# Power to Collections!

## Generalizing Polymorphism by Unifying Array Programming and Object-Oriented Programming

Stéphane Ducasse
Software Composition Group
University Of Bern
Bern, Switzerland
ducasse@iam.unibe.ch

Philippe Mougin
100 Rue du Moulin
04220 Sainte-Tulle, France
pmougin@acm.org

Preprint of the International Ecoop 2003 on Object-oriented Language Engineering for the Post-Java Era

## ABSTRACT

Array programming shines in its ability to express computations at a high-level of abstraction, allowing one to manipulate and query whole *sets* of data at *once*. This paper presents the OOPAL model that enhances object-oriented programming with array programming features. The goal of OOPAL is to determine a minimum set of modifications that must be made to the traditional object model in order to take advantage of the possibilities of array programming. It is based on a minimal extension of method invocation and the definition of a kernel of methods implementing the fundamental array programming operations. The model is validated in F-SCRIPT, a new scripting language.

## Keywords

Array Programming, Object-Oriented Programming, Smalltalk, APL, F-SCRIPT, Query Language, High-Order Messages, Programming Language Design

## 1. THE PROBLEM

While object-oriented programming offers high-level tools for data *modeling* (abstract data type, polymorphism, inheritance), the same is not true for data *manipulation*. The basic operation provided for object manipulation is message sending, a fundamental operation as it supports polymorphism and encapsulation, but which remains a very fine-grained low-level operation. Indeed object-oriented programming does not offer a high-level model for manipulating whole sets of data (*i.e.*, collections of objects). In high-level programming models like array programming or relational algebra, complex expressions manipulating entire sets of data are easy to design and are expressed in an extremely compact manner, without requiring the explicit use of loops, tests, or navigation instructions in a data graph. The same expressions would require several tens or hundreds of lines of code in conventional object-oriented programming languages.

Our aim is to provide a new approach which offers higher-level capacities for data manipulation within the scope of object-oriented programming. For this purpose, we enrich object-oriented programming with concepts taken from array programming in a model called OOPAL (which stands for Object-Oriented Programming and array programming Language integration). This paper presents OOPAL and its implementation in F-SCRIPT, a object-oriented scripting language using a Smalltalk syntax. This article makes the following contributions: it identifies design principles for successful integration between object-oriented programming and array programming, defines the OOPAL model that enables this integration, and shows how it is validated through implementation of F-SCRIPT.

A long version of this paper is [14] to which the interested reader can find implementation notes. We start by a brief presentation of array programming (Section 2). An example in traditional object-oriented language is compared with its equivalent in F-SCRIPT (Section 3). The OOPAL model is described as three components: message patterns (Section 4), mapping between basic array programming and OOP concepts (Section 5), and array programming operations (Section 6). The OOPAL model is used, through F-SCRIPT, in various fields such as data analysis, game development, or software debugging [14].

## 2. ARRAY PROGRAMMING OVERVIEW

Array programming is the result of a mathematical notation invented by Ken Iverson. APL, the first array programming language, was developed by IBM in the 1960s and earned Ken Iverson the Turing Award [10] [1].

### 2.1 Principles

Array programming has two key characteristics:

- Operations can be directly applied to entire arrays of values.
- A set of special functions and operators provides powerful means of data manipulation and allows complex data manipulation processes to be expressed concisely.

The fundamental principle behind array programming is that operations are directly applied to entire arrays of values, without the need

for explicit loops. For example, if X and Y are two arrays of numbers, X+Y returns a new array which contains the sum, element-wise, of X and Y as shown by Figure **??**. It is also possible to combine scalars and arrays in the same expression. For example, X*2 returns an array which contains the result of multiplying each element in X by two. In an array programming language such as Fortran 90, the expression Z = W+X*SIN(Y) is legal, not only when W,X, Y, and Z are scalars, but also when they are arrays [6].

## 2.2 Array Programming Building Blocks

More precisely, array programming is built around scalars, multi-dimensional arrays, functions, and operators.

**Scalars.** Conventional array programming languages are oriented towards numeric computing. They offer a certain number of basic data types, including numbers and characters, and sometimes offer complex numbers, dates, and Booleans.

**Multi-Dimensional Arrays.** Traditionally, array programming supports multi-dimensional arrays and lets one specify the dimension onto which operations are carried out. For instance, in APL, one can index the operation symbol with a number specifying the dimension using [ and ]. Thus, in the expression X/[Z]Y, the array Y is compressed along its Zth dimension using the Boolean array X.

**Functions and Operators.** Furthermore, array programming features a set of powerful functions and operators which enable data manipulation to be expressed in a concise manner. As pointed out by the ACM SIGAPL, "The APL primitives express broad ideas of data manipulation. These rich and powerful primitives can be strung together to perform in one line what would require pages in other programming languages".

For instance, these primitives allow one to: select some elements in an array, cumulatively apply a function to the elements of an array, transpose the elements of an array, slice an array, rotate an array, reshape an array, or combine array elements with generalized outer and inner product operations. These array programming operations are in synergy with the automatic processing of arrays and offer great power for data manipulation. For example, *compression* may be used to select elements in an array which meet a particular criterion. Thus, in APL the expression (X < 60) / X selects elements from X which are less than 60 [17].

# 3. AN EXAMPLE

To illustrate our point we use the following object model in several examples throughout the paper. It presents the minimal model of an airplane company and represents flights, airplanes, pilots, and their relationships (an airplane and a pilot are associated with each flight).

We are only interested in manipulating objects via their behavioral interface. Thus, our UML schema does not specify the instance variables but only the methods. We define three collections of objects named A, F, and P which group together the references to all instances of airplanes (A), flights (F), and pilots (P) in our airplane company's fleet.

Suppose that we want to print the names of all the pilots, ranked by salary in increasing order, in charge of a flight to Paris on a B747 airplane. Using Java or other mainstream object-oriented languages

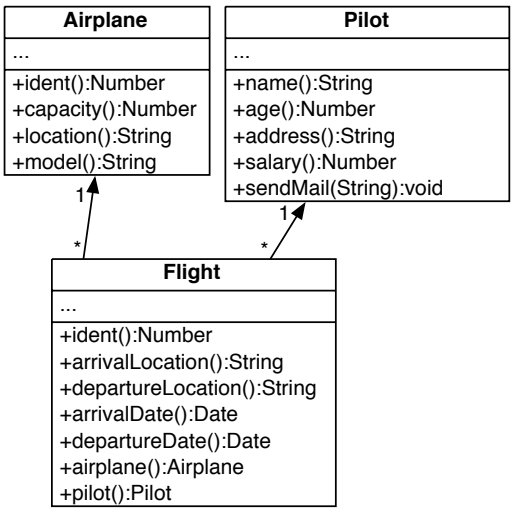


**Figure 1: A simple domain: airplanes, flights, pilots and their relationships.**

we would have to write this kind of code[1].

**Java.**

```
TreeSet pilots = new TreeSet(new Comparator()
{
   public int compare(Object o1, Object o2)
   {
      if (((Pilot)o1).salary() < ((Pilot)o2).salary())
         return -1;
      else if (((Pilot)o1).salary() == ((Pilot)o2).salary())
         return 0;
      else
         return 1;
   }
});
Iterator i = F.iterator();
while (i.hasNext())
{
   Flight currentFlight = (Flight)i.next();
   if (currentFlight.arrivalLocation().equals("PARIS")
   && currentFlight.airplane().model.equals("B747"))
   {
      pilots.add(currentFlight.pilot());
   }
}
i = pilots.iterator();
while (i.hasNext())
{
   System.out.println(((Pilot)i.next()).name());
}
```

Our solution, named the OOPAL model, is based on the unification of array programming and object-oriented programming. The main idea is that this unification lets one develop code using the object interfaces and still use the power of array programming. The unification is realized by the introduction of *message patterns* and array programming operations (See Section 4). As the complete

[1]By using a TreeSet we can both sort the selected pilots by salary in increasing order and avoid duplicates.

OOPAL model is implemented in a new Smalltalk dialect named F-SCRIPT, we use it as a notation to express the examples.

As we present in Section 4, the OOPAL model is based on *message patterns* that support the manipulation of object collections in an array programming-like fashion.

With F-SCRIPT, the same operation on the same object model is expressed as follows:

```
F-SCRIPT.
pilots := (F at: F arrivalLocation = 'PARIS'
          & (F airplane model = 'B747')) pilot distinct.
sys log:(pilots name at:pilots salary sort)
```

The expression F arrivalLocation = 'PARIS' returns an array of Booleans that indicates, for each flight, whether or not the arrival location is 'PARIS'. Such an array combined with the other expression F airplane model = 'B747' is then used by the at: method to select the corresponding flights.

# 4. MESSAGE PATTERNS

Now, we shall present the core aspect of OOPAL, *message patterns*[2]. As we have mentioned, message patterns are at the heart of the minimal extension of the traditional object model. Message patterns support encapsulation, extensibility, a minimal extension of the object model, and the handling of array specific messages and nested arrays. They also avoid the addition of constraints on object interfaces. Therefore *any* kind of object and *any* kind of method can be manipulated in an array programming-like fashion. The model is completed with a set of specific array programming operations that have been adapted to object-oriented programming.

## 4.1 Extended Message Passing

As explained, array programming enables operations to be applied to entire arrays without requiring the explicit use of a traditional loop control structure. OOPAL message patterns extend the traditional message passing operation backward-compatibly to support messages to be sent to collection of objects. Message patterns allow for the sending not only of a simple message, but of a complex group of messages.Traditional message passing then conceptually becomes a specific case of message pattern. Note that as regards speed of execution, normal method invocation is not impacted by message patterns.

In this paper we present the principal elements of message patterns. A complete description can be found in the F-SCRIPT User's Manual [12].

Message pattern notation makes it possible to:

- Specify for each array involved (*i.e.*, receiver and arguments) in a message pattern whether a loop should iterate over the elements of this array. Moreover, the nesting level of this loop in relation to other loops in the message pattern can be specified.
- Specify a different message pattern for each level of array nesting, should nested arrays be used.
- Use *implicit message pattern* which makes notation easier.

[2]"Message pattern" is sometime used in Smalltalk as a synonym for selector with method argument. This differs from our usage in this paper.

We also propose an abstract notation that makes it possible to express the structural properties of message patterns, regardless of their actual message selectors and arguments. These structural properties are called patterns. For example, the notion of an outer product corresponds to a particular pattern.

## 4.2 Simple Message Patterns

The message pattern notation involves indicating iterations in the message expression itself. When the @ symbol is placed after the message pattern receiver and/or just before an argument, it means that a loop iterating over the elements of such designated array(s) is executed to generate message sends. The two following rules using a Smalltalk syntax express it[3] (note that these two rules can be combined, as we show later):

- rec @max: arg means that all the elements of rec receives the message max: with the argument arg. {1,2,3} @max: 2 returns {2,2,3}.
- rec max:@ arg means that rec receives the message max: for each elements of arg. 2 max:@ {1,2,3} generates three message sends (*i.e.*, 2 max:1, 2 max:2, 2max:3) and returns {2,2,3}.

For example, the expression F @airplane sends the message airplane to all the elements of F and returns the resulting array (*i.e.*, an array of airplane objects, having the same size as F, where the airplane at index i in this array is the airplane associated with the flight at index i in F).

Both receiver and arguments can be iterated upon as shown by the following example: {1,2,3} @+@ {10,20,30} returns the array {11,22,33}

The messages generated as the result of the execution of a message pattern are sent sequentially. Arrays are iterated from the start of the array towards the end. The method invocation semantic is not altered by OOPAL and is defined by the object-oriented language at use.

**Composition.** A message pattern has the same precedence level as conventional message sends and can be composed in a similar manner. For example, the expression F @airplane @model generates the airplane array described in the previous example, and then sends the model message to each element in this array, which returns a new array giving the airplane model corresponding to each flight. The expression F @departureDate @> NSDate now returns a Boolean array which indicates, for each flight, whether it will take place after the current date [4].

**Advanced Combinations.** Thus far, we have seen that by using patterns on some arrays we generate messages that use the first element of each array, then the second, and so on. But we can also combine array elements in other ways. For example, suppose we want to get the outer product of X and Y, using the * method (Figure 2). To do this, we have to specify that we want a loop on X and

[3]In Smalltalk syntax invoking the method max: between two numbers is a max: b (equivalent to a.max(b) in Java).

[4]NSDate is the class representing the dates in F-SCRIPT and now is a class method returning the current date. The NSDate class implements conventional comparison methods and in particular the method > .

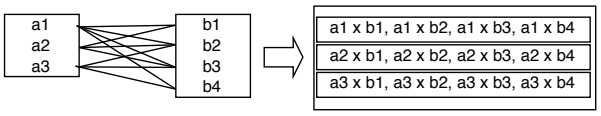


**Figure 2: Outer product.**

an inner loop on Y. We therefore use a number after the @ symbol to state the inner level of each loop. The outer product as shown in Figure 2 is then expressed as X @1*@2 Y.

For instance, if X is {1,2,3} and Y is {10,20,30}, then X @1*@2 Y returns {{10,20,30},{20,40,60}, {30,60,90}}.

If Z is {2,0,4} then X @1 between:@2 Y and:@3 Z returns {{{false, true, false}, {false, true, false}, {false, true, false}}, {{true, true, false}, {true, true, false}, {true, true, false}}, {{true, true, false}, {true, true, false}, {true, true, false}}}.

X @1 between:@2 Y and:@1 Z returns {{false, false, false}, {true, true, true}, {false, false, false}}

## 4.3 Implicit Message Pattern Notation in F-Script

The notation introduced previously supports the definition of message patterns and is fully executable in F-SCRIPT. However, the explicit notation is still too complex compared with conventional array languages. In APL we write X+Y to add the arrays X and Y, with the notation presented above we must write X @+@ Y.

To solve this problem, we introduce an implicit notation for message patterns[5], which in most cases eliminates the explicit use of @. Going back to our example, we can now write X + Y like in APL. This notation requires the language in which it is used to be able to recognize the specification of a message pattern even if the message pattern is not denoted by any particular signs.

In F-SCRIPT, we use the following rule to support implicit message patterns. When a message is sent to an array, if the method invoked is not defined by the class Array, the message is sent to the receiver elements. Moreover, if certain arguments of this message are also arrays F-SCRIPT takes the elements in these arrays one at a time.

When X, Y, and Z are arrays of numbers, this rule makes it possible to write expressions such as: X+Y, X*2, X cos, X between:Y and:Z, and X between:10 and:Z. These expressions follow the array semantics *i.e.*, they operate on the array elements. The rule also makes it possible to use the implicit notation for recursive message patterns. For instance {{1,2,3},{10,3.14}} * 2 is equivalent to {{1,2,3},{10,3.14}} @@* 2 and evaluates to {{2,4,6},{20,6.28}}

Thanks to implicit notation we can simplify the examples given in the previous sections. Thus, we can write F airplane instead of F @airplane and F airplane model instead of F @airplane @model. However, implicit notation cannot always be used, especially when the receiver is not an array but arguments should be iterated upon. For instance, the expression 2 * X where X is an array of numbers will not work. In this case, an explicit message pattern is required: 2 *@ X.

[5]Note that all message patterns specified implicitly can also be specified in an explicit manner.

Note that the proposed rule can be implemented in the message-sending routine with nearly no performance loss.

# 5. BUILDING BLOCKS

The OOPAL model would not be complete without showing how the basic elements of array programming are mapped into object-oriented programming.

## 5.1 Scalars

All the basic data types (*e.g.*, Booleans, numbers) manipulated by conventional array programming languages corresponds in our model to classes. Data is no longer limited to certain types but may be of any class. This is possible because, in OOPAL, array programming facilities are universal and not linked to any particular class.

## 5.2 Multi-dimensional Arrays

Array programming allows data to be grouped together in *multi-dimensional* arrays which vary in size, are heterogeneous, and may themselves contain arrays.

In object-oriented languages the closest type of array is ordered collection (*e.g.*, ArrayList in Java, OrderedCollection in Smalltalk). OOPAL uses the extensibility of OOP to define the main array-programming operations as dedicated methods on collection classes. In this article, we refer to these enriched collections as "arrays".

Conventional object collections are one-dimensional (called vectors in array programming languages) and do not offer direct support for the multi-dimensional manipulations which are so dear to array programming. Nonetheless, we believe that it is preferable in our context to avoid introducing the notion of multi-dimensional object collections for three reasons.

- First, it would involve a substantial addition leading to numerous complexities and this would contradict the minimal extension principle.
- Second, while multi-dimensional matrix manipulation may be a common occurrence in mathematical processes it is much less useful in our context which looks at the manipulation of any type of objects.
- Third, there is no loss in expressive power. We are not abandoning advanced support for operations requiring multi-level object nesting. First, collection nesting is still possible, as nothing prevents a collection from containing collections. Second, the notion of recursive message patterns allows array programming techniques to be used on nested collections regardless of the depth of nesting.

## 5.3 Functions and Operators

Array programming languages offer a certain number of operations which may be applied to scalar data and/or to arrays. Such operations are called *functions*. Functions may be primitive *i.e.*, provided by the language such as basic mathematical operations, Boolean operations, or array-specific functions such as compression[6]. They may also be *defined i.e.*, provided by the developer or via external libraries. In OOP, functions map naturally to methods.

Array programming also uses the notion of *operators*. An operator can be described as a function which applies to other functions to

[6]Compression means selecting elements of an array by providing an array of Booleans. F at: F arrivalLocation = 'PARIS'.

produce a function. Array programming is widely based on operators for data manipulation. For example, the reduction operator enables any function with two arguments to be cumulatively applied to all the elements in an array. As with functions, primitive operators exist and it is possible to define new operators.

Object-oriented programming does not have any direct equivalent to the operator concept but the main advantages of operators can be obtained by implementing methods which use *operations* as arguments. Depending on the language, one can use constructions such as blocks (lambda functions) or selectors in Smalltalk, anonymous methods in C#, selectors in Objective-C, or lambda functions in Python, CLOS. For example, an equivalent to a reduction operator (in a slightly modified form) already exists in Smalltalk with the inject:into: method, which takes a block as an argument.

# 6. ARRAY PROGRAMMING OPERATIONS BY EXAMPLE

The notion of message pattern alone is not sufficient to fully integrate array programming and object-oriented programming. Message patterns become powerful when they are associated with specific array programming operations. Note that these operations are implemented as simple methods. No new syntactic notation is necessary. These operations are generic and are meant to be provided by languages supporting the OOPAL model. They do not require support from the developer.

In particular, these operations enable:

- Easy navigation in the object graph,
- Concise and readable expression of the selection of objects according to arbitrarily complex criteria,
- Sophisticated data analysis, and
- Concise and readable expression of complex object manipulations.

As we illustrate now, using message patterns and these operations makes it possible to capitalize on the power of array programming principles in an object-oriented context. Here, we present only four examples of array specific operations, as implemented in F-SCRIPT, but there are many others: see [12] and [13] for a detailed list.

**Compression.** Compression selects certain elements of an array. The result of compression is a new array which contains the selected elements. Compression is an operation which requires two operands: the array from which we want to make a selection and a *Boolean array* of the same size. An element is selected if the corresponding Boolean (*i.e.*, at the same index) holds true.

For example, suppose that our array P contains height Pilot objects. If we want to select elements at index 0, 1, and 5, we can apply compression as follows[7]: P at:{true, true, false, false, false, true, false, false}.

Compression requires a Boolean array which specifies the selection that has to be made. And yet, thanks to message patterns, such arrays are very easy to generate. For example, P salary $>$ 3000, generates[8] a Boolean array which indicates, for each pilot, whether his/her salary is greater than 3000. We can then use this Boolean array to compress the P array and thus select only pilots whose salary is greater than 3000: P at: P salary $>$ 3000.

Message patterns enable complex Boolean conditions to be expressed naturally. For example, (P at: (P salary $>$ 3000) & (P address = 'PARIS')) sendMail:"Dear Pilot, etc. etc. etc." selects the pilots whose salary is greater than 3000 and who live in Paris, and sends them an email.

**Reduction.** Reducing an array consists in cumulatively evaluating a custom operation on the elements of an array. In F-SCRIPT, reduction is implemented as the method \ of the Array class. For example, you add up the elements of an array with the expression: {1,2,3,4} \ #+ which returns 10. The result is computed as if you had entered: 1 + 2 + 3 + 4.

Reduction may be used with any operation (method or block) which takes two operands and returns an object.

Some types of reduction are used very often:

- Reduction of an array of numbers using the min: method returns the smallest element in the array (alternatively, max: reduction returns the greatest element).
- Reduction of a Boolean array using the | method (*i.e.*, logical OR) enabling existential quantification. For example, P salary $>$ 3000 \ #| returns true if a pilot has a salary greater than 3000.
- Reduction of a Boolean array using the & method (*i.e.*, logical AND) enabling universal quantification. For example, P salary $>$ 3000 \ #& returns true if all the pilots have a salary greater than 3000, otherwise false.
- Reduction of a Boolean array using the + method which provides the number of objects which check a certain predicate. For example, P salary $>$ 3000 \ #+ returns the number of pilots whose salary is greater than 3000.

Reduction already exists in other object-oriented languages (*e.g.*, inject:into: method in smalltalk, reduce function in Python). However reduction is rarely used in these languages because of the verbose definition it requires and because object collections in these languages are not as omnipresent as in OOPAL. In OOPAL the reduction method works in synergy with the other specific high-order methods and idioms promoted by array programming.

**Sorting.** The result of the sort message sent to an array is an array of integers containing the indices that will arrange the receiver of sort in ascending order. For example, if X is {5,2,1,3,6,4} then X sort returns {2,1,3,5,0,4}.

You can then get X in ascending order by indexing it with the result of the execution of the sort method[9]: X at:(X sort) returns

[7] Here, we have extended the Smalltalk indexing method for carrying out compression when its argument is a Boolean array.

[8] The expression is evaluated as follows: the salary message is sent to each element in P (this message pattern is denoted implicitly), which produces an array of numbers. The message $>$ 3000 is then sent to each element in the array of numbers (here again, in accordance with the implicit notation of message patterns) which produces the desired Boolean array.

[9] In array programming, an array can be indexed by a whole array

{1,2,3,4,5,6}. The advantage of this method is that once you have the ordered index numbers, you can then apply them not only to the original scrambled array, but to any other array of the same size. For example, suppose we want to get an array of pilots sorted by ascending salary. We just have to evaluate the expression P at:(P salary sort).

**Joins.** A join is implemented by the >< method of the Array class. For each element, say e, of the receiver, this method computes an array containing the positions of e in the argument. It returns all the arrays packed into an array of arrays. {1,2,'foo'} >< {4,'foo',1,'foo','foo'} returns {{2},{},{1,3,4}}. This result means that the first element of the receiver is found in position 2 in the argument, the second element of the receiver is not present at all in the argument and the third element of the receiver is found in positions 1, 3, and 4 in the argument.

Suppose we want to obtain, for each pilot in P, the list of all the flights that the pilot is responsible for. That is, we want to construct an array of the same size as P, where each element is an array containing the flights associated with the corresponding pilot in P. The list of flight at index *i* in the resulting array contains the flights for which the pilot at index i in P is responsible. As shown in Figure 1, the Pilot class does not provide a method (say, "flights") returning the list of flights associated with a pilot instance. Thus we cannot just write something like P flights. So, how can we navigate from the pilots to the associated flights? It is in this kind of situation that the join method presented here is useful. This method can be used here because the Flight class provides a method (named pilot) that makes it possible to navigate from a flight to its associated pilot. The result can be obtained by combining the join method, an extended indexing, and two message patterns: F at:@ P >< F pilot.

As shown here, the join method makes it possible to easily navigate an object graph without requiring the object model to maintain and provide inverse relationship (*i.e.*, back-links) navigation capacities. Used wisely, this characteristic can simplify the implementation of an object model and allow one to express ad-hoc queries.

# 7. RELATED WORK

Several team teams have explored integration of object technology and array programming. In most cases, their goal has been to study how object technology could enhance an existing array language. See [5] [11] [8] [2] [7] [3] [16]. OOPAL takes a rather different approach as it involves bringing the notions of array programming to the object world. Its goal is therefore to determine the minimal set of modifications that must be made to the traditional object model in order to take advantage of the possibilities of array programming. To our knowledge, little research has been carried out in this area.

Several libraries provide array programming operations for existing object-oriented languages. For instance SmartArray [4] provides an advanced array programming library for C++, Java, and C#. Other examples include the widely used Numarray and Numerical Python packages [9] which add array programming operations to Python. The fundamental difference between the OOPAL model and such libraries is that these libraries are oriented towards numeric computing, whereas the OOPAL model attempts more fundamental integration between object-oriented programming and array programming to support the manipulation of any kind of object.

In the database realm, the confrontation between object-oriented databases and relational databases has led to the development of object query languages, which provide high-level object-oriented querying models. However, these developments, at least for the moment, have not influenced mainstream object-oriented programming languages such as C++, Java, C#, or Smalltalk. On the contrary, in their most recent incarnations (*e.g.*, JDO Query Language, EJB Query Language), object query languages adopt quite a low profile. They are merely used as a minimalist interface to an underlying database, for bootstrapping the data navigation and manipulation process (*i.e.*, getting some elements of the object graph out of the database) which is then carried out with the host object-oriented programming language. While object query languages are mainly designed around the interaction with persistent objects stored in a database, the OOPAL model is primarily designed to interact with instantiated objects lying in memory. Indeed, the problem we are tackling with OOPAL is to provide a higher-level programming model in the context of OOP, not a database query language. One important consequence of this difference in focus is the support for encapsulation. For performance reasons, most of the query languages provided by object-oriented databases or object/relational mapping tools break encapsulation. For example, the current version of JDOQL [15] does not support method invocation of business objects but only offers direct access to instance variables. The performance problem that justifies breaking encapsulation is due to the fact that database query languages do not actually manipulate objects but object representations stored on disk, which is a totally different thing. In such a case, the use of indices and the minimization of instantiations, which can be achieved by breaking encapsulation, is needed to achieve good performances. On the contrary, in the context of a general purpose object-oriented language, encapsulation is of paramount importance, and is well supported by OOPAL.

High-level object-oriented languages like Smalltalk or Python commonly provide sophisticated collections classes associated with high-level operations. OOPAL's association of message patterns and array programming operations subsume these models by allowing to express object manipulations in a more readable way and with much less code. [12] shows examples of how classical Smalltalk's high-level operations on collections are subsumed by OOPAL in F-SCRIPT. A key difference between the two approaches is that OOPAL allows one to think in terms of whole sets of objects, while conventional object collection protocols require thinking in terms of iteration over collection elements. For instance, compare the code needed to generate an array containing the names of all the pilots:

- Smalltalk using the collection protocol: P collect:[:aPilot | aPilot name].
- Python using the list comprehension construct: [aPilot.name() for aPilot in P].
- F-Script using OOPAL: P name.

To finish we want to stress one key point about the OOPAL model. This model provides a new conceptual tool for writing and designing program. The OOPAL model does not limit itself to a syntactical notation. It frees the developer to think in terms of individual

of indices. The translation of this capacity in our OOPAL model involves the array class providing an indexing method able to deal with an array of indices.

entities but in terms of complete set of objects. To a certain extent it acts as the natural generalization of polymorphism.

## 8. CONCLUSION

On one hand, object-oriented programming provides excellent support for data *modeling* but it falls short for the expression of complex expressions over *entire sets* of objects. The lack of expressiveness of the object-oriented approach was one of the reasons why some relational advocates saw object-oriented databases as a twenty-year step backwards. On the other hand, array programming supports the manipulation of entire sets of data and avoids the use of explicit loops, but does not support objects.

This article presented the OOPAL model that defines the integration of array programming in object-oriented programming. It is based on an extension of the concept of method invocation which implies a slight syntactic extension, and the addition of certain array manipulation methods, which do not call for the modification of the target language's syntax.

The manner in which F-SCRIPT handles encapsulation, inheritance, and polymorphism has not been addressed in this paper due to the similarity with other dynamic language such as Smalltalk. In the OOPAL model these concepts are orthogonal to array programming and don't need to be specifically re-designed to fit into the world of array programming. This point is clearly shown by the fact that F-SCRIPT allows standard Objective-C objects to be manipulated.

In OOPAL array programming and OOP fit well together. Not only can these two programming models benefit from each other's advantages, but they act synergically. While array programming makes it possible to write code with few explicit loops, OOP's dynamic binding decreases the use of explicit conditional control structures. It just so happens that the need to use explicit conditional control structures usually hinders the use of array programming, because it makes it harder to express array-oriented algorithms. OOP, which favors a programming style that is free from these structures, naturally allows for extended use of array programming.

By integrating array technologies into object-oriented languages, the OOPAL model allows OOP to benefit from decades of intensive research in the field of array programming languages. This provides a high-level notation which makes it possible to easily express complex object manipulations. The real-world applications developed with F-SCRIPT have shown that the integration of array technologies with object-oriented programming adds a considerable amount of power to the latter.

**Acknowledgments.** We would like to thanks Noury Bouraqadi, Jörg Garbers, Ralph Johnson, Joseph R. Kiniry, Brid Marire, Oscar Nierstrasz, Hannah Riley, and Roel Wuyts for the their feedback on early versions and parts of this article.

## 9. REFERENCES

[1] P. Berry. Apl 360 primer, 3rd edition. Technical report, IBM Corp, 1971.

[2] B. Best. Object-oriented programming and apl computer-language. http://www.benbest.com/computer/oopapl.html.

[3] P. Bottoni, M. Mariotto, and P. Mussio. Liseb: a language for modeling living systems with apl2. In *Proceedings of the International Conference on APL*, number 1292, pages 7–16, 1994.

[4] J. A. Brown and J. G. Wheeler. Smartarrays for the apl programmer. In *Proceedings of the International Conference on APL*, pages 50–60, 2002.

[5] R. G. Brown. Object-oriented apl: an introduction and overview. In *Proceedings of the International Conference on APL*, pages 47–54, 2000.

[6] G. Fox and N. McCracken. Array programming in fortran 90. www.npac.syr.edu/EDUCATION-/PUB/hpfe/module1/index.html.

[7] M. Gfeller. Object-oriented programming in aida apl. In *Proceedings of the International Conference on APL*, pages 164–168, 1989.

[8] J. J. Girardot and S. Sako. An object oriented extension to APL. In *Proceedings of the international conference on APL*, pages 128–137. ACM Press, 1987.

[9] Greenfield et al. Numarray, an open source project, 2002. http://stsdas.stsci.edu/numarray/numarray.pdf.

[10] K. I. Iverson. *A Programming Language*. John Wiley and Sons, 1962.

[11] R. MacDonald. Bob brown and object oriented apl. http://www.torontoapl.org/ga/ga9605/bbrown.txt.

[12] P. Mougin. F-Script guide, 1999. http://www.fscript.org.

[13] P. Mougin. High-level object-oriented programming with array technology. In *Proceedings of the International Conference on APL*, pages 163–175. ACM Press, 2000.

[14] P. Mougin and S. Ducasse. Oopal: Integrating array programming in object-oriented programming. 2003. Accepted to OOPSLA'2003.

[15] Sun-Microsystems. Java data object specification,1.0, 2002.

[16] N. J. Thomson. *J - The Natural Language for Analytic Computing*. Research Studies Press, 2001.

[17] R. G. Willhoft. Comparison of the functional power of apl2 and fortran90. In *Proceedings of the International Conference on APL*, pages 343–357, 1991.