\textbf{CL\textsc{ox}}: Common Lisp Objects for XEmacs

Didier Verna

didier@xemacs.org
http://www.xemacs.org
http://www.lrde.epita.fr/~didier

ELS 2010 – Friday, May 7th
Overview

■ What is it ?
  ▶ An implementation of CLOX for XEmacs
  ▶ Including the Meta-Object Protocol (MOP)

■ What’s in it ?
  ▶ A port of Closette to Emacs Lisp
  ▶ Deeper type/class integration
  ▶ A comprehensive test suite
Context
The state of XEmacs

- **About XEmacs**
  - Initially a fork of GNU Emacs v.19-to-be (1991)
  - Greatly diverged since then
  - User-level Emacs Lisp compatibility
  - Internals compatibility

- **About the internals**
  - Very high level of abstraction
  - 111 opaque Lisp types, 35 user-level
  - Core in “C+”: data-abstraction and OO infrastructure
A couple of “C+” examples
So why not use C++ directly? *plonk*

Polymorphism / Class-like abstraction

```c
struct console
{
    enum console_variant       contype;
    void                       *console_data;
    struct console_methods    *conmeths;
    /* ... */
}
```

Dynamic method lookup

```c
MAYBE_CONMETH (console, mark_console, ...);
```
Motivation
Bring the same level of abstraction to the Lisp layer

Why isn’t it the case already?
- GNU Emacs compatibility
- Emacs Lisp *backward* compatibility
- Less “OO pressure” at the Lisp level
  - Glue to the C level
  - Not the job of *package* authors
  - Requires more than the average coding skill
  - Ad-hoc abstraction easier to achieve in Lisp

What would be the benefits?
- C-based features: abstract away the Lisp glue
- Lisp features: improve maintainability / extensibility
- Also for third-party packages (*e.g.* Gnus)
Why CLOS?
Arguments in favor of it

1. **Emacs Lisp**
   - Close to MacLisp and Common Lisp
   - Many developers familiar with Common Lisp
   - CL package dependency: 16% files, 27% LoC

2. **CLOS**
   - One of the most powerful object system around
   - Well documented
   - No need to start from scratch

3. Porting Common Lisp libraries to Emacs Lisp
4. Attract more Common Lisp developers
5. Gain expertise in CLOS and its MOP ;–)
Alternatives
Other available object systems

1. **EOOPS** (Emacs Lisp Object Oriented Programming System)
   - Message passing (Smalltalk style)
   - No activity since 1992
   - No known Emacs Lisp package using it

2. **EIEIO** (Enhanced Implementation of Emacs Interpreted Objects)
   - Active
   - Part of the \textsc{CeDet} package (70,000 LoC)
   - \textsc{Clos}-like
   - Additional features (\textit{e.g.} debugging support)
   - Doesn’t aim at being \textit{fully} compliant
Emacs Lisp vs. Common Lisp
All them dialects, they make my head swim

■ Fundamental differences
  ▶ Dynamic vs. lexical scope
  ▶ No package system, Limited $\lambda$-list syntax
  ▶ Different types, condition system, printing facilities etc.

■ Less obvious ones
  ▶ Different function names
  ▶ Similar functions with different semantics
  ▶ function is different

■ (X)Emacs Lisp evolution
  ▶ Self-evaluating keywords since 1996
  ▶ #' syntax since XEmacs 19.8
  ▶ Primitive character type since XEmacs 19.20
  ▶ Built-in multiple values since a couple of months

► XEmacs no later than 21.5 beta-29 today is required
The CL package
Common Lisp emulation layer

- Provide missing standard utility functions or macros
  \((\text{e.g.} \ \text{loop})\)
- Extend existing but limited ones
  \((\text{e.g.} \ \text{mapcar}^*)\)
- Support full \(\lambda\)-list syntax
  \((\text{defun}^*, \text{defmacro}^* \ \text{etc.})\)
- \text{typep}
- \text{setf} / \text{defsetf} \ (\text{no} \ \text{setf} \ \text{functions})
Dynamic vs. lexical scope

A CloX work Orange

- CL’s `lexical-let` to the rescue
- Not necessary in most cases
  - Local use of `let` bindings or function parameters
  - “Downward funargs” situations
  - “Upward funargs” situations in some cases

- Only 6 actual uses of `lexical-let`
(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))
  #/.../#)
(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)))

    ;; Partially evaluated Emacs Lisp version:
    (let ((next-emfun (compute-primary-emfun (cdr methods))))
      '(lambda (args)
           (funcall (method-function ',(car methods))
                     args ',next-emfun))))
Upward funargs: explanation
Look at the next-emfun argument

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

- lambda is self-quoting
- (lambda ...) is a function designator
  e.g. (funcall '(lambda (x) x) 1)
- function ⇔ quote + byte-compiler hint
- Note: use byte-compile on the resulting form
CL provides the rest *(defun* etc.)*

What about generic functions and methods?
- CL provides `cl-transform-lambda`
- Use it in our `compute-method-function`

### λ-list transformation example

```lisp
;; (lambda (a &optional (b 'b) &key (key1 'key1)) #/.../#)
(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))))))
  #/.../#)
```

Thanks!
Types

*CloX* objects are vectors

- **Built-in types**
  - C level: integers, characters and *lrecord* types
  - User level: corresponding type predicate Lisp function
  - 30 LoC to filter them out

- **type-of doesn’t work on *CloX* objects**
  - Better not hide the *true* nature of Lisp objects
  - Not required to make *CloX* work
  - Will work *eventually* (C level)

- **typep is provided by CL**
  - Need predicate functions:
    
    ```lisp
    (typep obj 'my-type) ⇔ (my-type-p obj)
    ```
  - ensure-class creates them
  - Should work on class *objects*, not only *names*
  - *CloX*-specific *defavice around it*
Generic functions specific problems

**Generic function objects are funcallable in Common Lisp**

---

This is already working

```
(setq mygf (defgeneric gf #/.../#))
(typep mygf 'some-gf-class)
```

- **CL** generic functions are vectors
  - Not funcallable
  - ≠ discriminating function
- **function ≈ quote**
  - function *doesn’t return a functional value*
  - `symbol-function` *does*
Solution
Handle all references to generic functions

- by name (symbol)
- by functional value
- by object (vector)

\texttt{find-generic-function}\footnote{\texttt{my-class-predicate-p}}

\begin{itemize}
  \item \texttt{typep (symbol-function \textquote{gf}) \textquote{some-gf-class}}
  \item \texttt{typep \#\textquote{gf} \textquote{some-gf-class}} \texttt{;; or just \textquote{gf}}
\end{itemize}

\begin{itemize}
  \item \texttt{find-method (symbol-function \textquote{gf})} \texttt{;; ...}
  \item \texttt{find-method \#\textquote{gf}} \texttt{;; ... or just \textquote{gf}}
\end{itemize}

\begin{itemize}
  \item \texttt{typep (symbol-function \textquote{gf}) \textquote{function}}
  \item \texttt{typep \#\textquote{gf} \textquote{function}} \texttt{;; or just \textquote{gf}}
\end{itemize}
One (major) drawback
Rock around the CL\O\X

Method specialization on functions \textit{impossible}
Available Features...

- **Closette** (see section 1.1 of the AMOP)
  - Missing: class redefinition, non-standard method combinations, eql specializers and :class wide slots

- **And beyond**
  - :method option to defgeneric
  - Aware of the others defgeneric and slot options
  - slot-unbound protocol with emulated unbound-slot-instance and cell-error-name
  - Extended slot-missing protocol
  - Topmost class hierarchy (class, built-in-class, function, generic-function and method)
  - Almost complete type/class integration
... Compared to EIEIO

- **What’s missing**
  - No underlying MOP
  - No type/class integration
  - No :around methods
  - No :method option to defgeneric calls
  - Buggy \(\lambda\)-list support
  - Syntactic glitches

- **Additional bells’n whistles**
  - A class browser
  - Automatic \(\LaTeX\)info documentation
  - edebug support
  - Abstract classes, static methods, C++-like slot protection
Testing
GRR #1: it is better to be correct than to be fast

- Why early testing?
  - No `edebug` support
  - Limited printing facility (circular structures)
  - `gensym` infestation

- Paul Dietz’s ANSI Common Lisp test suite
  - 900 tests related to objects
  - Ported to Emacs Lisp
  - Corollary: `rt` package ported as well

- Current status
  - 50% of the tests pass
  - 100% of the tests that pass are ok ;-)
  - EIEIO: only 12%
Performance
GRR #2: it is better to be fast than to be slow (someday)
Conclusion

■ **What we have**
  - An implementation of CLOS for Emacs Lisp
  - A comprehensive test suite
  - Corollary: an Emacs Lisp port of \texttt{rt}

■ **Next steps**
  - Complete the feature set
  - Improve performance (C-level integration)
  - Port to GNU Emacs

■ **Potential uses**
  - Core features (\textit{e.g.} specifiers)
  - User libraries (\textit{e.g.} Gnus \texttt{*Hmpf! *} )
Any questions?

“And with a four-button mouse it gets even better. I bound something to Ctrl-Meta-Hyper-Super-button4, just because I could.”

-- Calle Dybbedahl