Designing Meroon V3*

Presented at the ECOOP '93 Workshop on Object-Oriented Programming in Lisp.

Christian Queinnec†
École Polytechnique & INRIA-Rocquencourt

The Meroon object system was originally invented for a book to describe implementations of Lisp and Scheme. It was designed to have a pedagogical but efficient implementation, to support separate compilation yet to be powerful enough to unify all the data types of Scheme even vectors and strings without restriction of inheritance.

While designing a distributed extension of Scheme [QD93], new needs appeared that a new release of Meroon tries to satisfy. This paper exposes these problems and how they are solved in Meroon V3. Among these new features are *(i)* compact dispatchers for generic functions, *(ii)* a new initialization protocol for a better control of mutability, *(iii)* a new vision of metaclasses as a code generation mechanism in relation to separate compilation.

The paper first recalls the previous state of Meroon, presents the new needs then their associated solutions.

1 The old Meroon

This section describes the currently distributed version of Meroon [Que91]. Presentation of the related problems and new needs are deferred to the next session.

The previous release of Meroon offered three macros: `define-class` to define classes, `define-generic` to define CLOS-like generic functions and `define-method` to enrich generic functions with mono-methods. In order to allow multiple simultaneous object systems as well as to avoid name clashes (a plea of Scheme), Meroon uses names close but not identical to CLOS ones [BDG+88, KdRB92].

Two kinds of fields (slots) exist: mono- or poly-fields. A mono-field is a location that contains a single value while a poly-field corresponds to a (second class) sequence of locations, the size of which is specified at allocation time. For instance, one can define the class of human beings as a class with a single mono-field containing its name:

```
(define-class Human Object (name))
```

It is possible to specialize this class to contain the nicknames that a human being may receive. Since more than one is possible, we define the class `Nicknamed-Human` with a poly-field to hold the nicknames. A syntax exists to easily define fields within a class definition: a mono-field may be prefixed by an equal sign while a poly-field may be signaled by a leading star.

```
(define-class Nicknamed-Human Human ((* nickname)))
```

These two classes can be similarly defined in Smalltalk [GR83] but Meroon allows to specialize further this class. We can for instance add two new fields to hold an age and a sequence of friends as in:

```
(define-class Friendly-Nicknamed-Human Nicknamed-Human
  ((= age) (* friend)) )
```

**Revision* : 1.8, printed on October 28, 1993 at 20:17.

†Laboratoire d'Informatique de l'École Polytechnique (URA 1437), 91128 Palaiseau Cedex, France – Email: `queinnec@polytechnique.fr` This work has been partially funded by GDR-PRC de Programmation du CNRS.

While classes are defined, the usual accompanying functions are created:

- predicates like `Nicknamed-Human?`,
- field readers which have a signature depending on the nature of the field: `Human-name` returns the name of a (direct or indirect) instance of `Human`; `Friendly-Nicknamed-Human-friend` takes a (direct or indirect) instance of the `Friendly-Nicknamed-Human` class as well as an index i and return the i^{th} friend of this instance.
- field writers, as field readers, have a signature that depends on the nature of the field, for instance, `set-Nicknamed-Human-nickname!` or `set-Friendly-Nicknamed-Human-age!`.
- field lengthers are unary functions that return the length of a poly-field within an object. An example is `Nicknamed-Human-nickname-length`.
- allocators such as `allocate-Nicknamed-Human` that take as many natural numbers as there are poly-fields in the mentioned class to specify their sizes. Sizes are specified along the order of poly-fields. The form `(allocate-Friendly-Nicknamed-Human 2 0)` allocates an instance of `Friendly-Nicknamed-Human` with two possible nicknames and no friends at all.
- constructors, for instance, `make-Friendly-Nicknamed-Human` which take values and build instances. Values are specified along the order of fields. Values for a poly-field are prefixed by their number. One can build an instance with two nicknames and no friends at all as ther result of the form: `(make-Friendly-Nicknamed-Human "Meroon" 2 'teddy 'bear 12 0)`.

MEROON offers single inheritance which is the sole inheritance mechanism I understand.

Generic functions are defined by `define-generic`. All the previous accessors (readers, writers or lengthers) are not generic. The definition of a generic function specifies what is the (unique) discriminating variable. It is also possible to define in the same time the behavior of the generic function when applied on a non-MEROON object. Thus there might be a default behavior on the top class `Object` and another one for non-MEROON objects. We refused to provide equivalent MEROON classes for native Scheme types since it would be difficult and unefficient to specialize them in portable Scheme. Just consider for instance how you can specialize dotted pairs and expect `map` (or `apply`) to work on them.

Here is a simple example of a generic function that takes a mandatory argument (the variable on which dispatch is done, `self`) and other optional arguments gathered in a list, value of `args`:

```
(define-generic (best-friend (self) . args) #f)
```

Methods can be added to a generic function if they have a congruent list of variables and specify the class on which the generic function is specialized. As in CLOS, `call-next-method` can be used to obtain the method that would have been invoked had the current one not been there. Contrarily to CLOS, `call-next-method` does not accept arguments in order to leave the receiver invariant.

```
(define-method (best-friend (self Object) . args) self)
(define-method (best-friend (self Friendly-Nicknamed-Human) . args)
  (if (> (Friendly-Nicknamed-Human-friend-length self) 0)
      (Friendly-Nicknamed-Human-friend self 0)
      (call-next-method) ) )
```

MEROON describes large parts of itself in MEROON. Following the tradition of ObjVlisp [Coi87], classes are objects whose class is `Class`. Among the fields of a class are its name, fields, superclass as well as the associate predicate, allocator and maker functions. Fields are themselves described by instances of class `Field` which contain their name, the class that introduced them and their associate reader function. Class `Field` is further specialized into `Mono-Field` and `Poly-Field` classes which are themselves specialized into `Mutable-Mono-Field` and `Mutable-Poly-Field`.

MEROON runs under various classical i.e. toplevel-based, implementations of Scheme but also under Scheme–>C [Bar89] which offers a separate compilation facility. For that goal, a keyword named `:prototype` was allowed in a `define-class` form so that, at macroexpansion-time, a class can be subclassed but not created again at load-time. More on this later.

This release of MEROON was successfully distributed through anonymous `ftp` for nearly two years. You can get it from `ftp.inria.fr` (IP number 192.93.2.54) under name `INRIA/icsla/meroon.tar.Z`. Some features like poly-fields seem to be appreciated because they combined the compactness of vectors and the versatility of objects. MEROON allows to define vector- or string-like objects and specify methods on them thus making all data types of Scheme regular objects of a unified framework.

2 Problems

This section presents various problems encountered while implementing or using MEROON. Problems only related to the integration of MEROON and portable Scheme, such as merging generic functions and instances of class `Generic` or the possibility to write a portable `with-fields` (`with-slots` in CLOS) syntax, are not discussed in this paper. In all that section, the word "MEROON" refers to the old MEROON.

2.1 Linear-time class membership

The `is-a?` predicate tests whether an object belongs or not to a class. This predicate must be very efficiently implemented since it is at the heart of the dispatching mechanism. It was implemented in the old MEROON as a linear test climbing up the tree of classes. This is not so laughably bad since (roughly) half of the time, the given object is a direct instance of the given class and this is very efficiently checked. Another reason is that although there is a constant-time implementation of this test, its cost is greater due to the numerous redundant dynamic type-checks that cannot be removed since MEROON is not native but only written in portable Scheme.

2.2 Sparse dispatch vectors

The implementation associates to each class a unique natural number. Generic functions have an internal state containing a vector mapping these class numbers towards the methods they provide for these classes. Apart some rare functions like `show` (`print-object` in CLOS) that provide methods for most of the classes, it is often the case that generic functions usually provide a default behavior on one class, specialize it on its subclasses and leave it undefined elsewhere. Therefore there is an opportunity to compact dispatchers for a subtree of the inheritance tree.

2.3 Mutability control

The `eqv?` and `equal?` predicates correspond to two different semantics. The first roughly addresses substitutability while the second is more user-oriented i.e. can be recursive or not, can take care of cycles or not, can be tailored by the user etc. When distributing objects over several computers [QD93], it is important to preserve the concept of a unique value space. Whenever a process mutates an object, all causally dependent computations must be aware of this mutation. It is therefore important to know the mutability of fields. Two objects are substitutable if there does not exist a program that differentiates them. Our distributed extension of Scheme offers substitutability as a built-in predicate called, after [Bak90], `egal`. We approximate substitutability by the following two rules. Two immutable objects (whose fields are all immutable) of the same class are `egal` if all their fields are `egal`. Two mutable objects of the same class (which have at least one mutable field) are `egal` if they correspond to a same physical object. Migrating mutable objects is therefore complex since their mutation must be controlled by some sort of coherency protocol.

2.4 Co-instantiation

Mutability can be specified in MEROON for any field description if adding the keyword `:mutable`, another keyword also exists: `:immutable`. In the latter case, no field writer is generated and an appropriate field descriptor is built. Although rare, it is necessary to provide the ability to build cycles through objects i.e. to intantiate in the same time two or more objects mutually referent. MEROON needs them for its self-description since, for instance, classes refer to their superclass which holds the list of all their subclasses. To build these cycles requires mutability of fields but, once created, they are very often never mutated again. It would be useful to design a better solution for objects that are just mutated once.

2.5 Initialization versus mutation

Another aspect of the mutability problem concerns the creation of objects. In MEROON, objects can be allocated (using `allocate-`) then initialized therefore requiring mutability of fields. Another way to build them is to provide the content of all their fields (using `make-`) at instantiation time. This last solution has two defaults: *(i)* it is not possible to build direct cycles (this requires indirection through promises or equivalent) *(ii)* the order of arguments depends on the order of fields in the class definition making harder to memorize or to change it. Using keywords and default values seems attractive. All techniques based on progressive initializations with side-effects seem to be condemned.

2.6 Integrity

Metaclasses have definite advantages. They allow a precise self-description of the object system and thus favor writing *meta-methods*. For instance, the `egal` predicate mentioned above is a simple function that walks the list of field descriptors of the class of the two compared objects to inquire their mutability. The migration facility that takes objects from one machine and copies them on a remote computer is another instance of a meta-method that uses informations on the structure of objects that can be found in the class.

Even this simple level of self-description poses problems and mainly a problem of integrity. MEROON needs itself to run and makes some assumptions on the precise state of some instances that precisely control some internal aspect. To mutate them without obeying the meta-object protocol is easy and error-prone. For instance, the `subclasses` field within classes is mutable since new classes can be created. To leave it mutable means that it is possible to obtain, from its field descriptor, a field writer function then to apply it over a class and ruin this whole class subtree.

2.7 Naming accessors

The `define-class` macro, a macro with an internal state that cannot be written with the macro proposal of R^4RS [CR91], expands a class definition into a sequence of accompanying functions as well as some code to create and register the necessary meta objects: the class itself and its field descriptors. To define a class in MEROON imposes to know parts of the super class since accessors to inherited fields have a name which requires the name of the inherited fields; efficient code generation for accessors or allocators also require to know how many poly-fields there are in the super class.

MEROON's accessors are named *class-field*. Although simple to memorize since all accessor names are regularly built, this consumes a lot of names (symbols and global variables). For instance, when defining the class `Friendly-Nicknamed-Human`, MEROON must also define the inherited accessor function `Friendly-Nicknamed-Human-name`. As a consequence of this naming convention, observe that this function is not exactly similar to `Nicknamed-Human-name` although they coincide for instances of `Friendly-Nicknamed-Human`; they differ on direct instances of `Nicknamed-Human` since it is naturally an error to apply `Friendly-Nicknamed-Human-name` on them since they are not instances of `Friendly-Nicknamed-Human`.

2.8 Evaluation at macroexpansion-time

When compiling a file, it is often the case that a bunch of classes are defined together so a class definition must also register the class (at macroexpansion-time) so that subclasses may be defined further in the file. To build the meta-objects describing a class at macroexpansion-time is difficult in presence of poly-fields since their size may not be statically known. Suppose we subclass `Class` into `Memo-Class`, a metaclass for classes that want to remember the last n created instances. MEROON offers the possibility to specify the metaclass of a `Class` instance with the `:metaclass` keyword.

```
(define-class Memo-Class Class ((* last)))
```

An example of a memo class is the following:

```
(define-class Memo-Point Object (x y)
   :metaclass Memo-class *remembered-points-number* )
```

The class Memo-Point has an additional poly-field that can hold some values. The size of this poly-field is stated by a global variable, *remembered-points-number*, which belongs to the file to be compiled and which is unknown till load-time. The problem is that metaclasses gives the possibility of unrestricted computation when defining classes which in turn yields problems with compilation.

The definition of the accompanying functions appear in the macroexpanded definition of a class. This means that they cannot be used at macroexpansion-time but to use eval to extend the macroexpansion global environment which is not portable Scheme.

Separate compilation also brings some troubles. To define a class in a file requires to know its super-class and, if using a powerful meta object protocol, to know the specialization of this protocol on the intended metaclass. This is clearly difficult in portable Scheme. We only let the macroexpansion know of the structure of a class by adding the :prototype keyword to allow it to be subclassed. Such a prototype class definition is expanded into some code to check whether at load-time the prototype definition conforms to the real definition. We know of no module proposals for Scheme that handles these problems apart [QP91].

This ends the list of problems for which we want to present some ideas or solutions we implemented in Meroon V3. Some of these problems may be solved by defining a new Scheme-based language where define-class would be a real special form. It would also ease the integration of Scheme and Meroon to allow, for instance, funcallable objects and specialization of predefined types such as pairs, ports etc.

3 Meroon V3

This section presents and justifies the solutions of some of the previously exposed problems. They are implemented in MeroonV3.

3.1 Constant-time is-a?

The present idea is due to Luis Mateu and strongly depends on single inheritance. A class object holds a mono-field containing its depth in the tree of classes as well a poly-field containing all its ancestors from Object to itself. Observe that the depth is nothing but the length of the ancestor poly-field. To check if a class C_i is a subclass of C_j, it is sufficient to check whether C_j appears as the $depth(C_j)$ ancestor of C_i where $depth(C_j)$ is the number of ancestors of C_j.

The precise definition of this predicate supposes that Class has an ancestor poly-field. Due to the integration with Scheme, the value of o is checked to be a Meroon object. In the following definition of is-a?, object->class is Meroon equivalent of class-of in CLOS.

```
(define (is-a? o class)
  (and (Object? o)
       (let ((direct-class (object->class o)))
         (let ((cn (Class-number class))
               (depth (Class-ancestor-length class)) )
           (and (<= depth (Class-ancestor-length direct-class))
                (= cn (Class-ancestor direct-class (- depth 1))) ) ) ) ) )
```

This predicate cannot be efficiently compiled in portable Scheme due to the poor integration of Scheme and Meroon. The reason is that objects are represented by vectors and any call to functions dealing with vectors dynamically check that the first argument is a vector, the second one (the index) is a natural number not so great as to exceed the size of the vector. Furthermore, all Meroon accessors check the class of their argument. All these tests are redundant and useless since this code is expected to be safe. Therefore and although bounded in time, this predicate is slower compared to the old predicate which only handles references.

```
(define (meroon-v2-is-a? o class)
  (and (Object? o)
```

```
(let up ((direct-class (object->class o)))
  (or (eq? class direct-class)
      (let ((sc (Class-super direct-class)))
        ;#f is the super of Object
        (and sc (up sc)) ) ) ) ) )
```

The solution is therefore to integrate more tightly the object system with the language, a lesson already taught by the CLOS designers group.

3.2 Compacting dispatchers

A generic function is represented in Scheme by a regular function that closes an instance of `Generic` so that modifying the instance has an immediate effect on the function. Rather than recording all methods in a dispatch vector whose length is the total number of classes leading to a waste of room, we adopt the following encoding for mono-generic functions i.e. generic functions that dispatch on a single variable (MEROON V3 also offers multi-methods but since these are rare, no special care was taken to implement them). Dispatchers are themselves MEROON instances inheriting from the `Dispatcher` class. A dispatcher can be an instance of `Immediate-Dispatcher` that imposes the method to use independently of the class on which to dispatch; this dispatcher is mainly used to record the default method. Instances of `Subclass-Dispatcher` act as an if-then-else construction with respect to a given class: if the receiver belongs to that precise class then the dispatching continues with the then part otherwise the else part is used. The last type of dispatchers, instances of `Indexed-Dispatcher`, refine the previous one and correspond to the dispatch vectors of the old MEROON: an indexed dispatcher for a given class holds all the methods for all its subclasses including itself. Methods are retrieved from indexed dispatchers through indexing as explained below.

We did not change the representation of instances used in the old MEROON: they are represented by vectors. All classes are associated to a unique natural number, instances store this number in their first coordinate: this is the instantiation link.

To be compact, the tree of classes is numbered in prefix order, see the example on the left of figure 1 where class names are followed by their associated number and followed (between parentheses) by the minimal and maximal class numbers of all their direct or indirect subclasses (including themselves). This has two advantages: first, it is possible to test class membership in constant time by a simple arithmetic interval membership. Second, for any subclass of C, it is possible to record an associated method in a dispatch vector (an instance of `Indexed-Dispatcher`) whose length is the number of subclasses of C. The index to use is simply the difference of the number of the subclass with the number associated to C. The generic function `find-method` describes more precisely these behaviors.

The following methods define how to find the method associated to a class number `cn`, the number associated to the class of the receiver. To speed up subclass membership tests, indexed- and subclass-dispatchers hold a copy of the minimal and maximal numbers associated to a class subtree.

```
(define-method (find-method (d Immediate-Dispatcher) cn)
  (Immediate-Dispatcher-method d) )
(define-method (find-method (d Subclass-Dispatcher) cn)
  (if (and (>= cn (Subclass-Dispatcher-min d))
           (<= cn (Subclass-Dispatcher-max d)) )
      (find-method (Subclass-Dispatcher-yes d) cn)
      (find-method (Subclass-Dispatcher-no d) cn) ) )
(define-method (find-method (d Indexed-Dispatcher) cn)
  (if (and (>= cn (Indexed-Dispatcher-min d))
           (<= cn (Indexed-Dispatcher-max d)) )
      (Indexed-Dispatcher-method d (- cn (Indexed-Dispatcher-min d)))
      (find-method (Indexed-Dispatcher-no d) cn) ) )
```

With such an organization, dispatchers are compact. Consider for example figure 1 which exhibits the representation of a dispatcher for a generic function which concentrates its methods on subclasses of B. This technique is static and departs from the cache-based technique of [KR90].

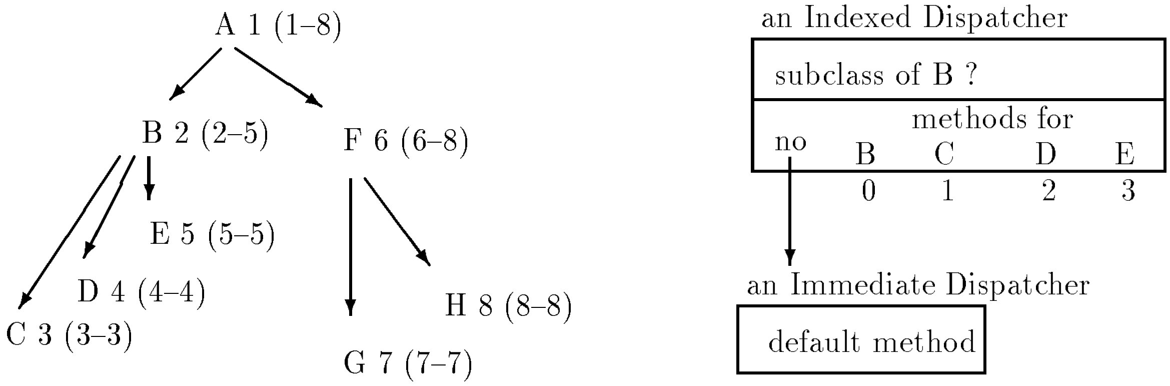

Figure 1: A generic function on B and its subclasses

3.3 Renumbering classes

The problem with classes numbered in prefix order is how to accommodate dynamic creation of classes. This is a problem (even if this facility is forbidden) since MEROON is written in portable Scheme, ignores the exact order along which classes are defined, and does not suppose existence of an intelligent linker. Back to figure 1, it is clear that adding subclasses to the rightmost classes (A, F or H) of the inheritance tree is painless since the basic requirements for the numbering is that all classes must have subclasses with non-overlapping min-max intervals. We propose, for the other cases, to renumber classes. Were we to define a new subclass, say I, of a class, say D, the key is to move to the right all the classes that have an ancestor to the left of D so that D is moved to a rightmost position. This algorithm leaves part of the inheritance tree in place and as such does not waste numbers inconsiderately. Figure 2 represents the renumbered tree of classes after adding I, a subclass of D. Note that this is still a prefix order renumbering but for the root class A.

To renumber classes may also affect the instantiation link of instances. To avoid scanning the whole heap to update these numbers, we adopt the following trick. Observe that whenever classes are renumbered, no class number is ever reused. Whenever a generic function is applied and if no method is found, then the instantiation link is updated and a new method lookup is started if needed. This penalty is only paid once for objects. Of course, a native MEROON implementation might use the GC for part of that task.

To renumber classes imposes to scan the inheritance tree and all generic functions to update the min and max subclass numbers. It respects the structure of dispatchers except for some indexed dispatchers that needs to be enlarged to accomodate the freshly created classes. Renumbering makes class definition expensive but renumbering does not occur so frequently with separate compilation except when initially loading the compiled files.

Another subtle problem is that during renumbering and since class numbers are incrementally changed, the tree of meta objects is not in a stable state and this prevents the normal generic function dispatching to be used. Actually, renumbering is done by regular functions, not by generic function; this is a nuisance when extending the meta object protocol.

3.4 Accessor names

The obvious solution for names of accessors not to depend on the super class is to name them simply after the field name. Thus to access the `name` field of a direct or indirect instance of `Human`, one just writes `(name` *o*`)`[1]. This avoids the definition of `Human-name`, `Nicknamed-Human-name` etc. To adopt this view means that accessors are sort of generic functions which are side-effected by class definitions to hold new methods. If accessors are regular generic functions then the user has the right to add personal methods to them with the

[1]or, for example, `(->name` *o*`)` as in [RM92].

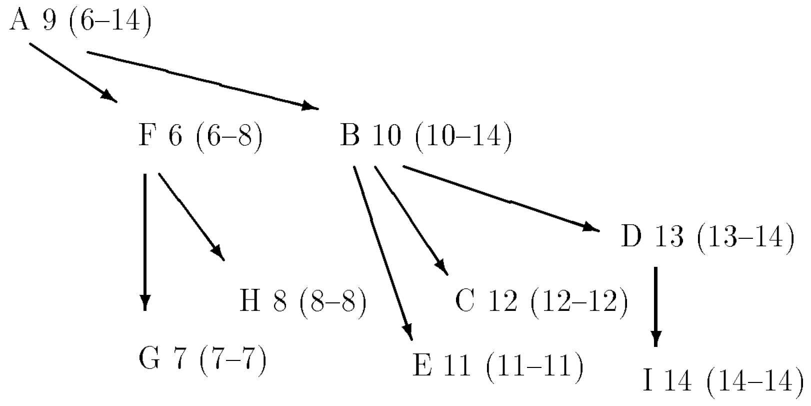

Figure 2: After adding class I as a subclass of D

potential to break all static optimizations on field access[2].

Another view is to consider generic functions as immutable objects. Adding a method to a generic function yields a new enriched generic function. But to attach this new generic function to the variable through which it was known, this variable has to be mutable. In other words:

> `(define-method (`*generic* ...`) ...)`
> *is somewhat like*
> `(set!` *generic* `(add-method` *generic* ...`))`
> *rather than*
> `(add-method!` *generic* ...`)`

But at that time, if the *generic* global variable is mutable then we cannot ensure anything on its value and we are back to our integrity problem. A solution is that the *generic* global variable should be an immutable global variable (to ease compilation) bound to a value with good compilation invariant: an accessor is close to a syntax setup by `define-class` that the user cannot directly alter. Accessors offered by MEROON V3 are named `get-`*field* and `set-`*field*`!` and belongs to a particular class of generic functions that the user cannot extend by its own methods. This saves a lot of symbols but still leaves a problem: the signature of these accessors is not static since there might be a mono- and a poly- field with a same name. In this case, the appropriate methods on the accessors do not have the same signature since the accessors to the poly-field take a supplementary index. This imposes a small overhead at run-time since the arity has to be checked. Of course some type recovery analysis [Shi90] may be performed to alleviate this impediment.

3.5 Allocation

MEROON V3 offers a macro (ideally a special form) for instantiation rather than a function (`make-instance` in CLOS). Our reasons are *(i)* the absence of keywords in Scheme, *(ii)* our incompatible use of keywords (keywords are always followed by a value in COMMON LISP, whereas they prefix a sequence of values in MEROON), *(iii)* our search for efficient allocation (excluding a keyword mechanism parsed at run-time) and, *(iv)* the desire to know every site of allocation (to control the initialization of fields, more on this later). An instantiation form has the following syntax:

> `(instantiate` *class* {`:`*keyword form*...} ... `)`

As usual, keywords are represented by symbols prefixed by a colon. They can name a mono-field and then are followed by a value to fill this field. They can also name a poly-field and then are followed by as many values as needed to fill this poly-field. Finally they can name a poly-field with a `-length` suffix in

[2]Remember that we are in portable Scheme in which no declaration exist to compensate these bad effects. Moreover we personally dislike declarations that lack a clear semantical status.

which case, only the length of the poly-field is specified. All these keywords allow to simulate the regular allocators of Scheme such as `vector`, `make-vector` etc.

When defining a class, each field may specify how it will be initialized if not mentioned explicitly in an instantiation form. The initialization is specified as a thunk for a mono-field and, for a poly-field, as a unary function taking an index. For example:

```
(define-class Friendly-Nicknamed-Human Nicknamed-Human
  ((= age    :initializer (lambda () 99))
   (* friend :initializer (lambda (i) (format #f "Friend#~A" i)))
   ) )
```

This yields the possibility of uninitialized fields i.e. fields lacking initializers in their definition and not mentioned in an instantiation form. This notion already exists in CLOS with `slot-boundp` and related functions. In fact, uninitialized fields are the solution to more than one of our problems. MEROON offers several keywords for the definition of a field.

- `:mutable` means that it is possible to modify the content of the field via the `set-`*class-field*`!` function or the generic mutator `set-field-value!`.
- `:immutable` means that, once initialized, the content of the field is never modified. Of course, `:mutable` and `:immutable` are exclusive.
- `:initializer` specifies how to initialize a field if not mentioned in an instantiation form. This keyword is not mandatory.
- `:never-uninitialized` qualifies a field as always containing a meaningful value. Of course, when an initializer is provided the `:never-uninitialized` holds implicitly.

General generic functions exist to access fields of instances. All these generic functions can work on mono- or poly- fields if given an additional index. The `field-value` function (`slot-value` in CLOS) takes an instance and a field and extract the corresponding value out of that instance; this extraction checks if the field is initialized otherwise it signals an error. The `set-field-value!` generic function modifies a field of an instance to hold a new value; this field must be mutable. The `initialize-field-value!` generic function initializes a field of an instance with some value; this field must not be already initialized otherwise an error is raised.

3.6 Integrity

We did not solve elegantly the integrity problem. The only feature we provided in MEROON V3 is to qualify some fields of meta objects as private. The user knows of their existence, name, position in the instance representation but cannot read (resp. write) them nor synthetize a field reader (resp. writer) for them. For example, the user cannot read (nor change) the number that is associated to a class.

Unfortunately, this concept is private to the implementation and the user cannot take benefit of it. To sum up, it is possible when defining fields to specify whether they are mutable and always initialized or not.

3.7 Co-instantiation

With the possibility of initialization, it is possible to build cycles without mutation. For example, this code might appear in a toy compiler for Scheme: `e` is a `lambda`-Sexpression which is converted into an instance of `Abstraction` referencing its variables (instances of `LexicalVariable`) which themselves refer to it.

```
(let ((fun (instantiate Abstraction
              :variable-length (length (cadr e)) ;a poly-field for all the variables
              :body (cddr e) )))
  (do ((i 0 (+ 1 i))
       (vars (cadr e) (cdr vars)) )
      ((null? vars) fun)
   (initialize-field-value!
```

```
    fun (instantiate LexicalVariable :name (car vars) :binder fun)
        'variable         ;the name variable is coerced into the adequate Field instance
        i ) ) )
```

One interest concerning efficiency of the `:never-uninitialized` keyword is that such a field can always be read without checking first that it holds a meaningful value since this is ensured by construction vof the object. Such a field either possesses an explicit initializer or is guaranteed to hold a value since all instantiation sites are known and van verify this fact. For the above example, the `variable` poly-field of `Astraction` being initialized after the creation of the object cannot be qualified with `:never-uninitialized`. Accesses to this field always have to check if the content of the field is meaningful.

In a distributed and/or concurrent setting, an immutable but non-necessarily initialized field mimics the I-structure of data-flow machines [ABU91] and allows some sort of transparent concurrency. It is a simple matter to stop a process that wants to read an uninitialized field and to delay it until the field is filled.

3.8 Metaclass as a code generation mechanism

We are interested to compile files that contain class definitions. To compile such a file means that class definitions must be turned into some code that will install the specified class when the file is loaded. A class definition must then be considered as a kind of macro form that generates some code. As in the old Meroon, we choose the following approach. A `define-class` form is parsed and builds a (direct or indirect) instance of `Class`. This class is immediately registered into the class hierarchy so it can be used for any operation that does not need its accompanying functions: one can instantiate it, access its fields via the generic `field-value` function and others and define new subclasses to specialize it.

A generic function named `generate-class-definition` exists to turn a class into some appropriate code. The method associated to `Class` generates some code that will recreate and register at load-time the class instance with all its depending fields. By default, classes created by a `define-class` form, have the `Handy-Class` metaclass that generates a predicate, a maker, a coercer as well as some accessors as accompanying functions. These methods are:

```
(define-method (generate-class-definition (c Class) class-options)
   (generate-class-registration c class-options) )
(define-method (generate-class-definition (c Handy-Class) class-options)
   `(begin ,(call-next-method)
           ,(generate-predicate c class-options)
           ,(generate-accessors c class-options)
           ,(generate-maker c class-options)
           ,(generate-coercer c class-options) ) )
```

In the latter method, the `generate-`*something* functions also are generic functions. A list of class options is also given to these functions so they can be further parameterized. All these generic functions can be specialized for new subclasses of `Handy-Class`.

To use a `define-class` form is not the only mean to define a class, a class can also be built, at macroexpansion-time, by whatever computation provided it is finally turned into some expanded code via the `generate-class-definition` generic function.

Let us complete our previously sketched example of the `Memo-Class` metaclass. First, we define it as a subclass of `Handy-Class`. Second, we specialize `generate-maker` to memorize the last created instances: we do it by specifying a particular `initialize!` method to register them (the `initialize!` generic function is guaranteed to be called on every allocated instance).

```
(define-class Memo-Class Handy-Class
  ((* last :mutable :initializer (lambda (i) #f))) )
(define-method (generate-maker (mc Memo-Class) class-options)
  `(begin
     ,(call-next-method)
     (let* ((c (->Class ',(get-name mc)))
            (number (last-length c)) )
       (define-method (initialize! (o ,(get-name mc)))
```

```
          ;;shifts all remembered instances by one position
          (do ((i 1 (+ 1 i)))
              ((= i number))
            (set-last! c (- i 1) (get-last c i)) )
          ;;remembers the fresh instance in the last position
          (set-last! c (- number 1) o)
          o ) ) ) )
(define-class Memo-Point Object (x y)
   :metaclass Memo-Class 3 )
```

In the previous definition, the `->Class` coercer takes a symbol and returns the class that has this name (`find-class` in CLOS). Pay attention to the fact that `(->Class ',(get-name mc))` is not tautological since `mc` is converted to a symbol at macroexpansion-time while this symbol is coerced back to a class stored in `c` at load-time. This illustrates some of the problems of compilation. Note that all direct or indirect instances of `Memo-Point` will be remembered in the additional fields of the `Memo-Point` meta-object even if some subclasses of `Memo-Point` do not have `Memo-Class` as metaclass.

Let us give an alternate definition of `Memo-Class` where we want to the memorization to be only triggered through the `make-Memo-Point` allocator. The key is again to change the generation of the maker function but to use the original maker to define the enhanced one. This code will not use the `instantiate!` generic function.

```
(define-class Memo-Class Handy-Class
  ((* last :mutable :initializer (lambda (i) #f))) )
(define-method (generate-maker (mc Memo-Class) class-options)
  (let ((maker-name (symbol-append 'make- (get-name mc))))
    `(define ,maker-name
       (let* ((c (->Class ',(get-name mc)))
              (number (last-length c)) )
         ,(call-next-method)   ;an internal definition
         (lambda args
           (let ((o (apply ,maker-name args)))
             (do ((i 1 (+ 1 i)))
                 ((= i number))
               (set-last! c (- i 1) (get-last c i)) )
             (set-last! c (- number 1) o)
             o ) ) ) ) ) )
(define-class Memo-Point Object (x y)
   :metaclass Memo-Class 3 )
```

With this new example only invokers of `make-Memo-Point` will record the three last created direct instances of `Memo-Point`.

Our view of metaclasses is more static than in other object systems since they more clearly separate class-definition-time from run-time. This view makes usage of metaclasses more akin to macroexpansion which is a relatively simple model that facilitates control over separate compilation.

4 Conclusions

We proposed in this paper some ideas as well as a rationale for the design of MEROON V3. Many of these new features are related to a distributed extension of Scheme where control over mutability is of paramount importance. Some other features are related to separate compilation even if they do not solve all these problems. More work is needed to reconcile metaclasses and separate compilation.

Acknowledgments

Thanks to Pierre Cointe, Luis Mateu and Lance Norskog for their conversation, implementation ideas and Meroon feedback during the maturation of this new release.

Bibliography

[ABU91] Arvind, Lubomic Bic, and Theo Ungerer. Evolution of data-flow computers. In Jean-Luc Gaudiot and Lubomic Bic, editors, *Advanced Topics in Data-Flow Computing*, chapter 1, pages 3–34. Prentice-Hall, 1991.

[Bak90] Henry G Baker. Equal rights for functional objects or, the more things change, the more they are the same. Technical report, Nimble Computer Corporation, 16231 Meadow Ridge Way, Encino, CA 91436, USA, October 1990.

[Bar89] Joel F Bartlett. Scheme–>C, a portable Scheme-to-C compiler. Research Report 1, Digital Western Research Laboratory, Palo Alto, Cal., January 1989.

[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:special issue, September 1988.

[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.

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

[GR83] Adèle Goldberg and David Robson. *Smalltalk-80, The Language and its Implementation.* Addison Wesley, 1983.

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

[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.

[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.

[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 ??–??, Boston (Massachussetts US), October 1993.

[QP91] Christian Queinnec and Julian Padget. Modules, Macros and Lisp. In *Eleventh International Conference of the Chilean Computer Science Society*, pages 111–123, Santiago (Chile), October 1991. Plenum Publishing Corporation, New York NY (USA).

[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.

[Que91] Christian Queinnec. Meroon: A small, efficient and enhanced object system. Technical Report LIX.RR.92.14, École Polytechnique, Palaiseau Cedex, France, November 1991.

[RM92] John R Rose and Hans Muller. Integrating the scheme and c languages. In *LFP '92 – ACM Symposium on Lisp and Functional Programming*, pages 247–259, San Francisco (California USA), June 1992. ACM Press. Lisp Pointers V(1).

[Shi90] Olin Shivers. Data-flow analysis and type recovery in scheme. Technical Report CMU-CS-90-115, CMU School of Computer Science, Pittsburgh, Penn., March 1990.