

High-level Object Oriented Programming with Array Technology

Philippe Mougín

pmougín@acm.org

The original version of this paper has been presented at the International Conference on Array Programming Language (APL-2000 Berlin). Revised version: May 21, 2003.

Abstract

Although classical object-oriented programming languages provide high-level modeling capacities (abstract data type, inheritance etc.), they remain low-level when it comes to data manipulation. Addressing this problem with object oriented programming languages is an important mission of today's research. Considerable work has already been done, leading to the development of tools such as object query languages, with mixed results. In this paper, we present the key points of a new approach to this problem. We propose an enhancement of object oriented programming at the core level, by integrating Array Programming, a high-level model for computing. Our solution is based on an extension to object oriented programming. This integration of object technology and Array Programming allows for high-level object-oriented programming. At the same time, it opens Array Programming to the powerful world of objects.

Keywords: Array Programming, APL, object-oriented programming, F-Script, Smalltalk, object query language, high order messaging.

ACM COPYRIGHT NOTICE. Copyright © 2000-2002 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept, ACM Inc., fax +1 (212) 869-0481, or permissions@acm.org.

1 Introduction

Object Technology (OT) has taken the world by storm, and today constitutes the basis of many development efforts. OT was invented at Xerox Park by Alan Kay in the late 1960s (with important ideas coming from the Simula language¹), and is modeled after biological systems. In fact, the way objects communicate by exchanging messages is directly inspired by the communication between living cells. This revolutionary technology and its implications have yet to be fully understood. Despite its great success, it has its own set of internal problems and failures.

We will take a look at object technology's failure to replace relational technology at the heart of corporate information systems. We must acknowledge the fact that when choosing a core technology many corporations still opt for relational rather than object technologies. Throughout the industry, most of the information systems built for corporations are based upon relational databases. More importantly, the use of object/relational (O/R) mapping is still marginal. This means that in most ISs developed today, the core paradigm for modeling and manipulation of business entities is the relational paradigm. We use OO languages, mostly Java, to build these systems. However, without O/R mapping, business entities are manipulated explicitly using relational algebra via calls to middleware such as JDBC. Indeed, the current commercial failure of Object-Oriented Database Management System (OODMS) is an argument to consider.

In fact, when relational proponents argued years ago that OODBMSs were a twenty-year step backward, they were right! This criticism prompted a major enhancement of Object Technologies (OT): the development of object query languages. This development was an answer to the lack of high-level features found in traditional object languages. This last sentence may not be very clear. Aren't "abstraction" and "high-level" hallmarks of Object Technology? To answer this question and to understand the problem addressed in this paper, we think it is useful to consider the concepts of "modeling" and "computing" separately.

We would argue that, historically speaking, OT has provided a high-level framework for modeling and a low-level framework for computing (object manipulation), especially in regard to the relational theory. OT's capacity to provide high-level modeling comes mainly from the Abstract Data Type (ADT) theory, which is fully supported by OT, as well as from extended modeling capacities like inheritance.

With the relational theory, a specific entity, such as a "client" of a corporation, is represented in one or more tuples. The possible operations applying to a single tuple are creating the tuple, destroying the tuple, reading the value of one attribute of the tuple, and setting the value of one attribute of the tuple. Conversely, with OT it is possible to define any operation, including high-level ones on entities. Indeed, we are not limited to the four operations available on tuples.

Clearly, when it comes to modeling, OT can hold its own weight against the relational theory. However, when it comes to data manipulation, OT offers a low-level paradigm: message sending. Message sending is an incredibly powerful mechanism at the core of OT. It supports polymorphism and greatly unifies concepts. Nevertheless, message sending is a very low-level concept. This is rather obvious when we compare it with relational algebra, the base paradigm for data manipulation of the relational theory that was introduced by Codd [Codd70] [Codd72]. Relational algebra defines a small kernel of fundamental operators for the manipulation of relations (i.e. whole groups of tuples) that clearly addresses the typical needs of today's information.

Addressing this historical problem with OO programming languages is an important mission of today's research in our community. Considerable

work has already been done, and has led to the development of tools like object query languages. Other solutions are currently being explored such as integrating logic programming with OT or enhancing OT by using Aspect-Oriented Programming (AOP) [AOPW99]. In the mainstream computing industry, many actors have recognized this problem and are trying to address it. For example, Software AG's new application server, Bolero, provides a new object-oriented language (an extension to Java) which integrates a subset of OQL, the object query language standardized by ODMG, in conjunction with an object/relational mapping system.

However, a pragmatic evaluation of how OT is used by corporations at the core level of their IS should encourage us to explore new solutions. Analysts admit that for the moment OODMSs have been commercial failures. Even François Bancilhon, creator of the O2 system and object database hero, clearly confirmed this situation during his keynote address at the Objet 99-Nantes conference. This would not be a problem if O/R mapping were of common use. But this is clearly not the case—even though modern O/R tools can offer excellent performance. A third possibility would be to turn to the OO features found in relational databases from such leading vendors as Oracle or Sybase. But few corporations today use the object-oriented features found in these hybrid databases.

In this paper, we try to address the problem in a new way. We propose enhancing OOP at the core level, by integrating Array Programming, a high-level model for computing.

This paper is structured as follows: Section 2 gives a brief presentation of Array programming and how it is used today. Section 3 is a brief introduction to object technology. Section 4 introduces the base elements of F-Script syntax. F-Script is a new, object-oriented scripting language implementing our high-level model. In this paper it is used as both an example and a notation. Sections 5 and 6 present the key points of our Array/object integration model. The last sections provide some additional results about the possibility offered by this model and discuss related work.

2 Array Programming

The fundamental idea behind Array Programming is that operations apply to an entire set of values. Consequently, Array programming is a high-level programming model that lets us easily operate on whole Arrays of data, without having to resort to explicit loop constructions. For instance, to add two vectors of numbers (say A and B) you simply write A+B. In an Array language such as Fortran 90 [Fox94], the expression $A=B+C*\text{SIN}(D)$ is legal, not only for scalar A, B, C, and D but also for cases in which A, B, C, and D are Arrays of exactly the same shape or scale [Willhofs91].

In APL, the compression function, combined with the basic Array capacities described above, allows you to select the elements of an Array that satisfy particular criteria. For example, the expression $(A<60)/A$ returns all of the elements of A that are lesser than 60. APL is the seminal language for Array programming. Its creator, Ken Iverson, received the prestigious Turing Award for his work. APL's expressiveness and power are recognized throughout the industry and, as pointed out by ACM SIGAPL, "...comes from its direct manipulation of n-dimensional Arrays of data. 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." More than a mere programming language, APL and its various dialects can be seen as an active branch of mathematics [Iverson].

J is a modern dialect of APL created by Ken Iverson. In [Smillie99] Keith Smillie introduces Array programming and J. Many Array languages exist. For example, we can look at K [Whitney93] and Nial [NialSystem] [Glasgow89] [JenKins89].

It is also worth noting that Array programming provides great synergy with parallel hardware [Albert88].

Today, Array technology is used in many branches of our industry, from scientific to business computing, including software and hardware design¹. Array technology itself takes many forms

and is used, for instance, in spreadsheets and in mainstream mathematical and statistical packages. Finally, it is worth noting that one of the fastest growing domains in IS today, decision support with Online Analytical Processing (OLAP), is based on Array technology (using the notion of hypercube to represent data).

3 Object Technology

Fundamentally, OT is based on two conceptual and technical pillars: the abstract data type (ADT) theory and dynamic binding. ADT enables implementation of new data types that are adapted to each particular situation and thus facilitates high-level programming. Dynamic binding technology makes it possible to organize information systems in modules (the objects) representing abstract data types that communicate dynamically between each another by sending messages. Communication by sending messages differs from simple procedure calls of non-object languages:

- Each message is sent to a specific object.
- The code executed by the object receiving the message is selected automatically and dynamically at execution according to the message name and the ADT associated with the object (i.e. the class of the object). In the case of a simple procedure call, the code is selected statically at link time. This ability of different objects to respond, each in its own way, to identical messages is called polymorphism [Larkin].

Basically, dynamic binding makes it possible to write a chunk of code that is able to process in the same way various kinds of objects adhering to a same interface (i.e. able to respond to a common set of messages). This model enables high-level, modular programming, high reuse of objects and development using frameworks. This approach enables implementation of flexible and extensible applications.

¹ For instance, [Grice] explains that IBM's famous chess "player," the Deep Blue System, was designed with

the help of array programming. IBM is a major player in the Array programming community with its APL2 product.

4 Introduction to F-Script Notation

In this section, we present the base syntax and concepts of F-Script since we will use it as a notation and example in this paper.

4.1 Base Elements

F-Script is an object-oriented scripting language [Mougin99b] we have developed. It is based on our proposed solution for high-level object computing.

F-Script is based on the object paradigm. With the exception of a few details, the base syntax and concepts of F-Script comes from those of Smalltalk². Of course, when it comes to array programming, F-Script differs significantly from Smalltalk since the latter does not support this technology.

Like Smalltalk and APL, F-Script provides an interpreted, interactive environment and supports the notion of workspace. F-Script provides a rich set of functionality including object persistence, distributed objects, graphical interface framework, integration with a GUI Builder etc. Like Smalltalk, F-Script is a pure OO language. This means that every entity manipulated with the language is an object, from a simple number to a string to an arbitrarily complex business concept. This unifies the way in which things are manipulated.

F-Script's syntax is very simple. It does not include specific notation for control structures such as repetitions or conditions. As is the case in Smalltalk, these functions, along with the possibility of defining new control structures, are provided in a unified manner by the only control structure available at the syntax level (i.e. message send). The syntactic elements are:

- The literals, which make it possible to create the instance of a particular object: numbers, tables, blocks, character strings, booleans, etc. (Ex: 2.57, 'a string', true, false, {1,2,3,4}).

- The assignment of a variable to an object reference (Ex: x: = 3.14).
- Sending messages (« message pattern »)
- Comments (Ex: "a comment")
- Paranthezation (Ex: 3*(4+10)).
- Delimitation of instructions with the separator "." (Ex: instruction1. instruction2).

F-Script provides an array class. An array instance is an ordered collection of objects (i.e. a heterogeneous vector of objects). F-Script syntax provides an array literal notation using curly brackets. The general form is: {Item 1, Item 2, Item 3, ... Item n}

An array literal can contain any F-Script expressions. For instance, {6*5>10,1+1,'foo'} will be evaluated as {true,2,'foo'}. An empty array is denoted by {}. Array indexing starts at zero. An array is itself an object. Arrays of arrays are thus naturally supported.

4.2 Sending Messages

In object-oriented programming, sending a message (invoking a method) is the basic concept enabling object manipulation. Message expressions in F-Script are similar to those in Smalltalk. They describe which object is to receive the message, which operation is being selected, and any arguments needed to carry out the requested operation. The message expression components are the receiver, the selector, and the arguments respectively. Examples of message sending:

```
5 sin
```

The unary message "sin" is sent to a Number object with a value of 5, (the receiver). A Number object with value of -0.9589 is returned.

² Smalltalk is the traditional object-oriented language created by Alan Kay. It is one of the most powerful object-oriented language today.

2+4

The binary message "+" is sent to a Number object with a value of 2 with one argument, a Number object with a value of 4. A Number object with a value of 6 is returned.

```
35 between:0 and:100
```

The keyword message "between:and:" is sent to a Number object with a value of 35, with two Number objects as arguments whose values are 0 and 100. The "between:and:" method returns whether the receiver's value is greater or equal to the first argument and lesser or equal to the second. In this case, the boolean object "true" is returned.

As seen in the above examples, F-Script supports three primitive types of messages: unary, binary and keyword messages:

- Unary messages have no arguments, only a receiver and a selector.
- Binary messages have a single argument in addition to the receiver. Their selector is formed by a list of special characters. A binary message selector is a combination of the following characters: + - * / = > < ~ ? . % ! & | \. Sometimes a binary selector is called an operator³.
- Keyword messages contain one or more keywords, with each keyword having a single argument associated with it. The names of the keywords always end in a colon (:). The colon is part of the name of the message (the selector) – it is not a special terminator.

4.3 Priority and parentheses

The receiver of a message or any argument can also be a complex message expression. For example, the following message expression contains unary, binary and keyword messages:

```
4 sin between:2*3 and:100
```

F-Script's evaluation rules (which are the same in Smalltalk), are based on the type of message (unary, binary, or keyword) involved in the expression. In order of application, the evaluation order is as follows:

- 1) Parenthesized expressions
- 2) Unary expression (evaluated from left to right)
- 3) Binary expressions (evaluated from left to right, all binary selectors have the same priority level)
- 4) Keyword expression

Expression	Parenthesized Expression
2 sin negated	(2 sin) negated
3+4*6+3	((3 + 4)*6)+3
15 max:32/3	15 max:(32/3)
2 sin + 4	(2 sin) + 4
4 sin max:4*6	(4 sin) max:(4*6)
5 between:1 and:3 sin+4	5 between:1 and:((3 sin)+4)

³ This does not relate to the APL notion of operator, but to the more traditional term used in mathematics to define functions such as + or -.

5 New Message send paradigm

5.1 A few problems that need to be resolved

This section presents some of the problems that need to be resolved in order to integrate object technology and array programming. Our approach is based on the following remark: since sending messages is the fundamental operation in object technology, an array programming operation such adding two numerical vectors must be conveyed as the generation of a certain number of messages making use of the elements of the vectors used.

Where A and B are two vectors of the same size whose elements are objects of the Number class. If $A = \{1, 2, 3, 4\}$ and $B = \{10, 20, 30, 40\}$

Then $A+B$ must result in the generation of four messages: $1+10, 2+20, 3+30, 4+40$

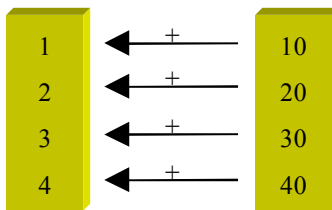


Figure 1. Four message sends are generated.

The result of this operation is a new object array containing the results of each message generated: $\{11, 22, 33, 44\}$.

The solution used by some array languages such as Fortran 90 and Perl Data Language [Glazebrook97] is to implement a certain number of operations (especially mathematical operators) in the language in such a way that they interact with all of the array elements used. We do not think this solution is satisfactory in an object context because it establishes a group of operations functioning in "array mode" and the whole idea behind objects is the creation of new data types and thus new operations (methods). It is thus impossible to establish in advance the operations that can be implemented. **Therefore, the proposed solution must be generic. In other words, it must work for any new method.**

Moreover, such a mechanism must work well for unary messages. For example, if we want to retrieve the class of each element of an array, we use the unary message "class" to which all of the objects know how to reply. Where C is an array $\{1, 2, 'a \text{ string}'\}$. In our example, we would like to be able to write something like "C class" and retrieve the following array: $\{\text{Number}, \text{Number}, \text{String}\}$. In this particular case, the interpreter must generate three send messages, i.e. sending a unary "class" message to each element of array C.

But a problem appears here. Indeed, array C is an object itself and it is thus capable of responding to the "class" message by returning the Array class. But when writing "C class," how do we know if it refers to finding out the class of array C or knowing the class of each element of C?

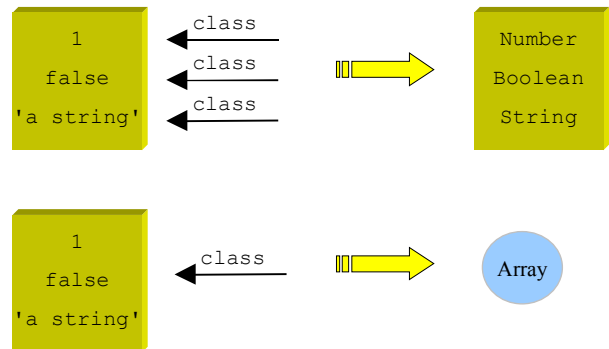


Figure 2.

Therefore, the language used must provide a means of differentiating the two possibilities (could something similar to APL's "each" function help out here?) Of course, this problem is not specific to unary messages.

This problem becomes more wide spread when we implements nested arrays (arrays of arrays). Indeed, the same operation can apply to different nested levels. You may wish to go to the maximum nesting level, stop at the first level or "go down" to an intermediate nested level.

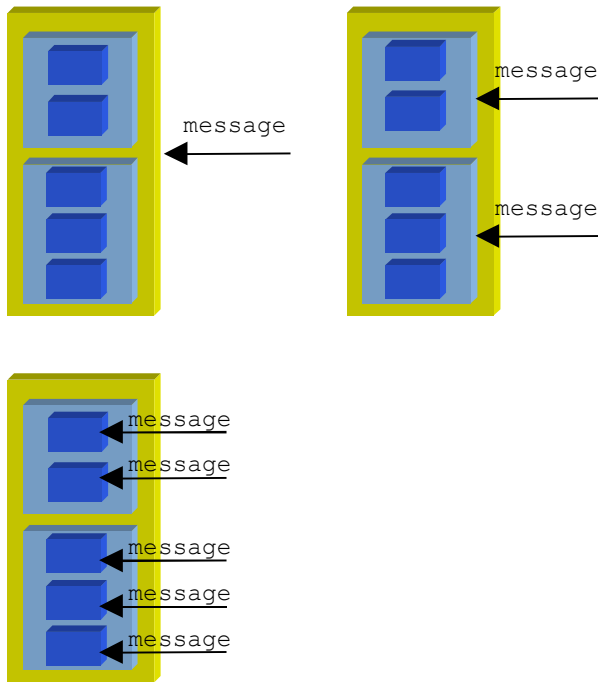


Figure 3. Nested arrays

Thus, the language must allow for specification of the nested level to be reached and this is true for each of the arrays concerned (receiver and arguments).

Below we have imagined the implementation of operations that associate array elements one by one. However, some other very useful combinations are possible. For example, we may want to run an outer product as in APL; in other words, combine an array element with each element of another array. For example, for the outer product:

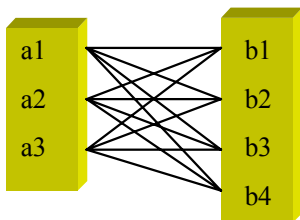


Figure 4. Outer product

The number of possible combinations becomes even larger if we consider messages with any given number of arguments. Indeed, these messages can bring any number of arrays and scalars into play.

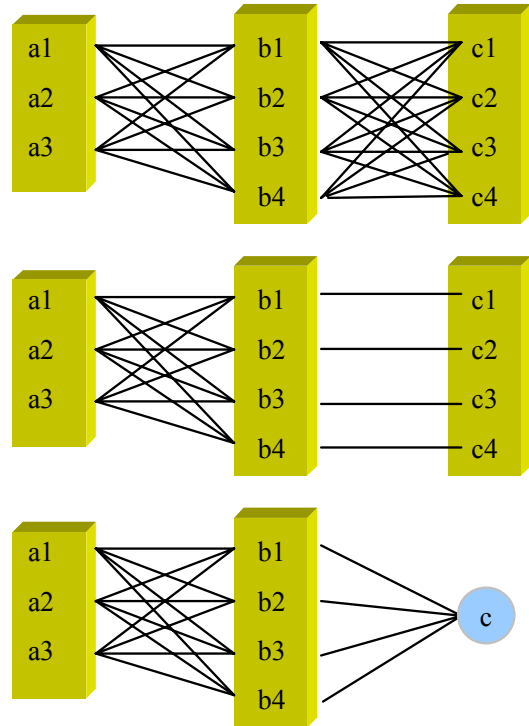


Figure 5. Array elements may be combined in complex ways.

Moreover, consideration of nested arrays makes the problem even more complicated. **The proposed language must cover a large number of useful combinations.**

5.2 Proposed solution: message patterns

An object-oriented program can be considered a specification that defines which messages are sent to which objects. In traditional object programming, the basic syntactic structure makes it possible to specify the sending of a simple message. We suggest adopting a new message sending paradigm that allows for the sending not of simple messages, but a complex group of messages. We call this new fundamental structure "multi-message pattern" or quite simply "message pattern." F-Script provides notation that allows for specification of message patterns in an operational manner. We also propose

abstract notation that makes it possible to express the structural properties of message patterns. These structural properties are called patterns. For example, the notion of an external product corresponds to a particular pattern. The notation proposed by F-Script, as well as the abstract notation, enables the expression of a wide range of patterns and has some very interesting practical characteristics (for example, the classic send message notation is still supported: it becomes naturally a particular case of message pattern). Of course, it is possible to imagine implementation of other notations that could express different patterns, for example.

The heavy use of explicit loop control structures (while, loop...until, etc) that is common in traditional languages is replaced with message patterns in F-Script.

5.3 Message patterns with F-Script

In this section, we present the principal elements of F-Script notation for message patterns. A more complete description can be found in the F-Script User's Manual [Mougin99b]. F-Script notation makes it possible to:

- Specify for each array involved in a message pattern whether a loop must be generated for browsing 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. A message pattern is thus a recursive structure.
- Implement the notion of "implicit message pattern," which makes notation easier.

5.3.1 Implicit Message Pattern

The array class, which defines array behavior, can respond to a selection of well-defined messages (for example, messages allowing for subscripting, insertion or deletion of elements, size consultation and so on). When a message that is not included in this group is sent to an array, it is considered that this message applies to each array element. Moreover, if certain arguments of this message are also arrays F-Script takes the elements in these arrays one at a time. When A, B and C are arrays, this mode makes it possible to write expressions such as:

```
A+B
A&B
A*2
A cos
A between:B and:C
A between:10 and:C
Etc.
```

These expressions have the expected array semantics.

5.3.2 Explicit Message Pattern

When the "@"⁴ symbol is placed after the message recipient and/or just before the arguments, it means that a loop on designated arrays must be run in order to generate messages sends. For example:

```
{1,2,3} @ class returns the array
{Number,Number,Number}.
```

```
2 max:@{0,1,10,20} returns the array
{2,2,10,20}
```

All implicit message patterns such as the one implemented by the notation "A+B" when A and B

⁴ Bob Brown uses the "@" notation in his Object-Oriented APL project [MacDonald] in order to specify a message send.

are arrays can of course always be specified in an explicit manner. In this case, `A @+@ B`.

Up until now, we have seen that by using patterns on some arrays we can generate messages that use the first element of each array, then the second and so on. But we can also combine an array element in other ways. For example, suppose we want to do the outer product of `A` and `B`, using the `*` message. To do this, we have to specify that we want a loop on `A` and an inner loop on `B`. To do this, we use a number after the `@` symbol to state the inner level of each loop: `A @1*@2 B`

Other example:

```
A @1 between:@2 B and:@3 C
```

Some loops can be found at the same level:

```
A @1 between:@2 B and:@1 C
```

Notation becomes more widespread in nested tables thanks to the possibility of denoting recursive message patterns and stringing them together. This is a very important notion. For example:

```
{{1,2,'aString'},{5}} class returns
Array
```

```
{{1,2,'aString'},{5}} @ class returns
{Array, Array}
```

```
{{1,2,'aString'},{5}} @@ class returns
{{Number,Number,String},{Number}}
```

In the last case we have a recursive message pattern. The first level of the message pattern is “@”. It indicates that the rest of the message pattern must be applied to each element of the array receiving the message. Here, the rest of the message pattern is composed of the “@ class”.

Another, more complex example of a recursive message pattern, using `++`, the concatenation message:

```
{{'a11','a12','a13'},{'a21','a22','a23'}}
@1@2 ++ @2@1
{{'b11','b12'},{'b21','b22'}}
```

returns

```
{{{'a11b11','a12b11','a13b11'},{'a11b12',
'a12b12','a13b12'}},{'a11b21','a12b21',
'a13b21'},{'a11b22','a12b22','a13b22'}},
{{{'a21b11','a22b11','a23b11'},{'a21b12',
'a22b12','a23b12'}},{'a21b21',
```

```
'a22b21','a23b21'},{'a21b22','a22b22',
'a23b22'}}}}
```

This pattern has the form `ij:kl` where `i` is `@1`, `j` is `@2`, `k` is `@2` and `l` is `@1`. To execute this pattern, F-Script first looks at the first level of the pattern, so it considers `i:k`, which is `@1:@2`. It then applies this first level pattern: it generates the “messages” between the combinations (described by this pattern) of the elements of the arrays in the message expression. These combinations, described by the semantic of the `@1:@2` pattern, are:

```
{'a11','a12','a13'}<-message->{'b11','b12'} (1)
{'a11','a12','a13'}<-message->{'b21','b22'} (2)
{'a21','a22','a23'}<-message->{'b11','b12'} (3)
{'a21','a22','a23'}<-message->{'b21','b22'} (4)
```

But what exactly is “<-message->”? <-message-> is in fact the original message pattern expression (more precisely, the part of the original message pattern expression composed by the pattern specification and the selector) minus the first level pattern.

So <-message-> is `@2++@1` (i.e. `j:l`).

So (1) is: `{'a11','a12','a13'} @2++@1 {'b11','b12'}`

(2) is: `{'a11','a12','a13'} @2++@1 {'b21','b22'}`

(3) is: `{'a21','a22','a23'} @2++@1 {'b11','b12'}`

(4) is: `{'a21','a22','a23'} @2++@1 {'b21','b22'}`

F-Script now executes (1), (2), (3) and (4) and returns an array with the four results of these four executions. The final result is: `{{result of the execution of (1), result of the execution of (2)}, {result of the execution of (3), result of the execution of (4)}}`.

5.3.3 Abstract Notation

We can extract the structural information of a particular message pattern expression and represent it:

- The pattern of the message pattern expression `{1,2,3} @+@ {10,20,30}` is `@:@`
- The pattern of the message pattern expression `2 max:@ {0,1,10,20}` is `:@`
- The pattern of the message pattern expression `{{1,2,3,4},{'oliver','henry'},{10,100}} @ count is @`

Some patterns are frequently used and have even been given standard, well-known names. For example, `@1:@2` is called "outer product." Of course, the pattern notion also apply to implicit message pattern expression:

- In `{1,2,3}+{10,20,30}` we have the pattern `@:@`
- In `{1,2,3} between:3 and:10` we have the pattern `@::`

The pattern of a traditional simple message send (one that does not involve generation of a loop) is denoted by the symbol " μ ". For instance, the pattern of the message pattern expression "1+2" is μ . This is the simplest message pattern. When executed, it yields just one message send. In classical OO languages such as Smalltalk or Java, the pattern of all message expressions is μ .

6 Operations for Array Programming

The notion of message pattern alone is not sufficient. This notion becomes powerful when it is associated with specific array programming operations. Here, we present a group of operations that are implemented by various base classes of the F-Script system. When associated with the use of message patterns, these operations make it possible to capitalize on the power of array programming principles in an object context. It is worth noting that all of these operations are implemented as simple methods on objects. No new syntactic notation is necessary.

6.1 Note about operators and blocks

Traditional array programming languages use the notion of operator extensively. The definition of

the term «operator» as used in APL differs greatly from its sense in classical languages. In APL, an operator is an operation (usually on a function) which produces a function (in terms of J it is adverb or conjunction). It is even possible to define new operators in APL. F-Script provides a powerful construct that attains the same goals, but in a different way. This construct is called a "block". Blocks represent one of the most powerful notions of F-Script (and Smalltalk). In particular, blocks cover the functionality of defined functions and defined operators in APL language. Basically, a block is an object that contains some F-Script code. You usually create a block by bracketing a segment of code. The F-Script interpreter identifies this as a literal for a block object and in consequence returns such an object at evaluation time. For example: `[x:=x+1]` is a block. You can execute the code inside this block by sending it the "value" message.

Blocks may have parameters. For example `[:a :b| a+b]` is a block with two parameters named `a` and `b`. When executing this block you have to supply the value for the arguments using a valid "value..." message (for example: `[:a :b| a+b] value:3 value:4` will return 7). Blocks may also have local variables and bindings to other objects. They can be executed recursively. This is because a block is a closure (code + environment).

A block that just sends a single message to its first argument with its other arguments as arguments to the message may be represented using the compact notation for block literal. This notation represents a block by a "#" immediately followed by the message selector that the block must use when it is executed. For example, the block `[:a :b| a + b]` may also be represented in a compact form with `"#+"`.

Because blocks are objects, they can be used as arguments to methods, they can be saved and loaded, inserted into collection etc. Blocks also support F-Script's control structures. In F-Script there are no special instructions for control structures like `while...` or `if...else`. These control structures are realized by using the general messaging mechanism and blocks. For example, in the `FSBoolean` class, the method `ifTrue:ifFalse:` takes two blocks as arguments: one block to execute if the boolean is true and another to execute if the boolean is false.

This technique also allows for user-defined control structure.

Since blocks can be used as arguments, they allow for design methods similar to APL operators. For example, the reduction operator is implemented in F-Script as a method of the array class that takes one argument: a block describing the function to apply to the array.

6.2 Fundamental operations

6.2.1 Basic Array Operations

Here we are talking about traditional array operations: inserting an element, deleting an element, simple subscripting, consulting array size (“count” method), etc. Array indexing is carried out via the “at:” method. Indexing start at zero. For example `{10,11,12} at:2` returns 12. Combined with a message pattern, this message enables, for example, retrieval of the second column of a 3*3 matrix, whose lines are given in an array:

```
{{1,2,3},{4,5,6},{7,8,9}} @at:1  
returns {2,5,8}
```

6.2.2 The “distinct” Method

This method, which is implemented by the array class, returns the receiver elements but eliminates any duplicates. For example:

```
{1,2,1,1,3,2} distinct returns {1,2,3}
```

6.2.3 Reduction

Reduction is implemented as the “\” method of the array class. Its argument is made up of a two-argument block. For example, the sum of the elements of an array A can be found with: `A \ #+`

6.2.4 Extended indexing

Array indexing can be done using an array index; in other words, an array of numbers. It can also be done using an array of booleans. Array indexing is carried out via the “at:” method.

An example of indexing by an array of number:
`A at:{1,2,3,15}`

Indexing by an array of booleans carries out the operation called compression. This operation comes into play when doing a selection. For instance, selecting the elements of an array A that are greater than 60: `A at:A>60`. This example combines use of compression and implicit message patterns. Another example: Suppose we manage a car dealership. We have an array named “cars” that contains objects of class Car. Each one represents a particular car we have sold. The Car class has a method “owner”, which returns an object of class Person. This Person class has a method “mail:” with which you can send an e-mail to a person. The class Car has also a method “type” that returns the type of the car, and a method “groupId” that returns a group identification for the car. The FSBoolean class has the “&” method for logical AND. Suppose you want to send a message to all the owners of a car of type “BMW A” of the group 544. You type:

```
(cars at:cars groupId=544 & (cars type  
= 'BMW A')) owner distinct mail:'Dear  
customer blah blah blah'
```

This contrasts with classical OOP in which we don't benefit from array technology. For instance, here is an Objective-C version (using a powerful collection framework) of the same query:

```
int i, nb;  
NSMutableSet *selectedOwners;  
NSEnumerator *enumerator;  
Person *customer;  
  
SelectedOwners = [NSMutableSet set];  
  
for (i=0, nb=[cars count]; i < nb; i++)  
{  
    Car *currentCar=[cars  
objectAtIndex:i];  
    if ([currentCar groupId] == 544 &&  
[[currentCar type] isEqual:@"BMW A"] &&  
[currentCar owner])
```

```

    [selectedOwners
addObject:[currentCar owner]];
    // To only have distinct owners
}
enumerator=[mySet objectEnumerator];
while(customer=[enumerator nextObject])
{
    [customer mail:@"Dear customer blah
blah blah"];
}

```

6.2.5 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. This method involves comparisons between the receiver’s elements and those of the argument⁵. Example: `{1,2,'foo'} >< {4,'foo',1,'foo','foo'}` returns `{{2},{}, {1,3,4}}`

Let’s continue with our example of a car dealership used in the previous chapter. Imagine that there is an array named “*p*” that contains objects of the Person class. We would like to retrieve a list of the cars bought in the store by each person. If our Person class has a method for sending this list back (say the “`getCarList`” method), we simply have to do “`p getCarList`” in order to obtain the results we want—an array that is the same size as *p* and that contains a list of cars for each person. However, if we do not have a method such as “`getCarList`” enabling us to navigate from people directly toward the cars, we must use a join. This is the case given that we have the “`owner`” method described in the previous chapter that makes it possible to navigate from a car to its owner. The result can then be obtained by combining the join method, an extended indexing and two message patterns: `cars at:@ p >< cars owner`

Here is an example of a F-Script expression that mixes compression, joins, implicit and explicit message patterns and reduction. Imagine we want to know if all of the people between 40 and 50 years old own at least two cars. We will suppose that an

⁵ F-Script implements the join method in an optimized manner by using hash tables on the objects involved.

“`age`” method on the Person class exists and sends back a number. We would write:

```

(cars at:(p at:(p age between:40
and:50)) >< cars owner) @count >= 2 \
#&

```

If we details the execution:

- 1) `p age` is executed first. *p* is an array and since the array class does not implement the “`age`” method, the F-Script interpreter recognize the situation as the use of an implicit message pattern. Thus, it send the message “`age`” to each element of *p*, returning an array of number. We note this partial result *r1*.
- 2) `r1 between:40 and:50` is then executed. This time, we are again using an implicit message pattern. The message “`between:40 and:50`” is sent to each element of *r1*. The result is an array of booleans that tell for each person if she is between the age of 40 and 50. We note this partial result *r2*.
- 3) `p at:r2` is then executed. It’s simply the compression of *p* by *r2*. It returns in an array only the people that are between 40 and 50 years old. This result is *r3*.
- 4) `cars owner` is executed. “`cars`” is an array and the array class does not implement the “`owner`” method: this detected by F-Script as an implicit message pattern. The “`owner`” message is sent to each car object. The result (noted *r4*) is an array of Person objects giving, for each car, its owner.
- 5) `r3 >< r4` is now computed. It returns an array of array that gives for each person in *r3* the indices in “`cars`” of the cars she own. This result is noted *r5*.
- 6) `cars at:@ r5` is then computed. It’ an explicit message pattern that specify to generates messages send by iterating on the element of *r5*. The result (*r6*) is an array of array that gives, for each corresponding person in *r3*, the list of cars she owns.
- 7) `r6 @ count` is computed. It is an explicit message pattern that specifies to send the “`count`” message to each element of *r6*. The result (*r7*) is an array that gives, for each person in *r3*, the number of car she owns.
- 8) `r7 >= 2` is computed. An implicit message pattern is recognized: the “`>= 2`” message is sent to each element of *r7*. The result (*r8*) is an array of

booleans that tells, for each person in r3, if she own at least two cars.

9) `r8 \ #&` is then computed. This is a reduction by the logical AND message. The result is a boolean telling if all of the people between 40 and 50 years old own at least two cars.

6.3 Other array programming operations

F-Script includes many array-programming operations in addition to those presented in the previous sections. Below we will present some of these operations as an example.

6.3.1 The iota Method

Provided by the Number class, this method returns an array with all the integers between 0 and the receiver-1. For instance: `10 iota` returns `{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}`

6.3.2 Transposition

Transposition is provided by the array class with the “`transposedBy:`” method. This method computes and returns an hypercube that is a transposition of the receiver according to the transposition vector passed as argument. Example:

```
{ {11,12,13}, {21,22,23} } transposedBy: {1,0}
returns { {11,21}, {12, 22}, {13, 23} }
```

6.3.3 Sorting

Sorting is provided by the method “`sort`” of the array class. This method returns, in an array, the indices that will arrange the receiver in ascending order. In order for this to work, the elements of the array must respond to the message “`<`” so as to provide a total order. For example: `{12,15,13,11,14} sort` returns `{3,0,2,4,1}`

6.3.4 Set methods

Arrays can be used as if they were sets, thanks to the methods `union:`, `intersection:` and `minus:`. For example:

```
{1,2,true,'foo'} intersection:{3.14, 1, 'foo'}
returns {1, 'foo'}
```

6.3.5 Scan

Provided by the “`\`” method of the Array class, this operation can be seen as doing a reduction and returning an array with all the intermediates results computed during the reduction.

```
{1,2,3,4,5} \ \ #+ returns {1,3,6,10,15}
```

6.3.6 Replication

Provided by the Array class, the “`replicate:`” method requires its argument to be an array (with the same size as the receiver) of positive integers. Each element of this argument represents the number of references to the corresponding element of the receiver that have to be put into the returned array.

```
{10,11,12,13,14} replicate:{0,1,2,0,1}
returns {11,12,12,14}
```

6.3.7 Rotation

Rotation is provided by the Array class with the “`rotatedBy:`” method, which returns a circular permutation of the receiver. The permutation is made to the left or to the right, according to the sign of the argument.

```
{1,2,3,4,5,6} rotatedBy:2 returns
{3,4,5,6,1,2}
```

6.3.8 Concatenation

The “`++`” method of the Array class returns an array that is the concatenation of the receiver and operand2. The receiver and operand2 are left unchanged. Example: `{1,2,3} ++ {4,5}` returns `{1,2,3,4,5}`

6.3.9 Searching

The “`!`” method provided by the Array class resembles the dyadic `iota` function in APL. It returns the index of the argument in the receiver. If the argument isn't found in the receiver, the first illegal index of the receiver is returned. Example: `{10,11,true,13,'foo'} ! 11` returns `1`.

6.3.10 Subparts

The method "subpartsOfSize:" invoked on an array returns, in an array, all the receiver's subparts of the size given as argument. Example:

```
{1,2,3,4} subpartsOfSize:0 returns {}
```

```
{1,2,3,4} subpartsOfSize:1 returns  
{{1},{2},{3},{4}}
```

```
{1,2,3,4} subpartsOfSize:2 returns  
{{1,2},{2,3},{3,4}}
```

6.3.11 Reversing

The "reverse" method invoked on an array returns an array with the same elements as those of the receiver arranged in reverse order. Example:

```
{1,2,3,4} reverse returns {4,3,2,1}
```

6.3.12 Prefixes

The "prefixes" method returns, in an array of array, all the prefixes of the receiver. Each sub-array of the result is a prefix.

```
{1,2,3,4} prefixes returns  
{{1},{1,2},{1,2,3},{1,2,3,4}}
```

6.3.13 Index

This method invoked on an array, returns in an array all of the valid receiver indexes in ascending order. This is equivalent to applying successively the count and iota methods.

```
{10,11,12,"foo"} index returns {0,1,2,3}
```

7 Some Interesting Results

The first experiments with F-Script have shown that integration of Array technologies with object-oriented programming adds a considerable amount of power to the latter. In particular, it makes it possible to propose a high-level, object query language. This point is illustrated in [Mougin99b]: an object model using several related classes is defined and an explanation is given of how to express numerous complex queries.

There is a strong synergy between object technology and Array programming. In traditional OO programming, we use one object at a time, we think about individual objects, and when we have to apply the same process to multiple objects, we often have to write loops. But it's interesting to note that applying the same process to multiple objects is a key point of OOP, because polymorphism is only useful when we have to apply the same process to multiple objects. The synergy can be explained as follows: on one hand, Array programming makes it possible to write code with few explicit loops and on the other hand, object technology radically decreases the use of explicit conditional structures, as they are replaced by an automatic switchpoint during dynamic binding. It just so happens that using explicit conditional structures usually hinders the use of Array programming. It is for this reason that OOP, which favors a style of programming without these structures, naturally allows for extended use of Array programming.

Another example of synergy can be seen with regard to the implementation of certain optimizations. Indeed, processes no longer focus on isolated objects but are expressed in a synthetic manner in object Arrays. This allows for numerous optimization opportunities. For example, F-Script benefits from considerable optimizations that get it over its handicap of being a "pure object" language, in which even the slightest number or boolean is an object manipulated by message send, which is very costly. Indeed, the Arrays proposed by F-Script are capable of modifying automatically their internal representation to better adapt to the data they contain. This is why an Array containing only objects representing numbers will take on an optimized memory representation. Operations based on Arrays are programmed according to this representation and provide performances that correspond to those of a process written in C on an Array of numerical values. Message send is bypassed and the base C operators focused on numbers are directly applied. The boolean Arrays, which are frequently used, are also optimized. The most traditional message patterns are recognized by the interpreter and yield an optimized implementation instead of a generic one. Operators such as reduction take advantage of the information in optimized Arrays. For example, reduction of a boolean Array by a logical OR will stop as soon as a "true" value is found in the array. The way in

which Arrays manage their internal representation is transparent to the user, who from his point of view, manipulates only Arrays of objects. This internal management uses heuristics that make it very fast.

8 Related work

Several teams have explored integration of object technology and Array programming. In most cases, their goal has been to study how object notions could enhance an Array language. In particular, we find this type of integration in the latest versions of J, as well as in the K language. Bob Brown also presented his work on integrating object aspects and APL. [MacDonald]. [Giradot87] includes an in-depth discussion of this subject. Martin Gfeller talks about the use of object programming with the APL AIDA [Gfeller89]. In some cases, an object model is developed above an Array language to respond to a specific application context. An example of this type of development, as well as a discussion about this subject and references can be found in [Bottoni94].

F-Script takes a rather different approach as it consists in bringing the notions of Array programming to the object world. Its goal is therefore to find the minimum number 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 done in this area. It is worth noting the remarkable work of Marcel Weiher [Weiher99], who proposes a model, based on the notion of dynamic wrapper (or filter), which enables high-order messaging. This model can be attached to an object language without needing to modify its syntax or semantics (nonetheless, the language must be very dynamic and reflective). This model was implanted for Smalltalk and Objective-C.

9 Conclusion

Integration of array programming with object programming can be done by extending the notion of message send which implies a slight syntactic extension, and the addition of certain array manipulation methods, which does not require modification of the targeted language's syntax.

The manner in which F-Script handles encapsulation, inheritance and polymorphism has not been addressed in this paper. Nonetheless, it can be noted that in our 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 the manipulation of standard Objective-C objects with the power of array programming.

The integration between array technology and object-oriented programming presented in this paper provides a high-level notation, which makes it possible to easily express complex object manipulations. We think that it represents a strong offer for the evolution of object languages.

References

- [AOPW99] *Proceedings of the Aspect-Oriented Workshop at ECOOP'99, 1999.*
- [Albert88] EUGENE ALBERT, KATHLEEN KNOBE, JOAN D. LUKAS, GUY L. STEELE, *Compiling Fortran 8x Array Features for the Connection Machine, Computer System*, 1988, pp 42-56
- [Berry69] PAUL BERRY, *APL 360 Primer*, IBM Corp.
- [Bottoni94] P.BOTTONI, M.MARIOTTO, P.MUSSIO, *LISEB: a Language for Modeling Living Systems with APL2*, Proceedings of the international conference on APL: the language and its applications , 1994, pp.7-16
- [Codd70] E F CODD, *A relational model of data for large shared data banks*, Comm ACM 13, 6 June 70, pp.377-387

- [Codd72] E F CODD, *Relational completeness of data base sublanguages*, in Data Base Systems, Rustin, Ed, Prentice Hall (1972)
- [Fox94] GEOFFREY FOX, NANCY MCCRACKEN, *Array Programming in Fortran 90*, <http://www.npac.syr.edu/EDUCATION/PUB/hpfe/module1/index.html>
- [Geyer89] A.GEYER-SCHULZ, J.M.HOHENEN, A.TANDES, *An APL tutoring adventure game*, APL89, APL Quote Quad, vol.19, n.4, p.148-157
- [Gfeller89] M. GFELLER, *Object-Oriented Programming in AIDA APL*, APL89, APL Quote Quad, vol.19, n.4 pp. 164-168
- [Girardot87] JEAN JACQUES GIRARDOT AND SEGA SAKO, *An Object Oriented extension to APL*, APL 87, APL Quote Quad, vol.17, n.4, pp.128-137
- [Glasgow89] GLASGOW, JENKINS, MCCROSKY, MEIJER, *Expressing Parallel Algorithms in Nial*, Parallel Computing Journal, 11 3 pp 331-347, 1989.
- [Grice] DON GRICE, *Deep Blue designed with the aid of APL*, <http://www.software.ibm.com/ad/apl/apl2-deep.html> ,IBM Corp.
- [Glazebrook97] KARL GLAZEBROOK, FROSSIE ECONOMOU, *PDL: The Perl Data Language – High-speed number crunching in Perl*, Dr. Dobb's Special Report, Fall 1997
- [Holmes] DENIS HOLMES, JOHN E. HOWLAND, *APROL: a Hybrid Language*, Proceeding of the international conference on APL, 1993, pp 112-123.
- [Iverson] K. IVERSON, *Computers and Mathematical Notation*, <http://www.jsoftware.com/pubs/cam.html>
K. IVERSON, *MATH for the LAY-MAN*, <http://www.jsoftware.com/pubs/mftl/mftl.htm>
- [JenKins89] JENKINS, GLASGOW, *A Logical Basis For Nested Array Data Structures*, Computer Languages Journal, 141 pp 35-51, 1989.
- [Larkin] DON LARKIN, GREG WILSON, *Object-Oriented Programming and the Objective-C Language*, Developer's Library, Apple. <http://developer.apple.com/techpubs/macosx/ObjectiveC/ObjC.pdf>
- [MacDonald] RANDY MACDONALD, "*Bob Brown and Object Oriented APL, April 1996*", <http://www.torontoapl.org/ga/ga9605/bbrown.txt>
- [Mougin 98] PHILIPPE MOUGIN, *Base de données objet et modèle relationnel: intégration par la théorie des hyper-tuples virtuels*, SQLI on-line, 1998. Available upon request (pmougin@acm.org)
- [Mougin99b] PHILIPPE MOUGIN, *F-Script Guide*, 1999. Can be downloaded from the F-Script web site at <http://www.fscript.org>
- [NialSystem] NIALSYSTEM CORP, *About the Nial Language*, <http://www.nial.com/AboutNial/AboutNial.html>
- [Weiher99] MARCEL WEIHER, *MPW Foundation*. Can be downloaded as part of the Objective-XML package from <http://www.metaobject.com/Community.html>
- [Willhoft91] ROBERT G. WILLHOFT, *Comparison of the functional Power of APL2 and FORTRAN90*, Proceedings of the international conference on APL '91, pp.343-357
- [Whitney93] ARTHUR WHITNEY, <http://www.kx.com/technical/source/k.rtf>