MEROON:
A Small, Efficient and Enhanced Object System

Christian Queinnec*
École Polytechnique & INRIA–Rocquencourt

This report describes an object system called MEROON *that runs on top of Scheme. Due to the unavoidable changes that will certainly occur in the future, the user is urged to refer to the documentation packaged with the sources of* MEROON. *This documentation is guaranteed to be accurate and up to date since parts of it are automatically extracted from the source files and examples are run with the current version of* MEROON. *This is achieved using* LiSP2TEX. *This documentation (Revision : 2.9) describes the version of* MEROON *identified as:*

```
Meroon $Revision: 2.22 $ of $Date: 1992/11/12 18:37:32 $.
```

MEROON *sources are gathered in a file named* `meroon*.tar.Z` *and can be anonymously* `ftp`*-ed from the Scheme repository* `nexus.yorku.ca:pub/scheme/` *or from* `nuri.inria.fr:INRIA/icsla/`.

A mailing list exists, `meroon-info@cornas.inria.fr`, *where* MEROON *users can discuss. You can subscribe to it sending a message to* `meroon-request@cornas.inria.fr`. *The author can be sent e-mail at* `Christian.Queinnec@inria.fr`

This document is part of a book in progress (but in french) on the implementation of Lisp. Permission to use this document with the MEROON *object system is granted for all regular academic purposes.*

1 Reference Manual

The following documentation assumes some familiarity with object systems and their terminology. The design of this object system was strongly influenced by ObjVlisp [Coi87, BC87], Alcyone [Hul85] and CLOS [BDG+88, KdRB92]. A late influence from Picolo [Del89] drove the implementation. The resulting system is called MEROON and supports much of the ideas of [QC88, Que90].

MEROON is a very simple object system and therefore is easy to master at the usual level or at the meta-level. Regular usage requires to know only three macros (`define-class`, `define-generic` and `define-method`) as well as some naming rules to be able to use all the various functions that are automatically generated when classes are defined. Yet MEROON is not too inefficient since to send a message costs approximatively 2.5 function calls independently of the number of methods defined on the class. Moreover MEROON is easily portable. Some parts may of course be improved if functions can be inspected or if weak pointers are offered in the underlying Scheme system. MEROON also provides some features not found elsewhere such as the possibility to have indexed fields combined with unrestricted inheritance. This capability allows an implementation to only provide MEROON objects instead of vectors and structures etc.

The names of MEROON primitives were chosen to avoid name clashes with its great rival: CLOS[1] ! The name of MEROON itself comes from the nickname my son gave to his teddy bear.

*LIX, École Polytechnique, 91128 Palaiseau Cedex, France. Email Address: `queinnec@poly.polytechnique.fr` This work has been partially funded by GDR Programmation.

[1] ☺

1.1 Class

A class is defined using `define-class`. To define a class, you must give its name, the name of its superclass, the names and characteristics of its fields as well as some options. These fields are suffixed to the fields already present in the definition of the superclass (this facility is called slot inheritance). All MEROON objects have a contiguous representation i.e. are implemented in single vectors. Access to their fields do not require indirection through other objects. When a class is defined, many functions are also created: constructors of instance, selectors (field readers and writers) etc. These functions may be retrieved from the class itself considered as an object but are usually defined as the regular value of some variables whose name can be derived from the name of the class itself.

`(define-class` *class-name superclass-name* [*Macro*]
(field-description...)
class-options...)

class-name and *superclass-name* are regular identifiers. *superclass-name* must name a known class. A class is known from MEROON as soon as its defining form is expanded. It is thus possible to define dependent classes in one go. Some predefined classes exist, the most important being `Object`, the root of the inheritance graph.

A *field-description* is either an identifier naming a regular field or a list the `car` of which is the symbol `*` or `=`. A field defined as (`*` *field-name*) is an indexed field. An indexed field holds a sequence of items. The length of this sequence is defined at allocation time. A field defined as *field-name* can also be alternatively defined as (`=` *field-name*). These two syntaxes both define a regular field which can only hold a single value. This syntax is supported to allow the insertion of future extra field options such as typed field, or bounded indexed field ... left as metalevel exercises.

The `:immutable` field option specifies that a field is immutable. An immutable field keeps the value it was given at instance creation.

The `:immutable` class option is a short-hand that means that all proper fields, i.e. the fields that are specifically mentionned, are immutable.

The `:careless` field option specifies that the accessors will be careless i.e. they do not check that they only apply on correct instances of the class. They are faster than careful accessors but more dangerous, they are reserved for users that type-check their programs in their head and never make type mistakes[2].

The `:careless` class option is a short-hand that means that all proper fields, i.e. the fields that are specifically mentionned, are careless.

Another class option is `:prototype` which is useful when compiling separately classes definitions. This option allows to register a class which might then be subclassed. The conformity of the real class will be checked at load-time i.e. the prototype must define the same number of fields with the same names.

Two more options were recently added: `:metaclass` and `:eponymous`. By default, classes are anonymous. They can only be retrieved using `symbol->class` even if they are the predefined classes like `Object` or `Class`. If you prefer a class object to be bound to the eponymous variable, use the `:eponymous` class option.

By default the class of the classes that are created by `define-class` is `Class`. You can change this default with `:metaclass` and thus subclass `Class`.

Remark

The MEROON object which corresponds to the created class can be obtained through the specialized function `symbol->class`. `define-class` does not define the class as the value of a variable unless you use the `:eponymous` class option. Classes are truely anonymous even the predefined classes such as `Object` or `Class`. You can obtain the same effect as the `:eponymous` class option with your own macro, provided you use it at toplevel so that the assigned binding for `name` is the global one.

```
(define-oo-macro (my-define-class name super fields . class-options)
```

[2] It's perhaps a bad idea to propose this option which I never use. This might disappear but was recorded here to maintain the documentation uptodate.

```
  `(begin (define-class ,name ,super ,fields . ,class-options)
          (set! ,name (symbol->class ',name)) ) )
    alternatively
(define-oo-macro (my-define-class name super fields . class-options)
  `(define-class ,name ,super ,fields :eponymous . ,class-options) )
```

Examples

A planar point may be defined alternatively as:

```
(define-class Point Object (x y))   or
(define-class Point Object ((= x) (= y)))
```

A polygon, considered as a sequence of points, can be defined as:

```
(define-class Polygon Object ((* point)))
```

Instances of `Polygon` are sequences of points. Two different instances may contain a different number of points.

A colored polygon is simply a polygon with a color. `Colored-Polygon` can be defined as:

```
(define-class Colored-Polygon Polygon ((= color :immutable :careless)))
```

An instance of `Colored-Polygon` is thus a sequence of points followed by a color. The `color` field is immutable. The previous definition could also be written as:

```
(define-class Colored-Polygon Polygon (color) :immutable :careless)
```

That means that the `color` field, the proper field of `Colored-Polygon`, is read-only. This does not change the mutability of the `x` and `y` fields of an instance of `Colored-Polygon`. The `:careless` option will be discussed below.

Finally a colored polygon might be known under different nicknames:

```
(define-class Nicknamed-Colored-Polygon Colored-Polygon
  ((* nickname)) )
```

An instance of a `Nicknamed-Colored-Polygon` is a sequence of points followed by a color followed by the sequence of its nicknames. An instance of `Nicknamed-Colored-Polygon` inherits a `color` field from `Colored-Polygon`: this field is immutable.

Suppose you defined the `Colored-Polygon` in module $F1$, suppose also that you want to define a new class named `Nicknamed-Colored-Polygon` in module $F2$. To compile this latter class the compiler needs to know the structure and the names of the fields of the former class. You can inform the compiler of the structure of `Colored-Polygon`, using the `:prototype` option. Just prefix the `Nicknamed-Colored-Polygon` definition with the declaration of the `Colored-Polygon` class as in:

```
(define-class Polygon Object ((* point)) :prototype)
(define-class Colored-Polygon Polygon (color) :immutable :prototype)
```

You may then define the `Nicknamed-Colored-Polygon` class as well as other classes. When the module $F2$ will be loaded, the prototype will be checked for conformity i.e. the class `Colored-Polygon` will be checked to have two fields named `point` and `color`.

1.2 Predicate

When a class is defined a predicate is automatically generated. Its name is the name of the class suffixed by a question mark. This predicate can be applied to any object and returns true only if its argument belongs to the class or to one of its subclasses.

Due to the embedding of MEROON in Scheme, the `Object?` predicate is not true of any object. It tries to recognize all MEROON objects but can err on Scheme vectors which have a number as first coordinate. This cannot be corrected safely and efficiently in pure Scheme since it is not possible to create new types.

Examples

Continuing the previous example, any colored polygon is a polygon and so is any nicknamed colored polygon. A polygon might be a pure polygon or a colored polygon. In other words, if `(Colored-Polygon?` *o*`)` is true then `(Polygon?` *o*`)` is true.

1.3 Allocator

An allocator is a function that returns a bare instance i.e. an instance with empty slots. The allocator takes as much numeric parameters as there are indexed fields in the class. The name of the allocator is the name of the class prefixed with `allocate-`. The content of the fields of a bare instance are initially unspecified but they can be initialized with the generic function `initialize!` which is automatically applied on any freshly allocated objects. The default method of this generic function is to leave the object as it is, `initialize!` can be specialized to handle more complex initializations.

The `make-vector` function of standard Scheme is a kind of allocator since `(make-vector 4)` returns a vector with four coordinates whose content is unspecified.

Examples

An instance of Point can be allocated by `(allocate-Point)`. A polygon defined with three points is returned by `(allocate-Polygon 3)`. A colored polygon with four points and two nicknames is the result of `(allocateNicknamedColoredPolygon 4 2)`.

1.4 Constructor

A constructor is a function that allocates and initializes instances. It thus takes arguments with which the fields of the allocated instance are initialized. Regular fields are just given while indexed fields are prefixed by their repetition number followed by the values of these fields. All the fields of the allocated instance must be initialized. The instance yielded by a constructor is not submitted to the `initialize!` generic function. The name of the constructor is the name of the class prefixed with `make-`.

The `cons` function is an example of constructor in standard Scheme since it allocates a pair and fills its two slots at the same time.

Examples

A point with coordinates 22 and 44 can be created by evaluating `(make-Point 22 44)`. A colored polygon with two points and three nicknames is returned by:

```
(make-Nicknamed-Colored-Polygon
   2 (make-Point 22 44) (make-Point 51 90)
   'blue
   3 'Joe 'Jack 'Jill )
```

Note the numbers 2 and 3 that prefix the given points and nicknames.

1.5 Field Reader

An instance is a data aggregate. Individual components can be retrieved using field readers. The name of a field reader function is the name of the class suffixed by a dash and the name of the field. When the field is regular i.e. not indexed, then the field reader returns the content of the field. When the field is indexed, the field reader must be called with an index and returns the content of the indexth item of the field considered as a sequence of items. Sequences are zero-based. An error is raised if the index is out of the sequence. If the field was defined as not `:careless` then an error will be raised if trying to read the content of this field from an instance of a incorrect class. When a field is careless, the same field of indirect instances is also careless.

Examples

Suppose that the previous instance of `Nicknamed-Colored-Polygon` is the value of the variable `ncp`, then:

```
(Nicknamed-Colored-Polygon-point ncp 1)          → <Point 51 90>
(Polygon-point ncp 0)                            → <Point 22 44>
(Colored-Polygon-color ncp)                      → blue
(Nicknamed-Colored-Polygon-nickname ncp 2)       → Jill
```

The `Polygon-x` and `Nicknamed-Colored-Polygon-nickname` are careful readers so they can only be applied on instances of `Polygon` and `Nicknamed-Colored-Polygon`. Since `Colored-Polygon-color` is careless, it can only be safely applied on instances of `Colored-Polygon` and gives unspecified results elsewhere.

1.6 Field Writer

The content of fields can be altered with field writers. The name of a field writer is the name of the associated field reader prefixed by `set-` and suffixed by an exclamation mark. When a field is regular, not indexed, the field writer takes two arguments: the instance and the new value. When a field is indexed, the field writer takes the instance, the index and the new value. Sequences are zero-based. An error is raised if the index is out of bounds. The value returned by a field writer is unspecified.

Examples

Suppose that the previous instance of Nicknamed-Colored-Polygon is still the value of the variable `ncp`, then the following forms are regular forms:

```
(set-Nicknamed-Colored-Polygon-point! ncp 1 (allocate-Point))
(set-Polygon-point! ncp 0 (Colored-Polygon-point ncp 1))
(set-Nicknamed-Colored-Polygon! ncp 2 'Dick)
```

The following form is an error since `color` is an immutable field.

```
(set-Colored-Polygon-color! ncp 'purple)
```

`set-Colored-Polygon-color!` is a careless writer with unspecified results if applied on non-instances of `Colored-Polygon`.

1.7 Lengther

An instance which contains an indexed field can be inquired to return the length of this indexed field. The name of the lengther is the name of the associated field reader suffixed by `-length`. Errors concerning index out of bounds can be avoided by appropriate use of lengthers.

Examples

Always supposing that the previous instance of `Nicknamed-Colored-Polygon` is the value of the variable `ncp`, then:

```
(Nicknamed-Colored-Polygon-nickname-length ncp) → 3
(Colored-Polygon-point-length ncp)              → 2
```

2 Generic Functions

A generic function is a bag of functions known as methods. When invoked on a MEROON object a generic function determines the class of the discriminating variable and invokes the appropriate method. Generic functions implement single inheritance. A generic function is defined using `define-generic` macro.

`(define-generic (`*generic-name variable-list*`)` [*Macro*]
[default-body] `)`

A generic function has a global name under which it can be retrieved (see function `symbol->generic`). Like a normal function, the generic function is defined with a list of variables among which only one discriminating variable must appear syntactically enclosed in a list. A generic function can be defined with a default body which will be invoked if no method can be found for the discriminating variable. This feature is particularly useful since MEROON is embedded in regular Scheme: a generic function applied on a non-MEROON object invokes the default body. The default default-body is to signal a Domain anomaly since there is no appropriate method.

Some predefined generic functions exist which can be specialized. These are

`(initialize!` *object*`)` This generic function is invoked on any freshly allocated objects. This function is only applied on instances created by allocators, it is not applied on instances created by constructors.

`(clone` *object*`)` This generic function copies a MEROON objects. Only the toplevel spineal structure is copied i.e. the copy is non recursive.

`(show` *object [stream]*`)` This generic function can be used to print any kind of objects or other Scheme entities. Objects are predefined to be printed as `#<`*Something*`>` where *Something* is the name of the class of the object, except classes and fields which are printed in a more eye-opener way.

`(meroon-error` *anomaly*`)` All anomalies reported by MEROON are handled through this generic function. New subclasses of `Anomaly` can be devised and `meroon-error` can be specialized on them.

Examples

The implementation of MEROON itself uses a lot of generic functions specially to synthesize code. The generic function `show` is defined as follows[3]:

```
(define-generic (show (o) . stream)
  (let ((stream (if (pair? stream) (car stream) (current-output-port))))
    (cond ((symbol? o) (display o stream))
          ((boolean? o) (display (if o "#T" "#F") stream))
          ((null? o) (display "()" stream))
          ((string? o) (display o stream))
          ((number? o) (display o stream))
          ((pair? o) (show-list o stream))
          ((vector? o) (show-vector o stream))
          ((procedure? o) (display o stream))
          ((char? o) (display o stream))
          (else (meroon-error
                 (create-Domain-Anomaly
                  (symbol->generic 'show)
                  "Does not know how to show" o ) )) ) ) )
```

It is possible to specialize this behaviour on particular classes of MEROON objects using `define-method`.

3 Methods

Methods can be defined to specialize a generic function provided this generic function already exists. The method which must specialize the generic function must have a compatible variable list. The discriminating variable is the only variable which must appear, associated to a class, in the list of variables.

[3] This code as well as the other excerpts of the sources of MEROON are automatically extracted from the source files of MEROON using a tool (written in Scheme→C) named LiSP2TEX.

`(define-method (`*generic-name variable-list*`)` [*Macro*]
body `)`

Methods can use the form `(call-next-method)` to invoke the method that would have been called if not present. It is analogous to the `send-super` mechanism of other OO languages. The super method (if any) will be invoked with the same arguments as the method itself. The `(call-next-method)` cannot be used out of a method definition.

If there is no appropriate method, a domain anomaly is signalled.

Examples

Let us give the methods specifying how MEROON objects as well as MEROON classes are printed:

```
(define-method (show (o Object) . stream)
  (let ((stream (if (pair? stream) (car stream) (current-output-port)))
        (name (Class-name (object->class o))) )
    (display "#<a" stream)
    ;;a french misinterpretation of english pronunciation
    ;;(case (string-ref (symbol-¿string name) 0)
    ;;((#\a#\e#\i#\o#\u#\y) (write-char #\nstream))
    ;;(else #f) )
    (write-char #\space stream)
    (display name stream)
    (display ">" stream)
    o ) )
(define-method (show (o Class) . stream)
  (let ((stream (if (pair? stream) (car stream) (current-output-port))))
    (display "#<Class: " stream)
    (display (Class-name o) stream)
    (display ">" stream)
    o ) )
```

4 From names to objects

The class of an object can be retrieved with the following function:

`(object->class` *object*`)` [*Function*]

The *object* is assumed to be a real MEROON object. You can approximatively check it with `Object?`. Any MEROON object has a class. Classes themselves are objects and are instances of `Class`. `Class` is itself an instance of itself as predicted by ObjVlisp [BC87, Coi87].

Classes and generic functions can be obtained from their names using the (expensive) `symbol->class` and `symbol->generic` functions.

`(symbol->class` *symbol [default]*`)` [*Function*]

This functions takes a symbol and returns the class which has this name. If no class exists with this name, an error is signalled or, if present, the *default* function is applied on the unfortunate *symbol*.

`(symbol->generic` *symbol [default]*`)` [*Function*]

This functions takes a symbol and returns the generic function which has this name. If no such generic function exists with this name, an error is signalled or, if present, the *default* function is applied on the unfortunate *symbol*. What is returned is the generic object, a MEROON object, corresponding to the name. The generic function itself is the global value of the name.

5 Anomalies

Some anomalous situations are recognized by Meroon and reported to the user. Whenever an anomaly is detected, an instance of `Anomaly` is created and the generic function `meroon-error` is called on it. It is possible to subclass the `Anomaly` class to create new anomalies and new methods. By default, the anomaly is reported in a human-readable form (well, it might be better).

The `Anomaly` actually has three suclasses:

- `Syntax-Anomaly` for all anomalies syntactically detected,
- `Allocation-Anomaly` for all anomalies detected while constructing or allocating instances. These errors usually correspond to bad format of arguments or sizes.
- The last class is `Domain-Class` and usually corresponds when accessing an instance with an inappropriate accessor or when applying a generic function with no specialized method.

6 Miscellaneous library

Two functions are provided to display the offspring of a class or the set of classes on which methods are defined in a generic function.

(show-hierarchy *[class-name [stream]]*) [*Function*]

This function prints the hierarchy of *class-name* on `stream`. By default, *stream* is the current output stream and `class-name` is `Object`. You can alternatively give a class object instead of a name or call `show-hierarchy` without argument to display all classes. Classes are indented to indicate their relation, for example,

```
? (SHOW-HIERARCHY)
Subclass tree of OBJECT
#<Class: OBJECT>
 #<Class: ANOMALY>
  #<Class: ALLOCATION-ANOMALY>
  #<Class: DOMAIN-ANOMALY>
  #<Class: SYNTAX-ANOMALY>
 #<Class: FIELD>
  #<Class: POLY-FIELD>
   #<Class: MUTABLE-POLY-FIELD>
  #<Class: MONO-FIELD>
   #<Class: MUTABLE-MONO-FIELD>
 #<Class: GENERIC>
 #<Class: CLASS>
= #T
```

(show-generic *[generic-name [stream]]*) [*Function*]

This function prints the hierarchy of classes that define methods on the *generic* object. You can alternatively give a generic object instead of a name. If `show-generic` is called without argument, all generic functions are displayed. By default, *stream* is the default output port. For example,

```
? (SHOW-GENERIC 'SHOW)
Methods on SHOW
#<Class: OBJECT>
 #<Class: ANOMALY>
  #<Class: POLY-FIELD>
```

```
 #<Class: MONO-FIELD>
#<Class: GENERIC>
#<Class: CLASS>
= #T
```

Finally, some figures about Meroon can be displayed with `show-meroon`.

(`show-meroon` *[stream]*) [*Function*]

For example,

```
? (SHOW-MEROON)

Meroon $Revision: 2.22 $ of $Date: 1992/11/12 18:37:32 $.
Total number of classes: 12
Total number of generic functions: 9
= #T
```

MEROON objects can be displayed with the `show` generic function. It is sometimes useful to display them with as much details as possible. The `unveil` function allows to inspect objects in their most inner details even in the case of cyclic structures.

(`unveil` *object [stream]*) [*Function*]

The `unveil` function stops on non-MEROON objects which are just `show`-n. You can customize this behavior adding new methods on the `show-unveiled` generic function. Here is an example of `unveil` which, besides, is an example of a function using the meta object protocol as described in the next section to know the structure of its argument. This allows `show-unveiled` to be described by a single (meta-)method.

```
? (UNVEIL (SYMBOL->CLASS 'POLY-FIELD))

(a CLASS <------------- [Id: 1]
 NAME: POLY-FIELD
 NUMBER: 5
 FIELDS: (#<Field=!NAME> #<Field=!CLASS> #<Field=!READER> #<Field=!LENGTHER>)
 SUPER:
 (a CLASS <------------- [Id: 2]
 |NAME: FIELD
 |NUMBER: 3
 |FIELDS: (#<Field=!NAME> #<Field=!CLASS> #<Field=!READER>)
 |SUPER:
 |(a CLASS <------------- [Id: 3]
 | NAME: OBJECT
 | NUMBER: 0
 | FIELDS: ()
 | SUPER: #F
 | SUBCLASSES: (8 3 2 1)
 | PREDICATE: #*PROCEDURE*
 | ALLOCATOR: #*PROCEDURE*
 | MAKER: #*PROCEDURE* end CLASS)
 |SUBCLASSES: (5 4)
 |PREDICATE: #*PROCEDURE*
 |ALLOCATOR: #*PROCEDURE*
 |MAKER: #*PROCEDURE* end CLASS)
 SUBCLASSES: (7)
```

```
PREDICATE: #*PROCEDURE*
ALLOCATOR: #*PROCEDURE*
MAKER: #*PROCEDURE* end CLASS)= #T
```

7 Meta Object Protocol

Conforming to the ObjVlisp rule, classes are first class MEROON objects which can themselves be inspected as any other MEROON objects. Classes are defined as:

```
(define-class Class Object
  (name number fields super subclasses predicate allocator maker) )
```

The `number` field is the internal number associated to the class (see also the implementation section). The `fields` field contains the list of instances of `Field` describing the fields of instances of classes. The `subclasses` field contain the list of the number associated to the subclasses of the class[4]. The other fields are evident.

Fields are described via field-descriptors which are instances of:

```
(define-class Field Object (name class reader))
(define-class Mono-Field Field ())
(define-class Poly-Field Field (lengther))
(define-class Mutable-Mono-Field Mono-Field (writer))
(define-class Mutable-Poly-Field Poly-Field (writer))
```

It is a good exercice to design new field-descriptors to check the type the content of fields etc. The needed work is to define additional methods on these classes to synthesize code for these new cases. Read the source code for these additions.

A generic function is itself associated to a MEROON object described as:

```
(define-class Generic Object
  (name discriminator dispatch-table default variables))
```

A generic function is implemented as a Scheme function which is the value of the global variable the name of which is the name of the generic function, the associated object can be retrieved using `symbol->generic`. Ideally these two things should be the same object.

The structure of instances of `Field` might change. It is recommanded to use the following function and predicate to access the writer or to inquire the mutability of a field:

`(Field-writer` *field-object*`)` [*Generic function*]
`(Mutable-Field?` *field-object*`)`

Examples

`Class` can be subclassed. For instance, a class which knows the number of direct instances it has can be defined using a special metaclass. First we define a new class of classes with an extra field which will hold the number of created instances.

```
(define-class Counting-Class Class (counter))
```

Whenever such a class is created, the counter is initialized to zero and the constructor is redefined to count its number of invokations. This `initialize!` method is called after the class is initialized.

```
(define-method (initialize! (o Counting-Class))
  (call-next-method)
  (set-counting-class-counter! o 0)
  (let ((original-maker (class-maker o)))
```

[4] These numbers can be coerced into classes using the `number->class` function.

```
    (set-counting-class-maker!
     o (lambda args
         (set-counting-class-counter! o (+ 1 (counting-class-counter o)))
         (apply original-maker args) ) ) )
  o )
```

Then we define the intended class:

```
(define-class Counted-Point Point () :metaclass Counting-Class)
(Counting-Class-counter (symbol->class 'Counted-Point))  → 0
```

Then we try it:

```
(let ((pt (make-counted-point 22 33)))
  (list (counted-point-x pt) (point-y pt)
        (Counting-Class-counter (symbol->class 'Counted-Point)) ) )
  → (22 33 1)
```

Meta Object Protocol is very powerful if understandable.

8 Tracing Generic Functions

To ease debugging, generic functions can be traced. Two levels of trace exist. The first one is the simpler:

`(show-generic-trace` *generic-name ...*`)` [*Function*]

Displays (with the generic function `show`) all the arguments and results of generic functions *generic-name*. You can alternatively give a generic object instead of a name. The trace can be suppressed with:

`(show-generic-untrace` *generic-name ...*`)` [*Function*]

By default, no names to `show-generic-untrace` means suppress tracing on all traced functions.

A more detailed level exist where it is possible to hook specific functions before and after invokation. You can alternatively give a generic object instead of a name.

`(generic-trace` *generic-name before after*`)` [*Function*]

before `= (lambda (`*arguments of the generic function*`) ...)`

after `= (lambda (`*result of the generic function*`) ...)`

Modifies the methods of the *generic* function such that whenever the *generic* function is called, the *before* function is called on the arguments and the result is submitted to the *after* function.

`(generic-untrace` *generic-name*`)` [*Function*]

Reverts the *generic* function to the original untraced generic function.

9 Implementation

MEROON is so small that a single figure is sufficient to express much of the implementation. An object, see figure 1, is represented by a Lisp vector the first coordinate of which is the number associated to its class. The rest of the vector contains the other fields of the object. Consider the above `Nicknamed-Colored-Polygon` class and the now famous `ncp` instance:

```
(define-class Point Object ((= x) (= y)))
(define-class Polygon Object ((* point)))
(define-class Colored-Polygon Polygon ((= color :immutable)))
```

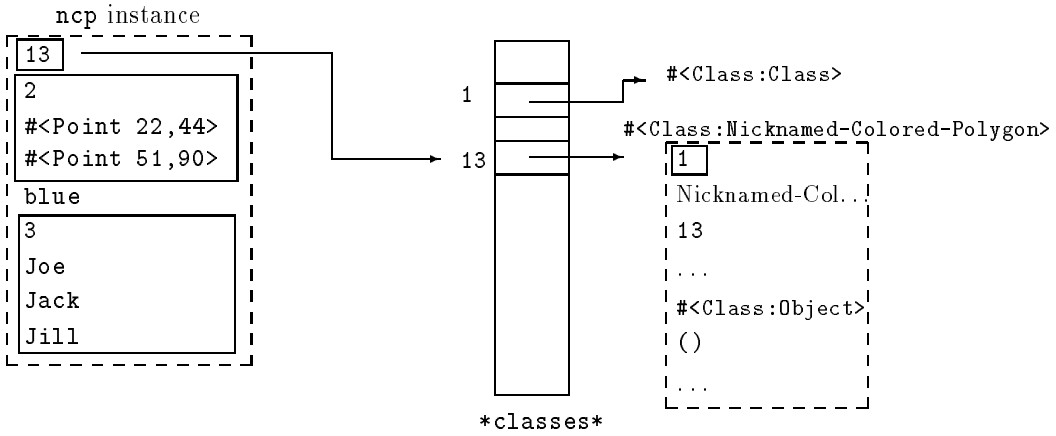

Figure 1: Implementation of an instance

```
(define-class Nicknamed-Colored-Polygon Colored-Polygon
  ((* nickname)) )
(set! ncp
      (make-Nicknamed-Colored-Polygon
         2 (make-Point 22 44) (make-Point 51 90)
         'blue
         3 'Joe 'Jack 'Jill ))
```

The `ncp` instance is implemented as a vector as shown on figure 1.

Note that we put a class-number instead of a class in the first coordinate of an instance. The function `object->class` is then slower but objects are not automatically circular and can be printed or debugged more easily by the underlying Scheme. Regular fields are implemented as coordinates. A sequence of items is prefixed by its length.

A class is represented by a regular MEROON object. The class of classes is indeed defined as:

```
(define-class Class Object
  (name number fields super subclasses predicate allocator maker) )
```

The `name` field holds the symbol which names the class while `number` holds the number associated to the class. Numbers can be coerced into classes using the `number->class` function. All fields are grouped in an ordered list held in `fields`. All fields are instance of the `Field` class which describes the field i.e.whether it is a regular field or a repeated field, mutable or not. The superclass of a class is directly obtained from the `super` field. To manage generic functions, one must know the subclasses of a class: the numbers of these subclasses appear in a list held in `subclasses` to avoid circularities again.

All classes are gathered in a sequence, values of the global variable `*classes*`.

Functions in Scheme or Lisp are atomic objects that cannot be dissecated. Since we decide that generic functions are applyable i.e. can be given to `apply`, they must be represented by native functions[5]. The problem is then to associate fields to a a function. We then decide to name generic functions and to associate informations to this name. This point is the worst feature of MEROON since it breaks the anonymity of generic functions.

The MEROON system is defined in the `meroon.scm` file. Adaptations to Scheme->C or PC-Scheme appear in files `meroon.s2c` or `meroon.pcs`.

[5] Another alternative is to represent generic functions by regular objects. They are therefore not applyable so an appropriate operator, often called `send` or `ask`, must be offered that allows to invoke generic functions.

10 Conclusions

This document described how to use and how is implemented a small object system called MEROON. MEROON offers traditional objects the representation of which is a concatenation of fields. MEROON also offers indexed fields the representation of which is a contiguous sequence of items prefixed by their exact number. Such objects supersede traditional vectors. MEROON provides an uniform and unique framework: all usual objects of Lisp or Scheme can be defined in MEROON and can benefit of generic functions, methods and inheritance.

The implementation is simple but efficient. Generic functions are particularly fast due to the implementation choices. The internal architecture of MEROON uses MEROON instances a lot and can be easily modified or extended. This induces to solve a bootstrap puzzle as well as some other compilation problems. A native implementation may improve this design especially if functional objects can be inspected (i.e. if variables in their closure can be retrieved) and if weak pointers are provided by the underlying Scheme system avoiding to keep all generic functions in a set.

References

[AR88] Norman Adams and Jonathan Rees. Object-oriented programming in scheme. In *Conference Record of the 1988 ACM Conference on Lisp and Functional Programming*, pages 277–288, August 1988.

[ASS85] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Mass., 1985.

[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*, September 1988.

[Coi87] Pierre Cointe. The ObjVlispkernel: a reflexive architecture to define a uniform object oriented system. In P. Maes and D. Nardi, editors, *Workshop on MetaLevel Architectures and Reflection.* North Holland, 1987.

[Del89] Vincent Delacour. Picolo expresso. *Revue Bigre+Globule*, (65):30–42, July 1989.

[Hul85] Jean-Marie Hullot. Alcyone. 1985.

[KdRB92] Gregor Kiczales, Jim des Rivières, and Daniel G Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge MA, 1992.

[Kes88] Robert R. Kessler. *Lisp, Objects, and Symbolic Programming.* Scott, Foreman/Little, Brown College Division, Glenview, Illinois, 1988.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction.* International Series in Computer Science. Prentice-Hall, 1988.

[QC88] Christian Queinnec and Pierre Cointe. An open-ended Data Representation Model for Eu-Lisp. In *LFP '88 – ACM Symposium on Lisp and Functional Programming*, pages 298–308, Snowbird (Utah, USA), 1988.

[Que90] Christian Queinnec. A Framework for Data Aggregates. In Pierre Cointe, Philippe Gautron, and Christian Queinnec, editors, *Actes des JFLA 90 – Journées Francophones des Langages Applicatifs*, pages 21–32, La Rochelle (France), January 1990. Revue Bigre+Globule 69.

A MEROON quick reference card

(define-generic (*generic-name variable...* (*discriminating-variable*) *variable...*)
    [default-body])

(define-method (*generic-name variable...* (*discriminating-variable class-name*) *variable...*)
    body)

(define-class *class-name superclass-name*
    (*field-name*
      (= *field-name* [:immutable] [:careless])
      (* *field-name* [:immutable] [:careless]) ...)
  [:immutable]
  [:careless]
  [:eponymous]
  [:metaclass *class-name*]
  [:prototype])

(*class-name*? *object*)
(make-*class-name field... length values...*)
(allocate-*class-name indexed-field-length...*)
(*class-name-field-name instance*)
(*class-name-indexed-field-name instance index*)
(set-*class-name-field-name*! *instance value*)
(set-*class-name-indexed-field-name*! *instance index value*)
(*class-name-indexed-field-name*-length *instance*)

Predefined classes:

```
(define-class Class Object
  (name number fields super subclasses predicate allocator maker) )
(define-class Generic Object
  (name discriminator dispatch-table default variables))
(define-class Field Object (name class reader))
(define-class Mono-Field Field ())
(define-class Poly-Field Field (lengther))
(define-class Mutable-Mono-Field Mono-Field (writer))
(define-class Mutable-Poly-Field Poly-Field (writer))
(define-class Anomaly Object (intention message (* culprits)))
```

Predefined generic functions:

(show *entity [stream]*)
(clone *object*)
(initialize! *object*)
(meroon-error *anomaly*)

Utility functions:

(object->class *object*) → *class*
(symbol->class *symbol [default-handler]*) → *class*
(number->class *number [default-handler]*) → *class*
(symbol->generic *symbol [default-handler]*) → *generic-function*
(show-generic-trace *['generic-name ...]*)
(show-generic-untrace *['generic-name ...]*)
(unveil *entity [stream]*)