CLOX: Common Lisp Objects for XEmacs

Didier Verna
(Epita Research and Development Laboratory, Paris, France
didier@xemacs.org)

Abstract ***CLOX*** is an ongoing attempt to provide a full Emacs Lisp implementation of the Common Lisp Object System, including its underlying meta-object protocol, for XEmacs. This paper describes the early development stages of this project. ***CLOX*** currently consists in a port of Closette to Emacs Lisp, with some additional features, most notably, a deeper integration between types and classes and a comprehensive test suite. All these aspects are described in the paper, and we also provide a feature comparison with an alternative project called Eieio.

Key Words: Lisp, Object Orientation, Meta-Object Protocol
Category: D.1.5, D.3.3

1 Introduction

***Note:** the author is one of the core maintainers of XEmacs. In this paper, the term Emacs is used in a generic way to denote all flavors of the editor. If a distinction needs to be made, we use either GNU Emacs or XEmacs where appropriate.*

1.1 Context

The XEmacs[1] project started almost twenty years ago after a split from GNU Emacs. This is perhaps the most popular "fork" in the whole free software history. Nowadays, the codebase of XEmacs consists of roughly 400,000 lines of C code [C, 1999] (including the implementation of the Lisp engine) and 200,000 lines of Emacs Lisp. In addition to that, the so-called "Sumo" packages, a collection of widely used third-party libraries distributed separately, amounts to more than 1,700,000 lines of Emacs Lisp code.

Over the years, XEmacs has considerably diverged from GNU Emacs. While the XEmacs development team tries to maintain compatibility for the common functions of the Lisp interface, some features, also accessible at the Lisp level, are exclusive to XEmacs (*extents* and *specifiers* are two common examples). Because of these differences between both editors, writing truly portable code is not easy. In some cases, the existence of compatibility layers makes the task a bit easier. For instance, the implementation of *overlays* (logical parts of text with local

[1] `http://www.xemacs.org`

properties) in XEmacs is in fact designed to wrap around *extents*, our native and more or less equivalent feature.

There is one place, however, in which compatibility with GNU Emacs is neither required nor a problem: the implementation of the editor itself, including its Lisp dialect. After almost twenty years of independent development, it is safe to say that the internals of XEmacs have very little left in common with the original codebase, let alone with the current version of GNU Emacs.

One thing that should immediately strike a newcomer to the internals of XEmacs is the very high level of abstraction of its design. For instance, many editor-specific concepts are available at the Lisp layer: windows, buffers, markers, faces, processes, charsets *etc.*. In XEmacs, every single one of these concepts is implemented as an opaque Lisp type with a well-defined interface to manipulate it. In the current development version, there is 111 such types, 35 of which are visible at the Lisp level (the rest being used only internally).

The other important point here is that although the core of the editor is written in C, there is a lot of infrastructure for data abstraction and sometimes even for object orientation at this level as well. The examples given below should clarify this.

Polymorphism. Many data structures in XEmacs provide a rudimentary form of polymorphism and class-like abstraction. For instance, the `console` type is a data structure containing general data members, but also a type flag indicating which kind of console it is (X11, Gtk, tty *etc.*), and a pointer to a set of type-specific console data. Each console type also comes with a set of type-specific methods (in a structure of function pointers) . This provides a form of polymorphism similar to that of record-based object systems [Cardelli, 1988] in which methods belong to classes.

Accessors. Instead of accessing structure members directly, every data type comes with a set of pre-processor macros that abstract away the underlying implementation. For instance, the macro `CONSOLE_NAME` returns the name of the console, whatever the underlying implementation. This might be considered as more typical of data abstraction in general than object-orientation though.

Dynamic method lookup. There is even an embryonic form of dynamic method lookup which looks very much like Objective-C's informal protocols, or (as of version 2.0 of the language), formal protocols with optional methods [Apple, 2009]. For instance, in order to "mark" a console, without knowing if it even makes sense for that particular console type, one would *try* to call the `mark_console` method like this:

```
MAYBE_CONMETH (console, mark_console, ...);
```

The XEmacs internals are in fact so much object-oriented that the author has raised the idea of rewriting the core in C++ [C++, 1998] directly several times in the past. However, this issue is still controversial.

1.2 Motivation

It is interesting to note that contrary to the internals of XEmacs, there seem to be much less trace of object-oriented design at the Lisp level (whether in the kernel or in the distributed packages), and when there is, it is also much less apparent. Several hypothesis come to mind, although it could be argued that this is only speculation.

– Most of our Lisp interface is still compatible with that of GNU Emacs, and maintaining this compatibility is an important requirement. This situation is completely different from that of the core, which we are completely free to rewrite as we please.

– The need for object orientation in the Lisp layer might be less pressing than in the core. Indeed, many of the fundamental concepts implemented by the editor are grounded in the C layer, and the Lisp level merely provides user-level interfaces for them.

– The Lisp packages, for an important part, are only a collection of smaller, standalone utilities written by many different people in order to fulfill specific needs, and are arguably of a lower general quality than the editor's core. Emacs Lisp contributors are numerous and not necessarily skilled computer scientists, as they are primarily Emacs *users*, trying to extend their editor of choice, and often learning Emacs Lisp on the occasion. Besides, it wouldn't be their job to provide a language-level feature such as a proper object system.

– Finally, it can also be argued that Lisp is so expressive that an object system (whether proper or emulated) is not even necessary for most packages, as similar features can be hacked away in a few lines of code. In other words, it is a well-known fact that good quality code requires more discipline from the programmer, and that the "quick'n dirty" programming paradigm is on the other hand very affordable, and unfortunately widely used.

These remarks make up for the first motivation in favor of a true object system: the author believes that if provided with such a tool, the general quality, extensibility and maintainability of the Lisp code would improve. More specifically:

– Existing C-based features could provide a true object-oriented interface to the user, in coherence with what they are at the C level.

– Lisp-based features would also greatly benefit from a true object-oriented implementation. The author can think of several ones, such as the `custom` interface, the widget code, the support for editing mode and fontification *etc.*

– Finally, the potential gain is also very clear for some already existing third-party packages. The author thinks especially of Gnus[2], a widely used mail and news reader that he also helps maintaining. This package is almost as large as the whole XEmacs Lisp layer, and provides concepts (such as *backends*) that are object-oriented almost by definition.

Once the gain from having a true Emacs Lisp object system asserted, the next question is obviously which one. We are far from pretending that there is only one answer to this question. Emacs Lisp being an independent Lisp dialect, we could even consider designing a brand new object system for it. However, the author has several arguments that would go in favor of CLOS [Bobrow et al., 1988, Keene, 1989], the object system of the Common Lisp language [ANSI, 1994].

– Emacs Lisp is a dialect of Lisp mostly inspired from MacLISP [Moon, 1974, Pitman, 1983] but also by Common Lisp. There are many similarities between Emacs Lisp and Common Lisp, and because of that, a number of Emacs Lisp developers are in fact familiar with both (the author is one of them). Emacs provides a Common Lisp emulation package known as `cl`. A quick survey of the Sumo packages shows that 16% of the distributed files require `cl` to work. In terms of actual lines of code, the ratio amounts to 27%. Given that these numbers don't even count indirect dependencies, this is far from negligible.

– At least in the subjective view of the author, CLOS is one of the most expressive object system available today, and there is freely available code on which to ground the work.

– There is one Emacs Lisp package that already uses a CLOS-like object system (see section 1.3 on the following page). This package is rather large, as it sums up to almost 70,000 lines of code.

– Having CLOS in Emacs Lisp would considerably simplify the porting of existing Common Lisp libraries that could be potentially useful in Emacs. It could also be a way to attract more Common Lisp programmers to Emacs Lisp development.

– CLOS is already very well documented (books, tutorials *etc.*) and we can take advantage of that.

– Last but not least, the author is interested in gaining expertise in the design and implementation of CLOS and its accompanying meta-object protocol [Paepcke, 1993, Kiczales et al., 1991] (MOP for short). Implementing one is a very good way to achieve this goal.

[2] `http://www.gnus.org`

1.3 Alternatives

The author is aware of two alternative object systems for Emacs Lisp.

- The first one is called Eoops [Houser and Kalter, 1992] (Emacs Object-Oriented Programming System). It implements a class-based, single inheritance, object system with explicit message passing in the vein of Smaltalk-80 [Goldberg and Robson, 1983]. This system dates back to 1992 and the code doesn't seem to have evolved since then. None of the Sumo packages use it and we are not aware of any other Emacs Lisp library using it either.
- The second one is called Eieio (Enhanced Implementation of Emacs Interpreted Objects). It is part of the Cedet[3] package (Collection of Emacs Development Environment Tools). Eieio is more interesting to us because our goals are similar: it *is* designed to be a Clos-like object system. Eieio provides interesting additional features like debugging support for methods, but apparently, it doesn't aim at being *fully* Clos-compliant.

The remainder of this paper is as follows. In section 2, we describe the first stage of this project, consisting in a port of Closette to Emacs Lisp. An overview of the differences between Common Lisp and Emacs Lisp is provided, as well as a more detailed description of how the most problematic issues are solved. In section 3 on page 9, we describe how a deeper integration between types and classes is achieved, with respect to what Closette originally offers. Finally, section 4 on page 13 provides an overview of the features available in ***CL@X***, and compares the project with Eieio.

2 Closette in Emacs Lisp

In order to combine the goals of providing Clos in XEmacs and learning more about its internals at the same time, starting from "Closette" seemed like a good compromise. Closette is an operational subset of Clos described in "The Art of the Meta-Object Protocol" [Kiczales et al., 1991] (Amop for short) and for which source code is available. Consequently, Closette constitutes a convenient base on which to ground the work, without starting completely from scratch. The first step of this project was hence to port Closette to Emacs Lisp. This section describes the most interesting aspects of the porting phase.

2.1 Emacs Lisp *vs.* Common Lisp

While both dialects are similar in many ways, there are some important differences that can make the porting somewhat tricky at times.

[3] `http://cedet.sourceforge.net/eieio.shtml`

2.1.1 Fundamental differences

"Fundamental" differences are obvious ones that might require deep changes in the code. The following differences are considered fundamental: Emacs Lisp is dynamically scoped instead of lexically scoped, has no package system, a different condition system, a limited lambda-list syntax, a different and reduced `format`*ing* and printing facility, and also a different set of types.

2.1.2 Subtle differences

"Subtle" differences are less important ones, but which on the other hand might be less obvious to spot. For instance, some functions (like `special-operator-p` *vs.* `special-form-p`) have different names in the two dialects (this particular case is now fixed in XEmacs). Some others like `defconst` *vs.* `defconstant` have similar names but in fact different semantics. Some functions (like `mapcar`) have the same name but behave differently.

Another example is the `function` special operator which returns a functional value in Common Lisp but simply returns its argument unevaluated in Emacs Lisp. In fact, in Emacs Lisp, `function` is just like `quote` except that the byte-compiler may compile an expression quoted with `function`.

The fact that `function` in Emacs Lisp behaves like `quote` might be puzzling for a Common Lisp programmer, but this is because Emacs Lisp accepts a list beginning with `lambda` as a function designator. For instance, the following two lines are equivalent *and* valid for an Emacs Lisp interpreter, whereas the first one would fail in Common Lisp:

```
(funcall  '(lambda (x) x) 1)
(funcall #'(lambda (x) x) 1)
```

2.1.3 Historical differences

In addition to that, Emacs Lisp is still evolving (this appears to be the case in both GNU Emacs and XEmacs) and the changes are not always clearly documented, if at all. For instance, Emacs Lisp keywords were not self-evaluating before 1996, the `#'` syntax exists since XEmacs 19.8 only, characters were not a primitive type until XEmacs 20, and until August 2009, Common Lisp style multiple values were emulated with lists (they are now built-in). ***CLOX*** is not interested in maintaining backward compatibility with legacy versions of XEmacs or Emacs Lisp, and to be on the safe side, running it typically requires a recent checkout of the 21.5 Mercurial repository[4] (no later than beta 29).

[4] `http://xemacs.org/Develop/hgaccess.html`

2.2 The `cl` package

Emacs provides a Common Lisp emulation package called "`cl`", which is of a tremendous help for porting code from Common Lisp to Emacs Lisp. For the most part, this package provides a number of utility functions or macros that belong to the Common Lisp standard but are not available in raw Emacs Lisp (the almighty `loop` macro is one of them). A number of already existing (but limited) functions are extended to the full Common Lisp power, in which case their names are suffixed with a star (*e.g.* `mapcar*`). `cl` provides `defun*`, `defmacro*` *etc.* to enable the full Common Lisp lambda-list syntax, a version of `typep` and generalized variables via `setf`, `defsetf` *etc.* (`cl`, however, does not support more modern "setf functions").

The remainder of this section provides more details on a couple of interesting porting issues.

2.2.1 Dynamic *vs.* lexical scoping

Perhaps the most important difference between Common Lisp and Emacs Lisp is the scoping policy. Emacs Lisp is dynamically scoped while Common Lisp has lexical scope by default. `cl` provides a construct named `lexical-let` (and its corresponding starred version) that simulates Common Lisp's lexical binding policy via `gensym`'ed global variables. While a brutal replacement of every single `let` construct in Closette would have been simpler, a careful study of the code reveals that this is unnecessary for the most part.

First of all, in the majority of the cases, function arguments or `let` bindings are used only locally. In particular, they are not propagated outside of their binding construct through a lambda expression. Consequently, there is no risk of variable capture and Emacs Lisp's built-in dynamically scoped `let` form is sufficient (it also happens to be more efficient than `lexical-let`).

Secondly, many cases where a true lexical closure is normally used actually occur in so-called "downward funarg" situations. In such situations, the closure is used only during the extent of the bindings it refers to. Listing 1 on the following page gives such an example. The extent of the variable `required-classes` is that of the function. However, the lambda expression that uses it (as a free variable) only exists within this extent. Note that this situation is *not* completely safe, as accidental variable capture could still occur in `remove-if-not`. With a proper naming policy for variables, this risk is considerably reduced, although not completely avoided. In particular, the `cl` package adopts a consistent naming convention (it uses a `cl-` prefix) so that true lexical bindings are unnecessary in practice.

There is a third case in which true lexical bindings can still be avoided, although the situation is an "upward funarg" one: a function that *is* being returned

```
(defun compute-applicable-methods-using-classes (gf required-classes)
  #|...|#
    (remove-if-not #'(lambda (method)
                       (every #'subclassp
                              required-classes
                              (method-specializers method)))
                   (generic-function-methods gf))
  #|...|#)
```

Listing 1: Downward funarg example

and hence might be used outside of the (dynamic) extent of the bindings it refers to. Listing 2 on the next page shows three different versions of the same function.

1. The first one is the original one from Closette. You can see that the returned function has a closure over two variables, which are lexically scoped and have indefinite extent: `methods` and `next-emfun`.

2. The second one follows the logical course of action in Emacs Lisp, using `cl`'s `lexical-let` construct. Recall that function arguments are dynamically bound as well, so we also need to lexically rebind the `methods` variable. Bindings established by `lexical-let` are garbage-collected when the last reference to them disappears, so they indeed get indefinite extent.

3. It turns out, however, that we can still avoid lexical bindings here, by partially evaluating the lambda expression before returning it, as demonstrated in the third version. Remember that in Emacs Lisp, a lambda expression is in fact a self-quoting form, and simply returns a list with the symbol `lambda` in its `car`. Since `methods` and `next-emfun` happen to be constants here, we can pre-evaluate them before returning the lambda expression, hence getting rid of their names which should have been lexically scoped. Finally, note that it is not possible to use `function` or `#'` on a `quasiquote`'ed form, so one might want to call `byte-compile` explicitely on the resulting anonymous function.

Given these particular cases, it turns out that there are only half a dozen places where true lexical bindings are necessary.

2.2.2 Full blown Common Lisp lambda-lists

Because Emacs Lisp is restricted to mandatory, `&optional` and `&rest` arguments, the `cl` package provides replacements for `defun`, `defmacro` *etc.* ***ClOX*** uses these wrappers extensively for its own code, but the question of lambda-lists for generic functions and methods arise. By digging into the internals of `cl`, we are able to provide that at little development cost.

```
(defun compute-primary-emfun (methods)
  (if (null methods)
      nil

      ;; Common Lisp version:
      (let ((next-emfun (compute-primary-emfun (cdr methods))))
        #'(lambda (args)
            (funcall (method-function (car methods)) args next-emfun)))

      ;; Lexically scoped Emacs Lisp version:
      (lexical-let ((methods methods)
                    (next-emfun (compute-primary-emfun (cdr methods))))
        #'(lambda (args)
            (funcall (method-function (car methods)) args next-emfun)))

      ;; Partially evaluated Emacs Lisp version:
      (let ((next-emfun (compute-primary-emfun (cdr methods))))
        `(lambda (args)
          (funcall (method-function ',(car methods)) args ',next-emfun)))
```

Listing 2: Upward funarg example

Internally, `cl` uses a function named `cl-transform-lambda` to both reduce a full blown Common Lisp lambda-list into what Emacs Lisp can understand, and provide the machinery needed for binding the original arguments. Listing 3 on the following page shows an example of lambda-list transformation. Note that the purpose of the second `let` form is to check for the validity of keyword arguments, and disappears if `&allow-other-keys` is provided. In the ***CLOX*** function `compute-method-function`, we take care of wrapping method bodies into a call to `cl-transform-lambda`, hereby providing generic functions with full blown Common Lisp lambda-lists.

Internally (for efficiency reasons), `cl-transform-lambda` uses `memq` to retrieve keyword arguments and hence looks for them at odd-numbered locations as well as even-numbered ones. The drawback of this approach is that a keyword parameter cannot be passed as data to another keyword. Since this does not appear to be much of a problem, we didn't do anything to fix this. This could change in the future, under the condition that the performance of keyword processing in ***CLOX*** does not turn out to be critical.

Also, note that we don't want generic calls to behave differently from normal function calls, so the bindings established by methods remain dynamic.

3 Type/classes integration

Aside from the language differences described in the previous section, the next big challenge to have a working system is to integrate types and classes. This section provides some insight on how this is currently done.

```
;; Original lambda-expression:
(lambda (a &optional (b 'b) &key (key1 'key1))
  BODY)

;; Transformed lambda-expression:
(lambda (a &rest --rest--39249)
  (let* ((b (if --rest--39249 (pop --rest--39249) (quote b)))
         (key1 (car (cdr (or (memq :key1 --rest--39249)
                             (quote (nil key1)))))))
    (let ((--keys--39250 --rest--39249))
      (while --keys--39250
        (cond ((memq (car --keys--39250)
                     (quote (:key1 :allow-other-keys)))
               (setq --keys--39250 (cdr (cdr --keys--39250))))
              ((car (cdr (memq :allow-other-keys --rest--39249)))
               (setq --keys--39250 nil))
              (t
               (error "Keyword␣argument␣%s␣not␣one␣of␣(:key1)"
                      (car --keys--39250))))))
    BODY))
```

Listing 3: Lambda-list transformation example

3.1 Built-in types

As mentioned in the introduction, XEmacs has many opaque Lisp types, some resembling those of Common Lisp (*e.g.* numbers), some very editor-specific (*e.g.* buffers). In XEmacs, there are two basic Lisp types: integers and characters. All other types are implemented at the C level using what is called "lrecords" (Lisp Records). These records include type-specific data and functions (in fact, function pointers to methods for printing, marking objects *etc.*) and are all cataloged in an `lrecord_type` enumeration. It is hence rather easy to keep track of them.

Some of these built-in types, however, are used only internally and are not supposed to be visible at the Lisp layer. Sorting them out is less easy. The current solution for automatic maintenance of the visible built-in types is to scan the `lrecord_type` enumeration and extract those which provide a type predicate function at the Lisp level (the other ones only have predicate *macros* at the C level). Provided that the sources of XEmacs are around, this can be done directly in a running session in less than 30 lines of code.

3.2 Type predicates

3.2.1 `type-of`

Emacs Lisp provides a built-in function `type-of` which works for all built-in types. Because this function is built-in, it doesn't work on ***CLOX*** (meta-)objects. It will typically return `vector` on them, because this is how they are implemented (Closette uses Common Lisp structures, but these are unavailable in Emacs Lisp so the `cl` package simulates them with vectors). In theory, it is possible to wrap this function and emulate the behavior of Common Lisp, but this has not be done for the following reasons.

– Firstly, we think it is better *not* to hide the true nature of the Lisp objects one manipulates. What would happen, for instance, if a ***ClOX*** object was passed to an external library unware of ***ClOX*** and using `type-of` ?

– Secondly, having a working `type-of` is not required for a proper type/class integration (especially for method dispatch).

– Finally, since ***ClOX*** is bound to be integrated into the C core at some point, this problem is likely to disappear eventually.

3.2.2 `typep`

Having an operational `typep` is more interesting to us, and in fact, the `cl` package already provides it. `cl`'s `typep` is defined such as when the requested type is a symbol, a corresponding predicate function is called. For instance, `(typep obj 'my-type)` would translate to `(my-type-p obj)`.

In order to enable calls such as `(typep obj 'my-class)`, we simply need to construct the appropriate predicate for every defined class. In ***ClOX***, this is done inside `ensure-class`. The predicate checks that the class of `obj` is either `my-class` or one of its sub-classes.

In Common Lisp, `typep` works on class *objects* as well as class *names*. This is a little problematic for us because class objects are implemented as vectors, so `typep` won't work with them. However, we can still make this work in a not so intrusive way by using the `advice` Emacs Lisp library. Amongst other things, this library lets you wrap some code around existing functions (not unlike `:around` methods in CLOS) without the need for actually modifying the original code. ***ClOX*** wraps around the original type checking infrastructure so that if the provided type is in fact a vector, it is assumed to be a class object, and the proper class predicate is used.

3.2.3 Generic functions

Generic functions come with some additional problems of their own. In Common Lisp, once you have defined a generic function named `gf`, the generic function *object* returned by the call to `defgeneric` is the *functional value* itself (a *funcallable* object). In Emacs Lisp, this is problematic for two reasons.

1. Since generic functions are objects, they are implemented as vectors. On the other hand, the associated functional value is the generic function's discriminating function, which is different.

2. Moreover, Emacs Lisp's `function` behaves more or less like `quote`, so it will return something different from `symbol-function`.

In order to compensate for these problems, ***ClOX*** currently does the following.

– A function `find-generic-function*` is defined to look for a generic function (in the global generic function hash table) by name (a symbol), functional value (the discriminating function, either interpreted or byte-compiled), or directly by generic function object (note that this can be very slow).

– Assuming that a generic function is defined like this:

  ```
  (setq mygf (defgeneric gf #|...|#))
  (typep mygf 'some-gf-class) ;; already working correctly
  ```

  Class predicates (most importantly for generic function classes) are made to decide whether the given object denotes a generic function in the first place, allowing for the following calls to work properly as well:

  ```
  (typep (symbol-function 'gf) 'some-gf-class)
  (typep #'gf 'some-gf-class) ;; #'gf is more or less like 'gf
  ```

  Note that thanks to the advice mechanism described in section 3.2.2 on the previous page, these calls will also work properly when given a generic function class *object* instead of a *name*.

– `find-method` is extended in the same way, so that it accepts generic function objects, discriminating functions or even symbols.

– Finally, ***ClOX*** defines a `function` class. Consequently, in order for `(typep obj 'function)` to work properly, a second advice on the type checking mechanism is defined in order to try the `function` class predicate first, and then fallback to the original `functionp` provided by Emacs Lisp.

Our integration of generic functions into the type system has currently one major drawback: it is impossible as yet to specialize on functions (either generic, standard, or even built-in). The reason is a potential circularity in `class-of`, as described below.

In order to be able to specialize on functions, we need `class-of` to call `find-generic-function*`. However, `find-generic-function*` might need to access a generic function's discriminating function which is done through `slot-value`, which, in turn, calls `class-of`. This problem is likely to remain for as long as ***ClOX*** generic function objects are different from their functional values. In other words, it is likely to persist until the core of ***ClOX*** is moved to the C level.

4 Project Status

In this section, we give an overview of the current status of ***CL*O*X***, and we also position ourselves in relation to EIEIO.

4.1 Available Features

As mentioned earlier, stage one of the project consisted in a port of Closette to Emacs Lisp. As such, all features available in Closette are available in ***CL*O*X***. For more information on the exact subset of CLOS that Closette implements, see section 1.1 of the AMOP. In short, the most important features that are still missing in ***CL*O*X*** are class redefinition, non-standard method combinations, `eql` specializers and `:class` wide slots.

On the other hand, several additional features have been added already. The most important ones are listed below.

– ***CL*O*X*** understands the `:method` option in calls to `defgeneric`.

– Although their handling is still partial, ***CL*O*X*** understands all standard options to `defgeneric` calls and slot definitions, and will trigger an error when the Common Lisp standard requires so (for instance, on multiply defined slots or slot options, invalid options to `defgeneric` *etc.*).

– ***CL*O*X*** supports the `slot-unbound` protocol and emulates `unbound-slot-instance` and `cell-error-name`. This is because the condition system in Emacs Lisp differs from that of Common Lisp. In particular, Emacs Lisp works with condition *names* associated with data instead of providing condition *objects* with slots.

– ***CL*O*X*** supports a `slot-missing` protocol similar to the `slot-unbound` one. In particular, it provides a `missing-slot` condition which Common Lisp doesn't provide. Common Lisp only provides `unbound-slot`.

– ***CL*O*X*** provides a full-blown set of the upmost classes in the standard Common Lisp hierarchy, by adding the classes `class`, `built-in-class`, `function`, `generic-function` and `method`. The other basic classes like `standard-object` already exist in Closette.

– Finally, ***CL*O*X*** provides an almost complete type/class integration, which has been described in section 3 on page 9.

EIEIO is currently farther away from CLOS than ***CL*O*X*** already is. EIEIO is not built on top of the MOP, doesn't support built-in type/class integration and misses other things like `:around` methods, the `:method` option to `defgeneric` calls, and suffers from several syntactic glitches (for instance, it requires slot definitions to be provided as lists, even if there is no option to them). EIEIO doesn't handle Common Lisp style lambda-lists properly either.

On the other hand, EIEIO provides some additional functionality like a class browser, automatic generation of `TeXinfo`[5] documentation, and features obviously inspired from other object systems, such as abstract classes, static methods (working on classes instead of their instances) or slot protection *ala* C++. The lack of a proper MOP probably justifies having these last features implemented natively if somebody needs them.

4.2 Testing

Given the subtle differences between Common Lisp and Emacs Lisp (especially with respect to scoping rules), the initial porting phase was expected to be error-prone. Besides, bugs introduced by scoping problems are extremely difficult to track down. This explains why a strong emphasis has been put on correctness from the very beginning of this project. In particular, we consider it very important to do regular, complete and frequent testing. This discipline considerably limits the need for debugging, which is currently not easy for the following reasons.

– ***CLOX*** is not equipped (yet) for `edebug`, the Emacs interactive debugger, so we can't step into it.

– ***CLOX*** is not (yet) grounded in the C layer of XEmacs, so we have to use the regular printing facility for displaying (meta-)objects. However, the circular nature of ***CLOX*** requires that we limit the printer's maximum nesting level, hereby actually removing potentially useful information from its output. We also have experimented situations in which XEmacs itself crashes while attempting to print a ***CLOX*** object.

– Finally, most of the actual code subject to debugging is cluttered with *gensym*'ed symbols (mostly due to macro expansion from the Common Lisp emulation package) and is in fact very far from the original code, making it almost unreadable. See listing 3 on page 10 for an example.

Apart from limiting the need for debugging, a complete test suite also has the advantage of letting us know exactly where we stand in terms of functionality with respect to what the standard requires. Indeed, tests can fail because of a bug, or because the tested feature is simply not provided yet.

When the issue of testing came up, using an existing test suite was considered preferable to creating a new one, and as a matter of fact, there is a fairly complete one, written by Paul Dietz [Dietz, 2005]. The GNU ANSI Common Lisp test

[5] `http://www.gnu.org/software/texinfo`

suite provides almost 800 tests for the "Objects" section of the Common Lisp standard, but there are also other tests that involve the object system in relation with the rest of Common Lisp (for instance, there are tests on the type/class integration). Currently, we have identified more than 900 tests of relevance for ***ClOX***, and we expect to find some more. Also, note that not all of the original tests are applicable to ***ClOX***, not because ***ClOX*** itself doesn't comply with the standard, but because of radical differences between Common Lisp and Emacs Lisp.

The test suite offered by Paul Dietz is written on top of a Common Lisp package for regression testing called "`rt`" [Waters, 1991]. Given the relatively small size of the package (around 400 lines of code), we decided to port it to Emacs Lisp. All porting problems described in section 2 on page 5 consequently apply to `rt` as well. The result of this port is an Emacs Lisp package of the same name, which is available at the author's web site[6]. The test suite itself also needed some porting because it contains some infrastructure written in Common Lisp (and some Common Lisp specific parts), but the result of this work is that we now have the whole 900 tests available for use with ***ClOX***.

As of this writing, ***ClOX*** passes exactly 416 tests, that is, a little more than 50% of the applicable test suite. It is important to mention that the tests that currently fail are all related to features that are not implemented yet. In other words, all the tests that *should* work on the existing feature set actually pass. Eieio, on the other hand, passes only 115 tests, that is, around 12% of the applicable test suite. As far as we could see, many of the failures are due the lack of type/class integration.

4.3 Performance

As mentioned earlier, we are currently giving priority to correctness over speed and as such, nothing premature has been done about performance issues in ***ClOX*** (in fact, the performance is expected to be just as bad as that of Closette). Out of curiosity however, we did some rough performance testing in order to see where we are exactly, especially with respect to Eieio. Figure 1 on the next page presents the timing results of five simple benchmarks (presented on a logarithmic scale). These benchmarks are independent from each other and shouldn't be compared. They are presented in the same figure merely for the sake of conciseness. The five benchmarks are as follows.

1. 1000 calls to `defclass` with two super-classes, each class having one slot.

2. 1000 calls to `defgeneric` followed by 3 method definitions.

3. 5,000 calls to `make-instance` initializing 3 slots by `:initarg`'s.

[6] `http://www.lrde.epita.fr/~didier/software/xemacs.php`

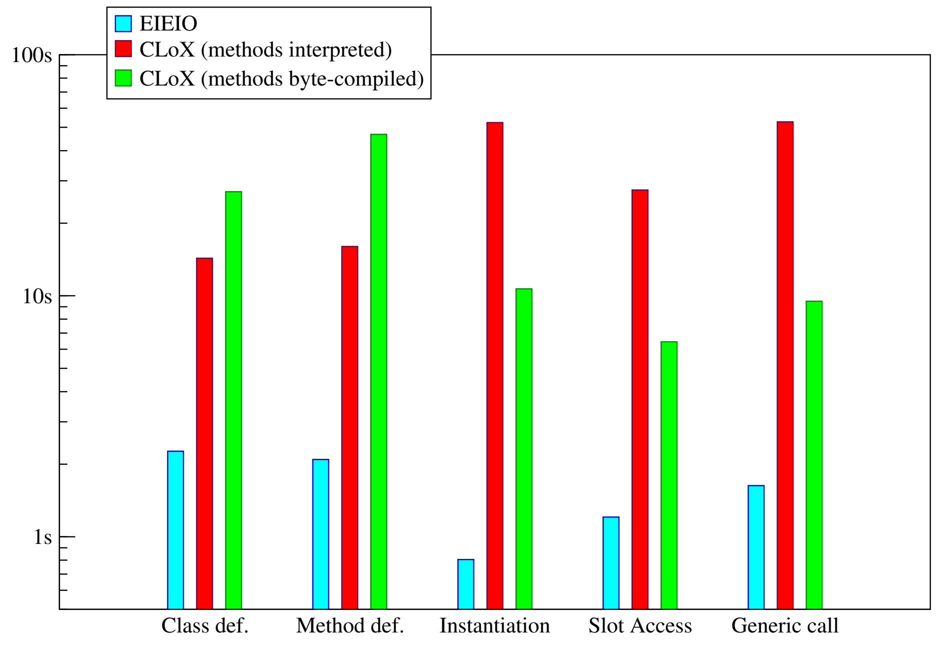

Figure 1: ***CLOX*** *vs.* EIEIO performance

4. 5,000 calls to 3 slot accessors as defined above.
5. 5,000 calls to a generic function executing 3 methods via 2 calls to `call-next-method`.

These benchmarks have been executed in 3 situations each: once for EIEIO, and twice for ***CLOX***, with method bodies and "upward funargs" either interpreted or byte-compiled.

As expected, EIEIO performs much faster than ***CLOX*** in general. Several specificities of these results can be analyzed as follows.

– Class, generic function and method creation are faster in EIEIO by a factor ranging from 7 to 23. This could be due to the fact that EIEIO doesn't have a MOP, so these operations go through ordinary functions (classes are implemented as vectors).

– When method bodies and other lambda expressions are byte-compiled, ***CLOX*** performs the operations above between 2 and 3 times slower. This is precisely because the results include the time used for byte-compilation.

– On the other hand, the situation is reversed for instantiation, slot access and generic calls, as they involve executing byte-code instead of interpreting the original Emacs Lisp code. Here, the gain is roughly a factor of 5.

– Finally, we can see that with method bodies byte-compiled (which is the case in Eieio), instantiation in ***ClOX*** is roughly 10 times slower than in Eieio, while slot-access and generic calls are about 5 times slower only. Given that Eieio is already optimized and does not go through a Mop, these results are better than what the author expected.

In order to improve the performance of ***ClOX*** in the future, several paths are already envisioned.

– First, it is possible to implement a caching mechanism and memoize different computation results within the Mop. The Amop even describes the exact conditions under which a memoized value can be used at some places, for instance in the specification for `compute-discriminating-function` (p. 175).

– Next, there is already abundant litterature on how to improve the efficiency of Clos (see for example [Kiczales and Rodriguez Jr., 1990]). We can benefit from that experience and also get inspiration from how modern Common Lisp compilers optimize their own implementation.

– Finally, when the core of ***ClOX*** is moved to the C layer of XEmacs, an important immediate speedup is also expected.

5 Conclusion

In this paper, we described the early stages of development of ***ClOX***, an attempt at providing a full Clos implementation for XEmacs. Details on the porting of Closette to Emacs Lisp have been provided, as well as some insight on type/class integration and how ***ClOX*** compares to Eieio.

In this project, priority has been given to correctness over speed from the very beginning, which lead us to port `rt` (a Common Lisp library for regression testing) to Emacs Lisp, and also import an important part of Paul Dietz's Gnu Ansi Common Lisp test suite. This priority will not change until all the features are implemented properly.

Ultimately, ***ClOX*** will need to be grounded at the C level, at least because this is necessary for a proper integration of generic functions into the evaluator, but also probably for performance reasons.

Once the system is fully operational, the author hopes to convince the other XEmacs maintainers to actually use it in the core, hereby improving the existing code in design, quality, and maintainability. Otherwise, the system will still be useful for third-party package developers willing to use it.

References

[Apple, 2009] Apple (2009). The Objective-C 2.0 programming language. `http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf`.

[Bobrow et al., 1988] Bobrow, D. G., DeMichiel, L. G., Gabriel, R. P., Keene, S. E., Kiczales, G., and Moon, D. A. (1988). Common lisp object system specification. *ACM SIGPLAN Notices*, 23(SI):1–142.

[C++, 1998] C++ (1998). International Standard: Programming Language – C++. ISO/IEC 14882:1998(E).

[C, 1999] C (1999). International Standard: Programming Language – C. ISO/IEC 9899:1999(E).

[Cardelli, 1988] Cardelli, L. (1988). A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164. A revised version of the paper that appeared in the 1984 Semantics of Data Types Symposium, LNCS 173, pages 51–66.

[Dietz, 2005] Dietz, P. (2005). The Gnu Ansi Common Lisp test suite. In *International Lisp Conference*, Stanford, CA, USA. ALU.

[Goldberg and Robson, 1983] Goldberg, A. and Robson, D. (1983). *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Houser and Kalter, 1992] Houser, C. and Kalter, S. D. (1992). Eoops: an object-oriented programming system for emacs-lisp. *SIGPLAN Lisp Pointers*, V(3):25–33.

[Keene, 1989] Keene, S. E. (1989). *Object-Oriented Programming in Common Lisp: a Programmer's Guide to* Clos. Addison-Wesley.

[Kiczales et al., 1991] Kiczales, G. J., des Rivières, J., and Bobrow, D. G. (1991). *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA.

[Kiczales and Rodriguez Jr., 1990] Kiczales, G. J. and Rodriguez Jr., L. H. (1990). Efficient method dispatch in pcl. In *ACM Conference on Lisp and Functional Programming*, pages 99–105. Downloadable version at `http://www2.parc.com/csl/groups/sda/publications/papers/Kiczales-Andreas-PCL/`.

[Moon, 1974] Moon, D. A. (1974). *MacLISP Reference Manual*. MIT, Cambridge, Massachusetts.

[Paepcke, 1993] Paepcke, A. (1993). User-level language crafting – introducing the Clos metaobject protocol. In Paepcke, A., editor, *Object-Oriented Programming: The CLOS Perspective*, chapter 3, pages 65–99. MIT Press. Downloadable version at `http://infolab.stanford.edu/~paepcke/shared-documents/mopintro.ps`.

[Pitman, 1983] Pitman, K. M. (1983). *The Revised MacLISP Manual*. MIT, Cambridge, Massachusetts.

[Ansi, 1994] Ansi (1994). American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999).

[Waters, 1991] Waters, R. C. (1991). Some useful Lisp algorithms: Part 1. Technical Report 91-04, Mitsubishi Electric Research Laboratories.