last example states that a `person` has a `name` which is of sort `string`, an `age` which is of sort `int` (integer), and a `friend` which is of sort `person`. `N` and `X` are reference tags which allow the fields name and friend to be referred at somewhere in their scope. Two ψ-terms are

unifiable if the largest common subsort of them is not the empty set. For example, unifying the following two ψ-terms:

```
U : car(color⇒red, wheels⇒4)
V : vehicle(wheels⇒N : int)
```

results in N=4, U=car(color⇒red, wheels⇒N), V=U.

LIFE is a generalization of Prolog by substituting ψ-terms to Prolog terms. A significant difference between Prolog terms and ψ-terms is that the latter does not have fixed arities. This causes the unification of two ψ-terms with the same sort symbol but different arities to be possible in LIFE (note : this is impossible in ordinary unification).

In addition to relations, functions are provided in LIFE. They may be higher-order and in curried fashion. Furthermore, they can be called with insufficiently instantiated arguments. In this case, the evaluation of the function will suspend until its variables are sufficiently instantiated. This suspension mechanism allows interleaving interpretation of relational and functional expressions.

An interpreter for LIFE called Wild LIFE has been implemented in C.

3.4.3 UNIFORM

UNIFORM [Kah85] is a language designed for unifying Lisp, Prolog and Act 1 [Lie87] based upon augmented unification which is an extension to syntactic unification in a way that two expressions are unifiable if the equivalence of the expressions can be deduced from the current set of assertions in the program. For example, given the following assertions of `factorial`:

```
(=(factorial 0)   1)
(=(factorial n) (* n (factorial (-n
1)))))
```

Then `factorial` can be used as a function:

```
(=(factorial 4) (integer n))
```

which results in `n=24`. It can also be used as an inverse function:

```
(=(factorial   n)    120)
```

which results in `n = 5`.

When an user inputs a goal, UNIFORM tries to unify it with assertions of the program in a breath-first manner. Also, users can specify whether or not the occur-check should be performed on each type of expression. If the occur-check is not performed, then the system may need to unify circular structures. For instance, the system should interpret the expression `(=X (con 'a X))` to mean that `X` is an infinite list of `a`.

UNIFORM programs are very similar to Lisp programs. For example, `append` can be defined in UNIFORM as :

```
(assert
   (= (append front back)
      (cond  ((null front) back)
      ((cons (first front)
      (append (rest front)
      back))))))
```

The major difference between UNIFORM and Lisp is that Lisp uses evaluation while UNIFORM uses augmented unification in problem solving.

Classes, methods and inheritance are supported in UNIFORM as Act1 does. For instance, the class of lists of successive integers, `list-of-integers`, may be defined as:

```
(= (list-of-integers begin (< begin))
      () ))
(= (list-of-integers begin (and end
   (>=  begin)))
   (cons begin
      (list-of-integers
      (+ begin 1)
   end))))
```

A method is applicable to an object if the type of the method is unifiable with the object. Subclasses can be defined under some parent classes. For instance, a subclass of `list-of-integers` which starts with 1 may be defined as:

```
   (= (first-n-integers n)
      (list-of-integers 1 n)))
```

3.4.4 G

G [Pla91] is a multiparadigm language which tries to integrate the functional, object-oriented, relational, imperative and logic paradigms within a single linguistic structure through a single data

type called stream. Every built-in type and user-defined type is a stream value. Some examples of stream values are shown:

```
2  /* an integer scalar */
[4, "foo", 'c']   /* a composite
stream of scalar */
[7.5, [11, 9]]    /* a composite
stream of streams*/
[]    /* the empty stream*/
```

Every stream has three principal characteristics: an environment, a value sequence and an enumeration protocol. For example, if an integer 7 is typed into the G interpreter, then it is a stream which has the global environment of the interpreter as its environment, the value 7 as its value sequence and a trivial enumeration protocol that simply enumerates the single value.

To illustrate how to express the logic paradigm in G, consider the following program:

```
grandfather := {
      father [?x, ?y],
      father [y, ?z] → [x, z]
                }
father := [["john", "tom"], ["alan",
"albert"], ["tom", "peter"]]


--->  grandfather
Ans: ["john", "peter"]
```

The relation `grandfather` formulates that `x` is the grandfather of z if there exists a `y` such that `[x,y]` and `[y,z]` are members of the stream `father`. The operator ? is a binding operator. The following example demonstrates how to write functions in G:

```
---> func range(x, y) x .. y
---> range ('a', 'e')
Ans: ['a', 'b', 'c', 'd', 'e']
```

`range` is a function with two parameters `x` and `y`. It enumerates all the ordinary values from `x` to `y` inclusively. Consider the following example of defining a class of objects `stack`:

```
--->  addtype(Stack,  Stream,  stack,
local[stack : 0])
```

This expression defines a new type named `Stack` which has a local instance variable `stack` with initial value 0. To create an instance of `Stack`, type

```
      ---> s := make (Stack)
      ---> s
      Ans: 0
```

To add a method called `push` to `Stack`, type

```
---> fun c {Stack} push (val) (stack
:= val; stack); val
---> s :: push (10)
Ans: [10]
---> s
Ans: [10, 0]
```

The G interpreter, which employs lazy evaluation, is implemented in C.

3.4.5 FOOPlog

FOOPlog [GM87] is an extension of FOOPS in the sense that FOOPlog axioms are Horn clauses with equality instead of just equations, and it allows evaluation of existential queries as well as method expressions. Also, logical variables and backtracking are present in FOOPlog. Semantically, unification of the three paradigms is achieved by adding reflection to Horn clause logic with equality.

3.4.6 Logic and Objects (L&O)

L&O [MaC92] incorporates the concepts of classes, expressions and functions into logic programs in order to simplify the task of programming large application. In general, a class template may have two components: a class body and a set of class rules. A class body is a collection of axioms in the form of facts and rules with a class label. For example, in the following L&O program

```
scotsman : {
   color(green).
   country(britain).
   speed(100).
   journey_time(Distance, T)
   :- speed(S), T is Distance/S.
   }
```

where `scotsman` is the class label. `color/1`, `country/1`, `speed/1` and `journey_time/2` are the axioms. Class rules

are used to relate different class templates to each other through inheritance. A class rule of the form: `mylabel⇐parentlabel` means that all axioms in the class identified by the label `parentlabel` is also in the `mylabel` class.

There are two types of query: relational query and expression query. A relational query is similar to the conventional query in Prolog except that it may be prefixed by a class label. For example, scotsman: `journey_time(100,X)?` is a valid relational query. An expression query consists of an expression suffixed by an '=?'. It is useful when we wish the system to evaluate an expression rather than prove a goal.

In addition to clauses, functions can be defined with a class body by means of conditional equalities. A conditional equality is a statement of the form:

$$f(X_1, \ldots, X_n) = G :- C_1, \ldots, C_m$$

This declares that the terms $f(X_1, \ldots, X_n)$ and G are equal if conditions $C_1, \ldots, C_m$ hold. For example, a class template which implements the insertion sort can be defined in L&O as follows:

```
insertionsort :   {
   sort([]) = [].
   sort([E] | List]) =
      insert(E1, sort(List)).
   insert(E1, []) = [E1].
   insert(E1, [E|List]) =
      [E1, E|List] :- E1 < E.
   insert(E1, [E|L1]) =
      [E|insert(E1, L1)]):- not E1<E.
}.
```

The semantics of L&O programming is established by constructing a transformation from L&O programs to ordinary logic programs. It is proved that L&O programs are still first order logic programs.

3.4.7 Discussion

Instead of comparing the features supported by each approach individually, let us examine them from the views of two fundamental components of any programming language: semantics and data objects. The semantics of the object-oriented part (mainly the inheritance) of LIFE and FOOPlog is based on the ψ-calculus [AK91b] which is a mathematical model for representing record-like data structures in logic and functional programming. Therefore, semantically speaking, both LIFE and FOOPlog can be regarded as an integration of three mathematical models: ψ-calculus (object-oriented programming), λ-calculus (functional programming) and first-order logic (logic programming). Thus, integration of paradigms in LIFE and FOOPlog is at a declarative level. On the other hand, the core of UNIFORM augmented unification is concerned with the language's operational semantics. In G, different paradigms cooperate in order to modify, compute and enumerate streams properly. Thus, integration of paradigms in UNIFORM and G is at an operational level. The semantics of L&O programs is essentially equivalent to that of logic programs. Paradise is just a proposal for future languages, so no semantics for it has been given yet. Nonetheless, since it comes after LIFE, we can foresee that the semantics of Paradise will also be described as a combination of the ψ-calculus, λ-calculus and first-order logic.

The various integrated approaches we have discussed in this section have three different levels of data objects. In UNIFORM, FOOPlog and L&O, the basic data objects are Herbrand terms. In G, linear data objects named streams are used. In Paradise and LIFE, data is uniformly represented by hierarchical structures: attributed records. These three different levels of data objects are analogous with the three data object levels in Pascal: basic data types such as integer and character, array data type and record data type. From our experience, we know that the hierarchical data type has the greatest power of data abstraction. FOOPlog notes this deficiency and so it has function-level modules to model ADTs.

4. CONCLUSION

As we have seen, there are at least four ways to integrate the three paradigms:

(1) Construct an interface of two or more languages. Examples are LEAF, Bridge and G.
(2) Extract useful characteristics from different paradigms and inject them into a dominant paradigm. Examples are DLP and FUNLOG.

(3) Redefine an existing language in the context of new theoretical insights and goals. Examples are LIFE and HOPE.
(4) Start from scratch on a consistent formal basis. Examples are Paradise and UNIFORM.

The degree of coupling of paradigms increases from (1) to (4). Obviously, (1) is the most practical but the least theoretical one among them. In contrast, (4) is consistent and elegant in theory but it requires a lot of effort to implement the system, and to attract a user community. Thus (2) and (3) can be viewed as two balance points between theory and practicality of the design of a multiparadigm language. Actually, we have found that most of the multiparadigm languages discussed in this paper follow ways (2) or (3).

Regardless of the way of integration a multiparadigm language employs, it is always wise to use the object-oriented paradigm as the language's framework. This approach has at least three advantages:

- Deficiency of history sensitivity in logic and functional programs can be augmented by the concept of objects.
- The modeling capability in object-oriented programs is more general and powerful than those in functional and logic programs.
- The highly modularized structure of object-oriented programs favors the adoption of other paradigms in a systematic way.

Furthermore, it is pointed out in [Weg90] that harmonious integration of multiple paradigms has often failed at the language level because of the unavoidable divergence between paradigms. So, integration should be done at the object or class level with loose coupling among paradigms.

Lastly, although all the three paradigms are suitable candidates for parallel programming, the possibility of parallel execution is not often discussed in literature concerning multiparadigm languages. Currently, most of the effort is concentrated on the integration part only. However, the capability of parallel programming is noteworthy in multiparadigm languages. We have also found that it is hard to evaluate a language based solely on its descriptions, even accompanied with detailed syntax and semantics specifications. The lack of large scale applications in most of the multiparadigm languages is a major obstacle to the evaluation process.

ACKNOWLEDGMENTS

We would like to thank the anonymous referees for their helpful comments. This research was supported in part by a Croucher Foundation Studentship.

REFERENCES

[Ale93] Alexiev, Vladimir. A (Not Very Much) Annotated Bibliography in Integrating Object-Oriented and Logic Programming. Internal Report, CS Department, University of Alberta, 1993.

[AK91a] H. Ait-Kaci. A Glimpse of Paradise. In J.W.Schmidt and A.A.Stogny, editors, *Next Generation Information System Technology*, pp.17-25. Springer-Verlag, 1991.

[AK91b] 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.

[AKMR92] H. Ait-Kaci, Richard Meyer and Peter Van Roy. *Wild LIFE A User Manual (Draft)*. Digital Equipment Corp. 1992.

[BBLM84] Barbuti, R., Bellia, M., Levi, G., and Martelli, M. On the integration of logic programming and functional programming. In *Proc. of International Symposium on Logic Programming*, IEEE, 1984, pp. 160-166.

[BBLM85] 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.

[BL86] Macro Bellia and Giorgio Levi. The Relation between logic and functional languages: a survey. *J. Logic Programming*, 1986 Vol.3, pp. 217-236.

[BSW91] M. Beaven, R. Stansifer, and D. Wetklow. A functional langauge with classes. *LNCS 507*, 1991, pp. 364-370.

[CL90] Antonio Corradi and Letizia Leonardi. Parallelism in Object-Oriented Programming Language. In *Proceedings of the 1990 International Conference on Computer Languages*, pp. 271-280.

[CM84] W.F. Clocksin and C.S. Mellish. *Programming in Prolog, 2nd edition*. Springer-Verlag, 1984.

[Coh85] 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.

[Con87] Conery, J.S. *Parallel Execution of Logic Programs*. K.A.P., 1987.

[Dav88] A. Davison. Polka: A Parlog object oriented language. Internal report, Dept. of Computing, Imperial College, London 1988.

[DFP85] 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.

[Eli92] Anton Eliens. *DLP A Language for Distributed Logic Programming Design, Semantics and Implementation*. John Wiley & Sons, 1992.

[FH88] Antony J. Field and Peter G. Harrison. *Functional Programming*. Addison Wesley, 1988.

[Gla92] John Glauert. Parallel Implementation of Functional Languages Using Small Processes. In *Proceedings Int. Workshop on Parallel Implementation of Functional Languages*, 1992.

[GM85] Joseph A. Gognen and Jose Mesegner. EQLOG: Equality, Types, and Generic Modules for Logic Programming. In DeGroot, D., and Lindstrom, G., editors, *Logic Programming : Relations, Functions, and Equations*. Prentice-Hall, Englewood Cliffs, N.J., 1985, pp. 295-364.

[GM87] 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.

[Gol84] A. Goldberg. *Smalltalk-80: the interactive programming environment*. Addison-Wesley, 1984.

[Gre87] Steve Gregory. *Parallel Logic Programming in PARLOG*. Addison-Wesley, 1987.

[GWB91] 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.

[Hai86] Brent Hailpern. Guest editor's introduction to multiparadigm languages. *IEEE Software*, January 1986, pp. 6-9.

[Har91] Rachel Harrison. Pure Functional Languages and Parallelism. Ph.D. Thesis, Dept. of Electronics and Computer Science, University of Southampton, March, 1991.

[HM90] 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.

[Hud89] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, Vol. 21, No.3, pp.359-411, Sept. 1989.

[IC90] 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.

[IT87] 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.

[Jon87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[Jon89] Simon L. Peyton Jones. Parallel implementation of functional programming languages. *The Computer Journal*, Vol. 32, No.2, 1989, pp. 175-186.

[Kah82] Kahn, K.M. Intermission-Actors in Prolog. In Clark, K.L. and Tarnuland, S. A. *Logic Programming*. Academic Press, 1982, pp. 213-230.

[Kah85] 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.

[KE88] 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.

[KTMB87] 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.

[Lie87] Lieberman, H. Concurrent Object-Oriented Programming in Act 1. In Yonezawa, A., and Tokoro, M., editors, *Object-Oriented Concurrent Programming*. MIT Press, 1987, pp. 9-36.

[Llo84] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1984.

[McC92] Francis G. McCabe. *Logic and Objects*. Prentice Hall, 1992.

[Mey90] Bertand Meyer. *Introduction to the Theory of Programming Languages*. Prentice Hall, 1990.

[Mil90] R. Milner. *The definition of Standard ML*. MIT Press, 1990.

[MN86] P. Mello and A. Natali. Programs as collections of communicating Prolog units. *LNCS 213*, Springer-Verlag, 1986, pp. 274-288.

[Nar90] Narain, S. Lazy evaluation in logic programming. In *Proc. of International Symposium on Logic Programming*, IEEE, 1990, pp. 218-227.

[Pla91] Placer, J. The multiparadigm language G. *Computer. Lang.* Vol.16, No.3/4 (1991), 235-258.

[Qui91] Quintus Corporation. *Quintus Prolog 3.1 Reference Manual*. Quintus Corporation, 1991.

[RS82] Robinson, J.A., and Sibert, E.E. LOGLISP: Motivation, Design and Implementation. In Clark, K.L., and Tarnuland, S. -A., editors, *Logic Programming*. Academic Press, 1982, pp. 299-314.

[Sha87] E. Shapiro, editor, *Concurrent Prolog: collected papers*. MIT Press, 1987.

[Sha89] E. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, Vol.21, No.3, Sept.89, pp.412-510.

[Ste87] L. Sterling. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, 1987.

[Str91] B. Stroustrup. *The C++ programming language 2nd edition*. Addison-Wesley, 1991.

[SY84] Subrahmanyan, P.A. and You, J. -H. Conceptual basic and evaluation strategies for integrating functional and logic programming. In *Proc. of International Symposium on Logic Programming*, IEEE, 1984, pp. 144-153.

[SY85] Subrahmanyan, P.A., and You, J. -H. FUNLOG : A computational model integrating logic programming and functional programming. In DeGoot, D., and Lindstrom, G., editors, *Logic Programming : Relations, Functions, and Equations*. Prentice-Hall, Englewood Cliffs, N.J., 1985, pp. 157-197.

[Szy91] B.K. Szymanski, editor, *Parallel Functional Languages and Compilers*. ACM Press, 1991.

[Tak92] Akikazu Takeuchi. Parallel Logic Programming. John Wiley, 1992.

[Tur90] D. Turner. An Overview of Miranda. In Turner, D., editor, *Research Topics in Functional Programming*. Addison-Wesley, 1990, pp. 1-16.

[Weg90] Peter Weger. Concept and paradigms of object oriented programming. *OOPS Messager*, Vol. 1, Number 1, pp. 7-87 Aug.90.

[WH89] P.H. Winston and B.KP. Horn. *LISP 3rd edition*. Addison-Wesley, 1989.

[Wu91] Shaun-inn Wu. Integrating Logic and Object-Oriented Programming. *OOPS Messager*, Vol. 2, No.1, pp.28-37, Jan. 91.

[Xer88] Xerox Corp. *XEROX LOOPS REFERENCE MANUAL*. Xerox Corporation, 1988.

[Yon90] Yonezawa, A., editor, *ABCL-- an object-oriented concurrent system*. MIT Press, 1990.

[YJB91] 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.

[YT87] Yonezawa, A., and Tokoro, M., editors, *Object-Oriented Concurrent Programming*. MIT Press, 1987.

[Zan84] C. Zaniolo. Object-Oriented Programming in Prolog. In *Proceedings of 1984 IEEE Symposium on Logic Programming*, pp. 265-270.