# Implementing Distributed Generic Functions

Christian Queinnec*
LIP6 & INRIA-Rocquencourt

*ABSTRACT. The network now gives the opportunity to combine code and data from everywhere in the world. However, the dominant paradigm is the client/server model where immobile objects with static interfaces can only be used as prescribed by their proprietary site. While this constraint corresponds to understandable industrial programming practices, it negates the point of view of dynamic clients that collect interesting objects and want to confer new behaviors to these collected objects. How to enrich objects "from the outside" that is, without access to their source code hence without re-compilation of their defining classes, is the problem addressed by this paper.*

*Generic functions,* à la *CLOS [BDG+88], separate classes from behaviors i.e., methods. Roughly said, a generic function is a bag of methods; when a generic function is invoked, the nature (type, class or, structure) of its argument(s) triggers the choice of an appropriate method. Methods are no longer the exclusive property of classes, they are regular functions anyone may define outside classes definitions.*

*This paper describes how generic functions may be conveniently and not so inefficiently implemented in a distributed world. Section 1 presents some of the constraints of distributed systems. Section 2 recalls the framework of generic functions as well as how we extend them to the distributed world. Significantly, we address the problem of mutual recursion over a bag of methods to which new methods may be adjoined at run-time. We also propose a new feature:* `call-former-method`*. Section 3 discusses implementation while Section 4 eventually discusses the incorporation of these results in a real system.*

## 1 Distributed World

The distributed world is composed of many independent and separate address spaces i.e., *sites*. These sites communicate only via messages: sites share nothing. However, messages may be used to support (more or less expensively) higher-level abstractions such as *distributed shared memory* (DSM) associated to some coherency protocol: DMEROON [Que95] is one of those systems.

DMEROON is a library of C functions that provide, for various languages and implementations (C, various brands of Scheme), a data model above a

*Université Paris 6 — Pierre et Marie Curie, LIP6, 4 place Jussieu, 75252 Paris Cedex, France – Email: `Christian.Queinnec@lip6.fr` This work has been partially funded by GDR-PRC de Programmation du CNRS.

coherently distributed shared memory. DMeroon allows simultaneous users to statically or dynamically create new classes hierarchically organized, to dynamically instantiate these classes and, to dynamically and coherently share the resulting instances over a network. DMeroon automatically takes care of representation and alignment, migrating and sharing objects, local and global garbage collections. Experimentation with DMeroon led us to the following observations:

1. Sites only have a partial vision of the hierarchy of classes: global knowledge is out of reach.
2. Mutable objects require coherency to be useful, coherency is expensive to maintain: the less mutable objects are, the better.
3. Cycles of objects are difficult to identify and reclaim: cycles should be avoided.
4. Remote pointers i.e., references onto objects from remote sites, are heavy data structures: coalesce objects.

These observations are important since they make the implementation of a multi-user distributed class system rather different from the implementation of a single-user, single-site class system where the speed of method-lookup and class-membership are the strongest constraints [VHK97]. In the DMeroon class system, the representation of an instance begins with a pointer to the representation of its class: classes are objects according to the ObjVLisp model [BC87]. A class naturally has an immutable 'super' field referencing its super-class, but a class does not know its sub-classes since *(i)* this would require a mutable field hence would imply some coherency protocol to cope with simultaneous creation of sub-classes (imagine the consequences on the universally shared `<Object>` class!), *(ii)* this would create cycles between super-classes and their sub-classes making classes more difficult to reclaim. To cut the link towards subclasses prevents some optimizations concerning method lookup but allows anyone to freely and quickly create sub-classes of any known class.

Classes are anonymous i.e., cannot be retrieved by their name, since this would imply a global shared (hence mutable) name-space for classes. Reciprocally, a class has an immutable 'name' field holding a label to identify it (for instance, for debug purpose). It is however possible for a user to maintain a hash-table mapping names to the classes s/he knows; but it is not possible to number classes since no one is able to know them all.

Multiple class systems coexist, among them are Corba [Sie95] and DMeroon. Our solution is independent of any languages such as Scheme, Java, or C; it only uses the primitives of table 1 which are commonly present in every object system provided by or on top of these languages.

| Name | Functionality |
|---|---|
| is-object? | check whether a value is an instance of `<Object>` or of one of its subclasses |
| subclass? | check whether a class is a subclass of another class |
| super-class | return the super-class of a class |
| class-of | return the direct class of an instance |

Table 1: Assumed primitives for a class system. The `<Object>` class represents the root of the hierarchy of classes; the `is-object?` predicate distinguishes direct or indirect instances of `<Object>` from other values if not all values are instances.

## 2 Generic Functions

This section presents generic functions. We then discuss our particular design to accommodate the distributed world.

A generic function is a bag of methods accompanied by a *dispatcher*, a function which selects the most appropriate method for a given set of (nature of) discriminating arguments. A generic function is defined by a *signature* (a kind of Java interface reduced to a single method) that explicits which variables participate to the choice of the appropriate method: these are the *discriminating variables*. According to [KR90], more than 95% of generic functions have a single discriminating variable which is then the equivalent of the *receiver* in traditional OO languages. Generic functions with at least two discriminating variables provide *multi-methods*: a feature arguably not belonging to OO mainstream, not easily nor efficiently simulated by multiple dispatch but far more simpler to master than the conjunction of single-dispatch and overloading as in Java [GJS96, pp. 323–340].

Our generic functions are anonymous. Generic functions may be bound to variables through which they may be named, manipulated and, chiefly, invoked. Methods may later be added to generic functions to form new, richer generic functions. The signature of methods must be congruent with the signature of the generic function to which they are added i.e., they should have same arity and similar discriminating variables. Methods should also respect the intention of generic functions, for instance, a printing generic function that collects the various methods that exist to print a variety of objects should probably not be adjoined unrelated methods: generic functions gather various behaviors under a common intention leaving the exact choice to the hidden dispatcher.

Our examples and explanations will be based on the Scheme language [CR91]. To simplify the programs of this paper, we suppose generic functions to have only one discriminating variable (the receiver in object parlance) as first argument. Because generic functions are represented by non-regular Scheme values such as DMeroon objects, we will suppose `send` to be the imposed function through which generic functions are invoked. With the above simplification, the `send` function has the following syntax:

(`send` *generic receiver arguments...*)

The syntax of the `generic` special form is as follows:

(`generic` (*name* (*discriminating-variable*) *other-variables...*)
*expressions-to-compute-the-default-value-of-the-generic-function* )

Here is an example of the creation of a generic function evaluating abstract syntactic trees (AST):

```
(define evaluate (generic (eval (ast) r k)
                   (k ast) ))
```

The `generic` special form returns a generic function. This generic function takes three arguments: an AST, an environment `r` and a continuation `k`. Only the AST is discriminant and participates to the choice of the method. The `eval` symbol that appears in the above example is a label that identifies the generic function for debug purposes; it will also serve for recursion as explained in Section 2.3. The body of the generic function, `(k ast)`, determines the behavior of the generic function when no specific method can be found (it simply returns the discriminated value `ast` to the continuation `k` ignoring the lexical environment `r`). This default method is particularly useful for languages where not every value is an instance of the distributed class system. A method for `<Object>` provides a default method for all objects; the default body of the generic function provides a method for values that are not objects. The embedding `define` form binds the freshly created generic function to the `evaluate` global variable.

### 2.1 Mutability of Generic Functions

According to the observations of Section 1, the main question about generic functions is: are they mutable or not?

If they were mutable then all the users that share a generic function (the printing generic function, for instance) may enrich it with new methods thus forcing some coherency protocol to run and update the view these users have of this generic function. To avoid this tremendous cost, we adopt a more purely functional point of view (already suggested in [Que93]): generic functions are immutable, therefore, adding a method to a generic function yields a new, different, enriched generic function.

In order to share a generic function with all the methods that may potentially be adjoined later on, one may update the variable that held the former generic function with the newly created generic function: we then swapped immutability of generic functions (that is, values) for mutability of shared variables (confined by lexical scope rules). It is far easier, for a compiler, to take care of and benefit from mutable variables (since mutation sites are statically visible) than to run a mutation analysis. Therefore, if $g$ is a variable holding a generic function to which a method $m$ is added (with the `add-methods` function), sharing this new generic function may be ensured with:

```
(set! g (add-methods g m))
```

Scheme programmers will observe the absence of a trailing "`!`" in the name of the `add-methods` function: the `add-methods` function has no side effect. Users of the former value of $g$ will not be impacted by the creation of a new generic function based on $g$. These users may use the new generic function via the new value of the variable $g$.

### 2.2 Definition of Methods

The `method` special form takes care of the definition of new methods or, more precisely, *installable methods*: we reserve the word "method" for the behaviors that are gathered in a generic function. An installable method is a value that may be given to `add-methods`. An installable method specifies a behavior (incarnated by a regular function) as well as the classes that trigger the election of that behavior.

The syntax of the `method` special form is similar to that of the `generic` special form. It specifies the class the discriminating variables must have for the method to be triggered. The body of the method appears right after.

Continuing our `evaluate` example, here follows the definition of an installable method for constants (of class `<Constant>`, a kind of AST node):

```
(method (consteval (c <Constant>) env cont)
  (cont c) )
```

Observe that the `consteval` label and the other variables `c`, `env` and `cont` may have names different from the names used by the other methods. Also note that this installable method and the default behavior of the generic function to which it will be added are exactly the same. The generic function supports only methods with congruent signature. This installable method may enrich the previous generic function as in:

```
(set! evaluate
      (add-methods evaluate
                   (method (consteval (c <Constant>) env cont)
                     (cont c) ) ) )
```

Differently from Dylan [App92], the value returned by the `method` special form cannot be invoked: this is not a function but an installable method. An installable method may enrich more than one generic function, this allows, for instance, to build concurrently two generic functions with not exactly the same bag of methods. The `add-methods` function may take more than one method to add: this allows to factorize part of the installation cost namely, the re-computation of the dispatcher.

### 2.3 Recursion between Methods

The immutability of generic functions brings a new problem which is: how to achieve recursion within a generic function i.e., within a bag of possibly mutually recursive methods? This problem is not exactly similar to the problem of defining local anonymous recursive functions (letrec in Scheme [Que96]) since, in a letrec form, the whole set of mutually recursive functions is statically known whereas, in a generic function, existing methods must also be mutually recursive with the methods that will be added later. Recursion is offered via the label provided by the definition of the method. Consider the following method definition for <Alternative>, another kind of AST node:

```
(method (eval (e <Alternative>) r k)
  (send eval (Alternative-condition e)
             r
             (lambda (bool)
               (send eval (if bool (Alternative-consequence e)
                                   (Alternative-alternant e) )
                          r
                          k ) ) ) ) )
```

In this example, the recursion is achieved through the eval label. When this method is selected, the eval variable is bound to the very generic function that was invoked. In other words, the definition of a method is parameterized by the generic function through which it is invoked.

This recursion not only concerns the added method, it also concerns all the previous methods that were present in the generic function as well as the other methods that will be added. Pursuing our example, one may add a method for binary sequences (of class <Sequence>) as in:

```
(method (eval (s <Sequence>) r k)
  (send eval (Sequence-first s)
             r
             (lambda (result)
               (send eval (Sequence-last s) r k) ) ) )
```

This method allows not only sub-expressions of the <Sequence> class to invoke recursively the complete evaluation generic function, but also sub-expressions of the <Alternative> class, to invoke as well the same complete evaluation generic function. Adding a method to a bag of methods make all recursive invocations present in the methods of this bag to invoke the new enriched generic function. Of course, this does not impact the old generic function value which keeps its recursive behavior unaltered: all methods of its bag continue to be mutually recursive and ignore the new methods that were added.

### 2.4 The call-former-method Feature

The call-next-method of CLOS allows a method to invoke the next most specific method of the current generic function. This special form may only appear within a method special form. The notion of the "next most specific method" is well defined if, as in Cecil [Cha92] or MeroonV3 [Que93], conflicting multi-methods are detected when added.

We propose another feature named `call-former-method` that corresponds to a different walk of methods. Let us show this feature in action. Continuing our running example, suppose we want to trace the evaluation of sequences: we may change the method, previously associated to `<Sequence>` in Section 2.3, into:

```
(method (eval (e <Sequence>) r k)
  (display `(eval <Sequence> <-- ,e ,r ,k))(newline)
  (send eval (Sequence-first e)
             r
             (lambda (result1)
               (send eval (Sequence-last e)
                          r
                          (lambda (result2)
                            (display `(eval <Sequence> -->
                                           ,result2) )(newline)
                            (k result2) ) ) ) ) )
```

This definition is sub-optimal since it duplicates the code of the former method. It would be better to refer to the former method without having to know how it is defined. If the former method was not replaced by the new one (i.e., was not triggered by the same classes) then, the former method is normally reachable via `call-next-method`. Otherwise, `call-former-method` allows to refer to it. Then, a better way to trace sequences is to write:

```
(method (eval (e <Sequence>) r k)
  (display `(eval <Sequence> <-- ,e ,r ,k))(newline)
  (call-former-method e r (lambda (result)
                            (display `(eval <Sequence> -->
                                           ,result) )(newline)
                            (k result) )) )
```

In this example, we create a new generic function that behaves as the old one except on instances of `<Sequence>` where the new behavior is the old one surrounded by `display` forms. The `call-former-method` special form invokes the former method with possibly modified arguments (for non discriminating variables only). Recursive calls inside the methods of the former method are still relative to the entire generic function, therefore all calls on `<Sequence>`s will be traced.

The `call-former-method` feature is not similar to what might be called `call-former-generic`. This latter may be programmed as:

```
(let ((former-generic evaluate))
  (method (eval (e <Sequence>) r k)
    (display `(eval <Sequence> <-- ,e ,r ,k))(newline)
    (send former-generic e r (lambda (result)
                               (display `(eval <Sequence> -->
                                              ,result) )(newline)
                               (k result) )) ) )
```

In this example, when a <Sequence> is traced, its sub-expressions will use the former generic function, therefore sub-expressions of class <Sequence> will not be traced.

The call-former-method allows effects similar to the :before and :after method combination operators of CLOS. But above all, call-former-method allows to compose methods, preserving at the same time: the immutability of generic functions and the reuse of shadowed methods. This latest feature favors the propagation and the enrichment of interesting generic functions.

## 3 Implementation

In this section, we first present an implementation of the previously mentioned features: generic, method, add-methods and send. This implementation uses Scheme [CR91] for convenience. Second, we expose how to select an appropriate method, given the constraints of distribution. Third and finally, we detail a specific and compact encoding for generic functions.

Recall that the following definitions are parameterized with the functions of table 1.

### 3.1 Basic Features

A generic function is a structure with four slots:

```
(define-class <Generic> <Object>
  (label signature default methods) )
```

The label is a name identifying the generic function for debug purposes. The signature tells the arity of the function as well as which variables are discriminating (to simplify this implementation, only one discriminating variable, in first position, is allowed). The default method is the method that is selected when no other more specific method can be found. This method is used when the discriminating value is not an instance of <Object> (this may be the case for regular values in Scheme, objects that are not instances of the java.rmi.Remote interface in Java, etc.). The fourth slot holds the dispatching structure that allows the selection of an appropriate method (more on this subject in Section 3.2).

The creation of a generic function is implemented as a Scheme macro:

```
(define-macro (generic call . body)
  `(let* ((variables
            `(,(car call) ,(car (cadr call)) . ,(cddr call)) )
          (default (lambda ,variables . ,body ))
          (generic (make-Generic
                     ',(car call)        ;label
                     ',(cdr call)        ;signature
                     default             ;default
                     (no-methods) )) )   ;methods
```

```
    (insert-method generic <Object> default) ) )
```

An installable method is represented by a structure with three fields:

```
(define-class <InstallableMethod> <Object>
  (class signature pre-method) )
```

An installable method can only be added to a generic function with a compatible signature. The method will be obtained from this installable method after applying (this should be considered as linking) *pre-method* to the class that will trigger the selection of this method and to the generic function that is extended (this is needed to implement `call-former-method`).

```
(define-macro (method call . body)
  `(make-InstallableMethod
    ,(cadr (cadr call))                 ;class
    ',(cdr call)                        ;signature
    (lambda (former-generic current-class) ;pre-method
      (lambda (,(car call) ,(car (cadr call)) . ,(cddr call))
        (define (call-next-method)
          ((lookdown ,(car call) (super-class current-class))
           ,(car call) ,(car (cadr call)) . ,(cddr call) ) )
        (define (call-former-method _1_ . ,(cddr call))
          ((lookdown former-generic current-class)
           ,(car call) ,(car (cadr call)) . ,(cddr call) ) )
        . ,body ) ) ) )
```

Observe that in this definition, the class that triggers the selection of the method is obtained via regular evaluation. This avoids a specialized, shared, mutable name-space.

The `add-methods` function installs installable methods. It checks the compatibility of signatures between the generic function and the methods to be installed (with the restrictions we chose for discriminating variables, we just check that the arities are the same). After this verification, the method is inserted in the dispatching structure of the new generic function.

```
(define (add-methods generic . installable-methods)
  (define (add-one-method g installable-method)
    (if (congruent?
         (Generic-signature g)
         (InstallableMethod-signature installable-method) )
        (let ((class (InstallableMethod-class
                      installable-method )))
          (insert-method
           g
           class
           ((InstallableMethod-premethod installable-method)
            generic class ) ) )
        (error 'add-methods "Non congruent signatures") ) )
  (let add ((g generic)(ims installable-methods))
```

| Node | **Meaning** |
|---|---|
| `(default)` | The appropriate method is the default method. |
| `(method .` $m$`)` | The appropriate method is $m$. |
| `(if` *class yes-tree no-tree*`)` | If the receiver is an instance of *class*, then search in *yes-tree* otherwise search in *no-tree*. |

Table 2: Nodes of the dispatching structure

```
    (if (pair? ims)
        (add (add-one-method g (car ims)) (cdr ims))
        g ) ) )
```

Finally, the `send` function uses the `invoke` function which takes care of the details of the particular calling protocol of the selected method (different protocols may be supported at the same time). The site where the invoked method resides may also be taken into account so the arguments of the invoked method may be made closer by replication or migration.

```
(define (send generic receiver . arguments)
  (invoke (if (is-object? receiver)
              (lookdown generic (class-of receiver))
              (Generic-default generic) )
          generic receiver arguments ) )
```

### 3.2 Dispatching Structure

The dispatching structure is initialized with `(no-methods)`, enriched with `insert-method` and searched via `lookdown` (see [Duc97]). Usual techniques cannot be used. The Smalltalk lookup procedure uses linked mutable dictionaries and do not support multi-methods. The big matrix solution (more or less compressed [VH94]) with rows corresponding to generic functions and columns corresponding to classes cannot be used either since it requires to know all the classes that exist. To use caches, as in [KR90], does not provide *per se* any technique for method lookup, but speeds up the dispatching process if some sort of repetition occurs. We propose a solution inspired from decision trees (similar to [Que98]) which is compatible with the constraints of the distributed world.

The dispatching structure is an ordered tree with three possible nodes as shown on table 2. The two nodes `default` and `method` correspond to leaves of the tree. The tree is quite naturally searched with `lookdown`.

```
(define (lookdown generic class)
  (define (search tree)
    (case (car tree)
      ((default) (Generic-default generic))
```

```
      ((method)  (cdr tree))
      ((if)      (if (subclass? class (cadr tree))
                     (search (caddr tree))
                     (search (cadddr tree)) )) ) )
  (search (Generic-methods generic)) )
```

What is surprising, at first, with this dispatching tree, is that we start the search with `<Object>` and then look down the tree of classes until finding the appropriate method; most techniques start from the class of the receiver and look up till `<Object>`.

With this organization, it is simple to explain how to initialize and enrich such a dispatching tree. The main difficulty is to keep the tree ordered that is, to always check against super-classes before sub-classes.

```
(define (no-methods)
  '(default) )

(define (insert-method generic class method)
  (define (insert tree topclass)
    (case (car tree)
      ((method default)
       (if (eq? class topclass)
           (cons 'method method)
           (list 'if class (cons 'method method) tree) ) )
      ((if) (cond
             ((subclass? class (cadr tree))
              (list 'if (cadr tree)
                    (insert (caddr tree) (cadr tree))
                    (cadddr tree) ) )
             ((subclass? (cdar tree) class)
              (list 'if class
                    (list 'if (cadr tree)
                          (caddr tree)
                          (cons 'method method) )
                    (cadddr tree) ) )
             (else (list 'if (cadr tree)         ;some latitude here
                         (caddr tree)
                         (insert (cadddr tree) topclass) )) ) )) )
  (make-Generic
   (Generic-label generic)
   (Generic-signature generic)
   (Generic-default generic)
   (insert (Generic-methods generic) <Object>) ) )
```

There is some latitude in the above `else` clause since testing an instance against two unrelated classes may be differently done. A more clever solution would be to dynamically profile the use of the dispatching tree and to rearrange it for speed. This transformation is better described by the following rule where

$c_1$ and $c_2$ are unrelated classes i.e., not in sub- or super- class relationship.

(if $c_1$ $m_1$ (if $c_2$ $m_2$ $m_3$)) $\equiv$ (if $c_2$ $m_2$ (if $c_1$ $m_1$ $m_3$))

Observe that the dispatching tree technique only uses the `subclass?` predicate and does not require to know the full hierarchy of classes. Moreover, when some methods are added to a generic function, the entire set of methods of the new generic function is known and a new appropriate dispatching tree can be computed. Also note that the dispatching tree is not only used for method lookdown, it also contains all the necessary information to enrich generic functions i.e., to compute enriched dispatching trees: there is no need to keep a separate data structure for that goal.

### 3.3 Linearizing the Dispatching Tree

If generic functions are to be used, they must not be inefficient. A tree is a set of nodes that, if implemented as separate objects, have to be related with pointers. Therefore it is tempting to coalesce the whole tree into a more compact linearized data structure so it may be read or marshalled efficiently.

To linearize a dispatching tree is easy. The linearization may also be compact if we are able to distinguish methods from numbers. We assume that the predicate `method?` is true on methods and false on natural numbers. If an implementation uses tags to distinguish pointers from numbers, this predicate is trivial. The two leaf nodes `(default)` and `(method .` $m$`)` can be represented by the (possibly default) method they stand for. An `if` node is linearized into a sequence starting with a number followed by the class followed by the linearization of the *yes-tree* followed by the linearization of the *no-tree*. To be able to skip the *yes-tree*, in order to jump onto the *no-tree*, one has to know the length of the linearization of the *yes-tree*: this is precisely the value of the first number[1].

Here is an example of a dispatching tree followed by its linearization.

(if $c_1$ (if $c_2$ $m_2$ $m_1$) (default)) $\equiv$ #(6 $c_1$ 3 $c_2$ $m_2$ $m_1$ $m_{default}$)

The variant of `lookdown` that searches through a linearized dispatching tree can be straightforwardly implemented. We skip over the new definition of `insert-method` that does not bring much. Note also that the linearized code is position independent, this eases the insertion of new methods.

```
(define (lookdown generic class)
  (let* ((instructions (Generic-methods generic))
         (pcmax        (vector-length instructions)) )
    (define (search pc)
      (if (< pc pcmax)
          (let ((word (vector-ref instructions pc)))
            (if (method? word)
```

[1]If profiling is adopted as suggested above, a counter may follow the class in an `if` node. The size of a linearized tree would then be an odd number making the difference between a method and an odd number simpler yet to recognize.

```
                word
                (let ((word2 (vector-ref instructions (+ pc 1))))
                  (if (subclass? class word2)
                      (search (+ pc 2))
                      (search (+ pc word)) ) ) ) ) )
          (Generic-default generic) ) )
    (search 0) ) )
```

To linearize a dispatching tree has a number of advantages: it uses less memory and is more compact since pointers are replaced by sequential access in a vector, it also speeds up `lookdown` since there is no pointer to follow any longer, finally, the dispatching tree can be marshaled as a single entity (and may even be coalesced with the first three slots of the generic function).

## 4 Realization

To incorporate these ideas into a language where every value is an object does not take into account the unavoidable heterogeneity of the current distributed world. We nevertheless intend to adapt these ideas to Icslas [QD93], a distributed and concurrent extension of Scheme with an all-object implementation above DMeroon.

Our first real experiment was led with DMeroon [Que95] bound with Scheme (more precisely Bigloo by Manuel Serrano). A generic function is implemented as a DMeroon object which may be passed, enriched and used everywhere. Scheme closures are encapsulated into DMeroon instances of the immobile (i.e., un-marshallable) `<SchemeMethod>` class. When such a method is invoked, its arguments are marshaled to the original site where sits the closure and the appropriate invocation is performed there. The result is marshaled back to the site of the caller.

Besides the `<SchemeMethod>` class, stands the `<CMethod>` class which allows to invoke C functions, on their site, on Scheme or DMeroon values. In these two cases, the class that encapsulates the real thing to invoke, specifies the calling protocol while the knowledge of the site where sits the method (as brought by DMeroon) allows to forward the call appropriately.

To fit within the framework of DMeroon, we had to slightly change the linearization previously described. We mainly had to segregate methods, classes and numbers into three different homogeneous zones.

To come back to the previous linearization example, here follows the new encoding mimicked by Scheme vectors. Positive numbers are indexes within the sequences of classes or methods (the position tells which one is intended (in the next example, class indexes are in italics while method indexes are left in typewriter font)) while negative numbers stand for the length of *yes-tree*s.

(if $c_1$ (if $c_2$ $m_2$ $m_1$) (default)) $\equiv$
  #( #($c_1$ $c_2$)                ; *classes*
     #($m_{default}$ $m_1$ $m_2$)        ; *methods*

```
    #(-6 0 -3 1 2 1 0) )      ;numbers
```

In these experiment, we also offer the higher-level `define-generic` and `define-method` special forms of CLOS or MEROONV3 since they are quite simple arrangement around `generic` and `method`. Observe how the label allowing recursion and the variable that names the generic function are confused:

```
(define-macro (define-generic call . body)
  `(define ,(car call)
     (generic ,call . ,body) ) )
(define-macro (define-method call . body)
  `(set! ,(car call)
         (add-methods ,(car call)
                      (method ,call . ,body) ) ) )
```

Our second experiment is still in progress, it aims to introduce distributed generic functions in Java and to make them compatible with those of the first experiment via the Remote Method Invocation protocol of Java. We plan to use the new reflection facilities of Java 1.1 to be able to embed methods inside objects and to perform computed calls to them. In this experiment, we will not need DMEROON any longer since Java offers its own distributed model.

## 5 Conclusion

The omni-presence of the network radically changes our everyday practice: more and more, we use binary libraries without access to their source code; we also have access to many existing entities for which we wish to invent new uses (for instance, xmosaic was invented as a new way to use ftp sites), etc. In this setting, companies are more than less tempted to elaborate and enforce standards to freeze as quickly as possible the situation in their favor. It is urgent to give back some dynamicity to users.

In this paper, mainly because they separate behaviors from classes, we propose to resurrect generic functions and to adapt them to the difficult constraints of a multi-user distributed world: our generic functions are immutable and do not rely on any global knowledge. We also propose a new inner special form, `call-former-method`, to enhance reuse of generic functions. Finally, some implementation techniques are discussed that improve lookdown and marshaling.

## 6 Acknowledgments

Thanks to Luc Moreau who dares reading the first drafts of this paper and to all the referees whose advices greatly improved this paper.

## References

[App92] Apple Computer, Eastern Research and Technology. *Dylan, An object-oriented dynamic language.* Apple Computer, Inc., April 1992.

[BC87] Jean-Pierre Briot and Pierre Cointe. A uniform model for object-oriented languages using the class abstraction. In *IJCAI '87*, pages 40–43, 1987.

[BDG+88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp object system specification. *SIGPLAN Notices*, 23, September 1988. special issue.

[Cha92] Craig Chambers. Object-oriented multi-methods in cecil. In O Lehrmann Madsen, editor, *ECOOP '92 — 6th European Conference on Object-Oriented Programming*, number 615 in Lecture Notes in Computer Science, pages 33–56, Utrecht (The Netherlands), June 1992.

[CR91] William Clinger and Jonathan A Rees. The revised$^4$ report on the algorithmic language Scheme. *Lisp Pointer*, 4(3), 1991.

[Duc97] R. Ducournau. La compilation de l'envoi de message dans les langages dynamiques. *L'Objet*, 3(3):241–276, 1997.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[KR90] Gregor Kiczales and Luis Rodriguez. Efficient method dispatch in pcl. In *LFP '90 – ACM Symposium on Lisp and Functional Programming*, pages 99–1052, Nice (France), June 1990.

[QD93] Christian Queinnec and David De Roure. Design of a concurrent and distributed language. In Robert H Halstead Jr and Takayasu Ito, editors, *Parallel Symbolic Computing: Languages, Systems, and Applications, (US/Japan Workshop Proceedings)*, volume Lecture Notes in Computer Science 748, pages 234–259, Boston (Massachussetts USA), October 1993.

[Que93] Christian Queinnec. Designing MEROON v3. In Christian Rathke, Jürgen Kopp, Hubertus Hohl, and Harry Bretthauer, editors, *Object-Oriented Programming in Lisp: Languages and Applications. A Report on the ECOOP'93 Workshop*, number 788, Sankt Augustin (Germany), September 1993.

[Que95] Christian Queinnec. DMEROON: Overview of a distributed class-based causally-coherent data model. In Takayasu Ito, Robert H Halstead, Jr, and Christian Queinnec, editors, *PSLS 95 – Parallel Symbolic Langages and Systems*, Lecture Notes in Computer Science 1068, pages 297–309, Beaune (France), October 1995.

[Que96] Christian Queinnec. *Lisp in Small Pieces*. Cambridge University Press, 1996.

[Que98] Christian Queinnec. Fast and compact dispatching for dynamic object-oriented languages. *Information Processing Letters*, 64(6):315–321, January 1998.

[Sie95] Jon Siegel. *Corba, Fundamentals and Programming*. John Wiley and Sons, 1995.

[VH94] Jan Vitek and R Nigel Horspool. Taming message passing: Efficient method look-up for dynamically typed languages. In *ECOOP '94 — 8th European Conference on Object-Oriented Programming*, Bologna (Italy), 1994.

[VHK97] J. Vitek, R.N. Horspool, and A. Krall. Efficient type inclusion tests. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Application, OOPSLA'97*, Atlanta, GA, October 1997. ACM Press.