MEROON V3:
A Small, Efficient and Enhanced Object System

Christian Queinnec*
École Polytechnique & INRIA–Rocquencourt

Abstract

This report describes an object system called MEROON V3 that runs on top of Scheme. Distinctive features of MEROON V3 are indexed fields without inheritance restriction (just think to a `ColoredVector` class), code-generating metaclasses and an instantiation protocol that respects immutability. MEROON V3 also supports multimethods, coercers, generic equality, metaclasses but not multiple inheritance.

This release enhances the previous versions with new features that are discussed in a companion paper [Que93]. This documentation refers to MEROON V3, version `3.36`.

MEROON *V3 sources are gathered in a file named* `MeroonV3*.tar.gz` *and can be anonymously retrieved by* `ftp` *from* `ftp.inria.fr:INRIA/Projects/icsla/Programs`*. You may also find it in the Scheme repository located on* `nexus.yorku.ca:pub/scheme`*.*

---

*Laboratoire d'informatique de l'École Polytechnique (URA 1439), 91128 Palaiseau Cedex, France. Email: `Christian.Queinnec@polytechnique.fr` This work has been partially funded by GDR-PRC de Programmation du CNRS.

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 V3 and supports much of the ideas of [QC88, Que90].

Meroon V3 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 four macros (`define-class`, `define-generic`, `define-method` and `instantiate`) 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 V3 is rather efficient. Moreover Meroon V3 is easily portable and actually works with Bigloo, Elk, Gambit, Guile, MIT Scheme, OScheme, Scheme->C, SCM and VSCM. Some parts may of course be improved if functions were inspectable or if weak pointers were offered in the underlying Scheme system. Meroon V3 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 V3 objects instead of vectors and structures etc.

The names of Meroon V3 primitives were chosen to avoid name clashes with its great rival: CLOS[1] ! The name of Meroon V3 itself comes from the nickname my son gave to his teddy bear[2]

1.1 Class

A class is defined using `define-class`. To define a class, you must give its name, the name of its (single) superclass, the names and characteristics of its fields as well as some class options. These fields are suffixed to the fields already present in the definition of the superclass (this facility is called slot inheritance). All Meroon V3 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 accompanying functions are also created: constructors of instance, selectors (field readers and writers) etc.

(`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 V3 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.

A *field-description* may contain some field options which are `:immutable`, `:mutable`, `:initializer` or `:maybe-uninitialized`.

The `:immutable` <u>field</u> option specifies that a field is immutable. An immutable field cannot be modified. The `:immutable` <u>class</u> option is a short-hand that means that all proper fields, i.e., the fields that are specifically mentioned in the definition of the class, are immutable. The actual default is that all fields are mutable.

[1] ☺

[2] Meroon V3 is not an acronym but I am open to hear from such definitions.

The `:initializer` field option specifies how to initialize a field if its initial content is not specified. The `:initializer` takes a niladic function `(lambda () ...)` as argument to initialize a Mono-Field. It takes a monadic function `(lambda (index) ...)` as argument to initialize a Poly-Field where the `index` variable will be bound to the index of the positions of the field to initialize). Fields are initialized from left to right, indexed fields are initialized from index zero.

The `:maybe-uninitialized` field option specifies that this field may be uninitialized. Reading this field will check if the field is defined. All fields that do not have this field option cannot be uninitialized i.e., must always be initialized either by an explicit `:initializer` field option either by a value in all `instantiate` or `co-instantiate` forms.

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.

The `:virtual` class option allows to suppress the generation of the associated coercer, allocator and maker functions.

A last option is the `:metaclass` class option. Classes are instances of `MEROONV2-CLASS` by default. This metaclass ensures that a lot of accompanying functions are defined whenever a class is defined. The `Class` metaclass does not define any accompanying functions.

Classes can be retrieved from their name using the coercer `->Class` even if the predefined like `Object` or `Class`. By the way, any defined class is also the value of the global variable *class-name*`-class`.

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 :maybe-uninitialized)))
```

Instances of `Polygon` are sequences of points. Two different instances may contain a different number of points. Some points within a polygon maybe left uninitialized.

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

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

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

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

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`.

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 suffix 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 as an instance of `MeroonV2-Class`, 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 V3 in Scheme, the `Object?` predicate is not true on any value. It recognizes all MEROON V3 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. It is sometimes possible to overcome this problem in particular implementation if they offer records or structures. This is for instance the case in Bigloo.

The default implementation of `Object?` may be strenghtened if MEROON V3 is compiled with the `safer-object` feature. This will cost you an additional hidden field in all MEROON V3 instances.

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 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 finally 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:

```
(define ncp
  (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.

1.4 Easier allocation

To ease allocation, there now exists three additional macros named: `instantiate`, `co-instantiate` and `with-co-instantiation`. It offers the possibility to specify the values of the fields in whatever order is felt convenient. It also checks statically whether there are uninitialized fields.

```
(instantiate class-name                                    [Macro]
    :mono-field-name form
    :poly-field-name forms ...
    :poly-field-name-length form
    ... )
```

A mono-field is associated to a keyword with its proper name followed by its value. A poly-field is associated to a keyword with its proper name followed by all the possible values of the sequence. It can also be specified by a keyword made from its name and suffix `-length` to just specify the length of the indexed field; in that latter case, the indexed field must have the `:maybe-uninitialized` field option or an anomaly will be signalled.

To be more precise, the constructor function is built upon the `instantiate` macro and provides the user with a different way to allocate objects. The constructor function is useful to allocate fully defined objects but the `instantiate` macro is more powerful. It allows to specify fields in whatever order and also knows how to handle unintialized fields. The constructor function is only created if using the default metaclass as provided by `define-class`.

Examples

A point with coordinates 22 and 44 can be created by evaluating `(instantiate Point :x 22 :y 44)` or `(instantiate Point :y 44 :x 22)`. The previous `ncp` instance could have been defined as:

```
(define ncp
  (instantiate Nicknamed-Colored-Polygon
    :point (make-Point 22 44)
           (instantiate Point :y 90 :x 51)
    :nicknames 'Joe 'Jack 'Jill
    :color 'blue ) )
```

It is not possible to forget, for instance, in the previous instantiation the `color` field since it does not have the `:maybe-uninitialized` field option. It is nevertheless allowed to define a polygon with three points (a triangle ?) as:

```
(instantiate Polygon
  :point-length (+ 2 1) )
```

1.5 Co-instantiation

An often neglected problem is to allocate simultaneously mutually referent objects, the `co-instantiate` and `with-co-instantiation` macros handle this problem.

```
(co-instantiate                                            [Macro]
  (variable class-name
           :mono-field-name form
           :poly-field-name forms ...
           :poly-field-name-length form ... )
  ... )
```

This form creates as many objects as they are clauses following the `co-instantiate` macro keyword. Each clause introduces a name that will name the MEROON V3 instance to be allocated then mentions the name of the class to be instantiated and is followed by the initialization parameters as for the `instantiate` form. A value may be computed using the variables that are simultaneously defined (a little like a letrec form).

The exact processing is as follows:

1. bare objects are allocated and assigned to their names.
2. fields are initialized with the parameters of the clauses. Objects are initialized from left to right. This is no such warranty for fields.
3. uninitialized fields if any are initialized with default initializers if any.
4. the value of the first co-created object is returned.

Examples

Appreciate in this example, the co-creation of two objects. The first one refers to the co-allocated instance of Point, the second is initialized after `p` so it can use the fields of `p`. Default initializers come finally to complete the initialization process.

```
(define-class TrucMuche Object
   ((= left :initializer (lambda () 33))
    (* right) ) )
(define p 'wait)
(define x 'wait)
(co-instantiate
  (p TrucMuche :right x x x)
  (x Point :x 22 :y (TrucMuche-left p)) )
```

1.6 Local co-instantiation

The previous macro generates a `begin` form of `set!` forms. The `with-co-instantiation` form creates co-instantiated local variables and allows some forms to be evaluated in this extended lexical environment. It is rather similar to the previous one.

`(with-co-instantiation` [*Macro*]
  `((`*variable class-name*
      `:`*mono-field-name form*
      `:`*poly-field-name forms* ...
      `:`*poly-field-name*`-length` *form* ... `)`
   ... `)`
 forms... `)`

Example

Let's just adapt our previous example to a local context of use:

```
(with-co-instantiation
     ((p TrucMuche :right x x x)
      (x Point :x 22 :y (TrucMuche-left p)) )
  (TrucMuche-right-length p) )
```

It is of course possible to create a single cyclic value as in:

```
(with-co-instantion
      ((p TrucMuche :right p))
  p )
```

1.7 Duplication

MEROON V3 also offers a duplication mechanism with possible updates. The `duplicate` macro allows you to duplicate an object and to specify at the same time which fields should be updated (compare this with the `clone` function).

(`duplicate` (*form class-name*) [*Macro*]
  : *mono-field-name form*
  : *poly-field-name forms* ...
  ...)

This form computes the value of *form* that will be known as the original. It then creates a fresh object of class *class-name*. The fields of this new object are initialized with the forms specified by the keywords. If a value is not specified with a keyword then it is copied from the original object; an error is raised if the field does not exist in the original object. Note that the class of the duplicated object can be a subclass of the superclass of the original object.

With help of the `duplicate` form, it is therefore possible to clone an object, to clone an object modifying some of its fields, to extend an object into a subclass (provided the specification of the additional fields (explicitly or through their initializers)), to restrict an object into a superclass.

Note that *field*`-length` keywords are not implemented for poly-fields.

Examples

A point named `pt` with coordinates 22 and 44 can be cloned with: `(duplicate (pt Point))`. The duplicated point may be updated on one of its fields as follows.

```
(let ((pt (make-Nicknamed-Colored-Polygon
           11 22 'blue 3 'Joe 'Jack 'Jill )))
  (duplicate (pt Nicknamed-Colored-Polygon)
    :nicknames 'Tina 'Tim ) )
```

The fresh object is blue but has only two nicknames instead of three.

Another more contorted example duplicates a `TrucMuche` to get a `Point`. This is possible, even if they have no field at all in common, if you specify all the fields of Point:

```
(let ((p (instantiate TrucMuche :right #t)))
  (duplicate (p Point)
     :x 11 :y 222 ) )
```

1.8 Modification

An instance may be modified with the `modify` form. This macro allows you to modify a number of (mutable) fields. This is more efficient than using field writers and maybe more readable. The form is quite close from the above `duplicate` form except that it does not create a new object but just modify the given original object.

(modify (*form class-name*) [*Macro*]
: *mono-field-name form*
: *poly-field-name forms* ...
...)

This form computes the value of *form* that will be known as the original. The original is then checked to be an instance of the class whose name is *class-name*. The fields are modified according to the forms specified by the keywords. The fields must exist and they must be mutable. A mutable indexed field may be modified but its length cannot change therefore the *field*`-length` keywords are not offered for poly-fields.

Examples

Here is an example of the modification on two different kinds of fields.

```
(let ((pt (make-Nicknamed-Colored-Polygon
           11 22 'blue 3 'Joe 'Jack 'Jill )))
  (modify (pt Nicknamed-Colored-Polygon)
    :nicknames 'Tina 'Tim 'Tom
    :x 33 ) )
```

One may use a superclass instead rather than the precise class of the instance to modify provided the modified fields are known. The following is an error since the `color` field is not a field of `Point` even if the original is a `ColoredPolygon` with such a field (not mentioning that this `color` field is moreover immutable).

```
(modify ((instantiate ColoredPolygon      ; WRONG
           :x 11 :y 22                    ; WRONG
           :color 'pink )                 ; WRONG
         Point )                          ; WRONG
   :color 'red )                          ; WRONG
```

1.9 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 anomaly is signalled if the index is out of the sequence. An anomaly will be signalled when trying to read the content of a field from an instance of a incorrect class. When a field was defined with `:maybe-uninitialized` and it is actually non yet initialized, then it is an anomaly to try to read it.

Examples

Suppose we still use the previously defined `ncp` instance, 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
```

1.10 Field Writer

The content of fields can be altered with field writers if defined with the `:mutable` field option. 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)
```

1.11 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
```

1.12 Coercer

Any defined class has an associated coercer. The name of the coercer is the name of the associated class prefixed with `->`. By default, the coercer acts as the identity on all direct or indirect instances of the class and signals an anomaly otherwise. The coercer is a generic function that can be tailored to user need.

Examples

If we anticipate on methods and generic functions, here is a coercion by projection. A `Colored-Point` is coerced into a `Point` is it forgets its color.

```
(define-class Colored-Point Point (color))

(define-method (->Point (o Colored-Point))
  (instantiate Point
    :x (Colored-Point-x o)
    :y (Colored-Point-y o) ) )
```

2 Generic Functions

A generic function is a bag of specific functions known as methods. When invoked on a Meroon V3 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 `->Generic`). Like a normal function, the generic function is defined with a list of variables among which only discriminating variables 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 variables. This feature is particularly useful since MEROON V3 is embedded in regular Scheme: a generic function applied on a non-MEROON V3 object invokes the default body. The default default-body is to signal a Domain anomaly since there is no appropriate method.

A discriminating variable can be followed by the name of the class. In that case only methods specializing this class can be defined for this generic function.

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

`(initialize!` *object*`)` This generic function is invoked on any freshly created objects.

`(clone` *object*`)` This generic function copies a MEROON V3 objects. Only the toplevel spineal structure is copied i.e., the copy is shallow, not 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 `#<a` *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 V3 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 V3 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 ((boolean? o) (display (if o "#T" "#F") stream))
          ((null? o) (display "()" stream))
          ((pair? o) (show-list o stream))
          ((vector? o) (show-vector o stream))
          ;;by default, use display
          (else (display o stream)) ) ) )
```

It is possible to specialize this behavior on particular classes of MEROON V3 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 variables are the only variables which must appear, associated to classes in the list of variables.

[3] This code as well as the other excerpts of the sources of MEROON V3 are automatically extracted from the source files of MEROON V3 using a tool (written in Scheme→C) named LiSP2TEX.

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

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

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.

A restriction exists when defining a method on a multi-method generic functions i.e., a generic functions with more than one discriminating variable. It is forbidden to add a method that would bring ambiguity for `(call-next-method)`. This may occur when defining a multi-method for classes $A' \times B$ when there already is a method on $A \times B'$ and A' is a subclass of A and B' is a subclass of B. The method to find on $A' \times B'$ would be ambiguous since there is no reason to consider that it is $A' \times B$ rather than $A \times B'$. And even if there is a method on $A' \times B'$ then `(call-next-method)` would be similarly ambiguous.

Examples

Let us give the methods specifying how Meroon V3 objects as well as Meroon V3 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) ) )
(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) ) )
```

4 Field access

Some general generic functions exist to access fields. They are generally slower than the direct named accessors synthesized at class definition. For all these functions the field may be specified by its name. The object is always the first argument. The field is always the last argument except when it is an indexed field, in which case it is followed by an index. The field may be specified by its name.

This function returns the value of the *field* within *object*. The field must be initialized.

(**field-value** *object field [index]*) [*Function*]

This function allows to modify the content of a mutable field.

`(set-field-value! ` *object value field [index]*`)` [*Function*]

This function allows to initialize an uninitialized field. It is not a pure side-effect (sic) since the field may be immutable (in which case, the field was defined with the `:maybe-uninitialized` field option).

`(initialize-field-value! ` *object value field [index]*`)` [*Function*]

This function allows to return the length of an indexed field.

`(field-length ` *object indexed-field*`)` [*Function*]

This function allows to know if a field is initialized.

`(field-defined ` *object indexed-field [index]*`)` [*Function*]

5 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 V3 object, this can be approximatively checked with `Object?`. Any MEROON V3 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) `->Class` and `->Generic` functions. These functions are the regular coercers for `Class` and `Generic`.

`(->Class ` *symbol*`)` [*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.

`(->Generic` *symbol*`)` [*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. What is returned is the generic object, a MEROON V3 object, corresponding to the name. The generic function itself is the global value of the name.

6 Predicates

Two generic predicates exist to relate objects and classes. This function tests if *object* is a direct or indirect instance of *class*. This function is generic although the implementation only defines a single behavior on `Class`.

`(is-a?` *object class*`)` [*Function*]

This function tests if *class1* is a direct or indirect subclass of *class2*. See the companion paper for the implementation.

`(subclass?` *class1 class1*`)` [*Function*]

---

7 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).

There exists a special kind of anomalies known as warnings of class `Warning` subclass of `Anomaly`. When a warning is produced, the `meroon-error` function is applied on it and the default method just prints it. It is possible to redefine this method to ignore all these warnings.

8 Miscellaneous library

Some 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: CLASS>
  #<Class: HANDY-CLASS>
   #<Class: MEROONV2-CLASS>
 #<Class: APPLYABLE-OBJECT>
  #<Class: GENERIC>
   #<Class: GENERIC-1>
   #<Class: GENERIC-N>
 #<Class: FIELD>
  #<Class: MONO-FIELD>
  #<Class: POLY-FIELD>
 #<Class: DISPATCHER>
  #<Class: IMMEDIATE-DISPATCHER>
  #<Class: SUBCLASS-DISPATCHER>
  #<Class: INDEXED-DISPATCHER>
  #<Class: LINEAR-DISPATCHER>
  #<Class: GLOBAL-DISPATCHER>
  #<Class: TRACING-DISPATCHER>
 #<Class: ANOMALY>
  #<Class: WARNING>
= #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: CLASS>
  #<Class: GENERIC>
  #<Class: MONO-FIELD>
  #<Class: POLY-FIELD>
  #<Class: IMMEDIATE-DISPATCHER>
  #<Class: SUBCLASS-DISPATCHER>
  #<Class: INDEXED-DISPATCHER>
  #<Class: LINEAR-DISPATCHER>
  #<Class: TRACING-DISPATCHER>
 #<Class: ANOMALY>
= #T
```

(`show-methods-for-class` *class-name [stream]]*) [*Function*]

This function shows all the generic functions that define a method for *class*. You can alternatively give a class object instead of a name. By default, *stream* is the default output port. For example,

```
? (SHOW-METHODS-FOR-CLASS 'CLASS)
#<Generic: ->CLASS> has a method for #<Class: CLASS>
#<Generic: IS-A?> has a method for #<Class: CLASS>
#<Generic: ADD-SUBCLASS> has a method for #<Class: CLASS>
#<Generic: GENERATE-ACCESSORS> has a method for #<Class: CLASS>
#<Generic: GENERATE-PREDICATE> has a method for #<Class: CLASS>
#<Generic: GENERATE-MAKER> has a method for #<Class: CLASS>
#<Generic: GENERATE-COERCER> has a method for #<Class: CLASS>
#<Generic: GENERATE-ACCOMPANYING-FUNCTIONS> has a method for #<Class: CLASS>
#<Generic: SHOW> has a method for #<Class: CLASS>
#<Generic: SHOW-MEROON-SIZE> has a method for #<Class: CLASS>
= #unspecified
```

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

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

For example,

```
? (SHOW-MEROON)

(Meroon V3 Ida+2 $Revision: 3.36 $)
Total number of classes: 20
Total number of generic functions: 65
(estimated) internal size of Meroon: 2501 pointers
= #T
```

MEROON V3 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, and more generally any value, in their most inner details even in the case of cyclic structures.

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

The `unveil` function produces a very detailed output and can be tamed if setting appropriately the `*unveil-maximal-indent*` variable, currently set to depth 10.

9 Meta Object Protocol

Conforming to the ObjVlisp rule, classes are first class MEROON V3 objects which can themselves be inspected as any other MEROON V3 objects. Classes are defined as:

```
(define-class Class Object
  ((= name              :immutable) ;0
   (= number            :immutable) ;1
   (= depth             :immutable) ;2
   (= fields            :immutable) ;3
   (= super-number      :immutable) ;4
   (= subclass-numbers  :mutable)   ;5
   (= next              :mutable)   ;6
   (= allocator         :immutable) ;7
   (= immutable?        :immutable) ;8
   (* suprel            :immutable) ;9
   ) )
```

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 more or less evident and some of them are private to the implementation.

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

```
(define-class Field Object
  ((= immutable?        :immutable)
   (= name              :immutable)
   (= class-number      :mutable)
   (= initialized?      :immutable)
   (= initializer       :mutable)
   (* path              :immutable)
   ) )
(define-class Mono-Field Field ())
(define-class Poly-Field Field ())
```

It is a good exercise 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 V3 object described as:

```
(define-class Generic Applyable-Object
  ((= name              :immutable)  ;0
   (= default           :mutable)    ;1
   (= variables         :immutable)  ;2
   (= dispatcher        :mutable)    ;3
   (= top-classes       :immutable)  ;4
   ) )
```

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 `->Generic`. Ideally these two things should be the same object.

When a class is defined with `Class` as metaclass, no accompanying functions are created. The `Handy-Class` metaclass offers you the possibility to define a variety of accompanying functions. The `MeroonV2-Class` metaclass defines the accompanying functions that were usual in the former Meroon. The accompanying functions are generated with the `generate-accompanying-functions` generic function. Attention, metaclasses are used as macros, they generate the code of the accompanying functions not the functions themselves. This function is defined on `Handy-Class` as:

```
(define-method
       (generate-accompanying-functions (class Handy-Class) class-options)
  `(begin
     ,(generate-accessors class class-options)
```

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

```
        ,(generate-predicate class class-options)
        ,(generate-maker class class-options)
        ,(generate-allocator class class-options)
        ,(generate-coercer class class-options) ) )
```

The invoked sub functions are themselves generic.

Examples

`Class` can be subclassed. For instance, a class which knows the number of its direct instances 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 :initializer (lambda () 0))) )
```

Whenever such a class is created, the counter is initialized to zero and the constructor is redefined to count its number of invocations.

```
(define-method (generate-maker (o Counting-Class) class-options)
  (let* ((name (Class-name o))
         (maker-name (symbol-concatenate 'make- name)) )
    `(define ,maker-name
       (let ()
         ,(call-next-method)
         (let ((old ,maker-name)
               (o (->Class ',name)) )
           (lambda args
             (set-Counting-Class-counter!
              o (+ 1 (Counting-Class-counter o)) )
             (apply old args) ) ) ) ) ) )
```

Then we define the intended class:

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

Then we try it:

```
(let ((pt (make-Counted-Point 22 33 44)))
  (list (Counted-Point-x pt) (Point-y pt)
        (Counting-Class-counter (->Class 'Counted-Point)) ) )
  → (22 33 1)
```

Meta Object Protocol is very powerful if understandable.

10 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 for all traced functions.
A more detailed level exist where it is possible to hook specific functions before and after invocation. 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.

11 Last remarks

It is hazardous for your sanity to redefine methods, generic functions and classes. The following are probably true: Whenever you redefine a generic function, all former methods are forgotten. Whenever you redefine a class, it keeps its methods. It also keeps its subclasses (sic) with unpredictable consequences.
Additional informations appear in the README file, the `man` page on MeroonV3 and, of course, in the sources.
There are some implemented features that may disappear without notice. They are there because some may like them. The two first features heavily depend on the macro system and since there is no portable way to know if something is a macro or a special form, these are or not portable or not accurate. We chose the second solution.

`(with-access` *instance* `(`*class-name field-names...*`)` [*Macro*]
forms... `)`

This macro expands its body so that any reference to the variables named *field-name* is replaced by the appropriate field access form. This is true for reference or assignment. For instance,

```
(with-access o (Point y) (set! y (+ y x)))
```

is expanded into:

```
(set-Point-y! o (+ (Point-y o) x))
```

Another way to define methods is `define-handy-method` which automatically wraps its body in the appropriate `with-access` form.

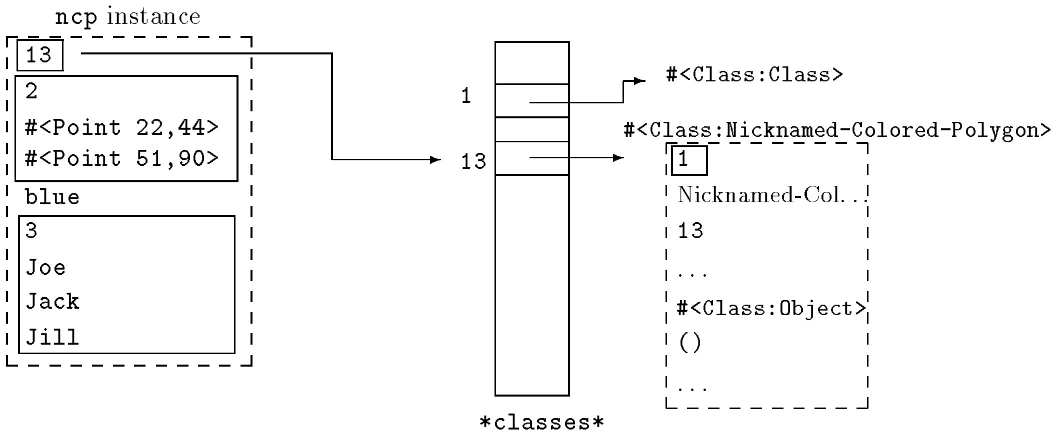

Figure 1: Implementation of an instance

12 Implementation

Meroon V3 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)))
(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 V3 object. The class of classes is indeed defined as:

```
(define-class Class Object
  ((= name              :immutable) ;0
   (= number            :immutable) ;1
   (= depth             :immutable) ;2
   (= fields            :immutable) ;3
   (= super-number      :immutable) ;4
   (= subclass-numbers  :mutable)   ;5
   (= next              :mutable)   ;6
   (= allocator         :immutable) ;7
   (= immutable?        :immutable) ;8
   (* suprel            :immutable) ;9
```

```
) )
```

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 `->Class` function. All fields are grouped in a 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, etc. The superclass of a class can be obtained with `Class-super-class` function. 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 V3 since it breaks the anonymity of generic functions.

13 Conclusions

This document described how to use and how is implemented a small object system called Meroon V3. Meroon V3 offers traditional objects the representation of which is a concatenation of fields. Meroon V3 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 V3 provides an uniform and unique framework: all usual objects of Lisp or Scheme can be defined in Meroon V3 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 V3 uses Meroon V3 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.

[AS85] Harold Abelson and Gerald Jay with Julie Sussman 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*, 23, September 1988. special issue.

[Coi87] Pierre Cointe. The ObjVlisp kernel: a reflexive architecture to define a uniform object oriented system. In P. Maes and D. Nardi, editors, *Workshop on MetaLevel Architectures and Reflection*, Alghiero, Sardinia (Italy), October 1987. North Holland.

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

[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.

[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.

[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.

A Meroon V3 quick reference card

```
(define-generic (generic-name variable | (discriminating-variable [maximal-class-name]) ...)
      [default-body] )

(define-method (generic-name variable | (discriminating-variable class-name) ...)
      ;;(call-next-method) is possible
      body )

(define-class class-name superclass-name
       (field-name
        (= field-name [:maybe-uninitialized] [:(im)mutable] [:initializer (lambda () form)])
        (* field-name [:maybe-uninitialized] [:(im)mutable] [:initializer (lambda (index) form)]) ... )
   [:(im)mutable]
   [:metaclass class-name]
   [:prototype] )

(instantiate class-name
  :Mono-Field-name form
  :Poly-Field-name forms...
  :Poly-field-name-length size ... )
(co-instantiate
  (variable as for instantiate) ... )
(with-co-instantiation
  ((variable as for instantiate) ... )
  forms... )

(class-name? object)
(make-class-name field... length values...)
(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)
(->class-name instance)
```

Predefined generic functions:

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

Utility functions:

```
(object->class object) → class
(is-a? object class)
(subclass? class class)
(->Class symbol) → class
(->Class number) → class
(->Generic symbol) → generic-function
(show-generic-trace [generic-name ...])
(show-generic-untrace [generic-name ...])
(unveil entity [stream])
```