CLOS discriminating functions and user-defined specializers

Christophe Rhodes*
Goldsmiths, University of London
New Cross, London SE14 6NW

July 23, 2007

Abstract

We discuss the possibility for users of CLOS to extend the `mop:specializer` metaobject class in the *de facto* standard Metaobject Protocol for Common Lisp, and how this possibility interacts with ANSI-standardized functionality. To motivate the discussion, we provide two simple examples: a specializer on a disjunction of classes and a simple pattern-matching specializer; we note the extent to which they can be accomodated with the standard mechanisms, and detail the work done to support that in a contemporary implementation of the CLOS MOP in Steel Bank Common Lisp, and discuss the remaining open problems and scope for resolving them.

1 Introduction

Lisp has a venerable history of object-oriented programming; at one point in time, early in the history of object-orientation, Flavors (Moon, 1986) and New Flavors, Common Objects, Object Lisp and Common Loops (Bobrow et al., 1986) all coexisted. The Common Lisp Object System (CLOS) was incorporated into the language in June 1988 (Steele, 1990, Chapter 26), and when the ANSI Common Lisp standard (Pitman and Chapman, 1994) was formalized in 1995, Common Lisp became the first ANSI-standardized programming language with support for object-oriented programming.

In addition, CLOS was developed in conjunction with the design of a Metaobject Protocol (MOP), as exemplified in Kiczales et al. (1991). Common Lisp as standardized only includes a very small portion of this metaobject protocol (for instance, a recommendation to use `mop:slot-value-using-class` in `slot-value`; some introspective functionality such as `find-method`; and arguably a little ability for intercession in `compute-applicable-methods`, though in fact the standard does not require that `compute-applicable-methods` be called as part of generic function dispatch), and so to customize the behaviour of the object system in Common Lisp, it is necessary to go beyond the standard language.

Many Common Lisp implementations support some of the MOP, to varying extents; a survey from a few years ago (Bradshaw and de Lacaze, 2000) revealed many aspects of MOP support as

*c.rhodes@gold.ac.uk

being incomplete, even at the coarse level of specified classes and generic functions being unimplemented. More recently, the Closer[1] project has provided both a set of test cases for implementations of the Metaobject Protocol – which has encouraged some implementations to enhance their support for it[2] – and a compatibility layer to provide an environment as close as possible to that described in AMOP in major implementations of Common Lisp.

This paper addresses primarily the issues found and resolved in supporting simple uses of non-standard specializers. Specifically, we examine those uses for which certain simplifying assumptions can be made, for example assumptions about ordering the specializers: either that there is a unique order of specificity in all possible specializers that are applicable to any given object, or else that any ambiguities in ordering do not affect the result of calling the generic function. Even with these simplifying assumptions, it is necessary to go beyond the standard portions of the Metaobject Protocol to achieve integration of non-standard specializers with the rest of the system.

The rest of this paper is organized as follows: after introducing some details about the historical treatment of specializers in the CLOS MOP, and other related work, this paper in section 2 attempts to motivate the use of non-standard specializers by presenting two simple examples. We discuss the problems that were overcome in providing the support for even simple examples of non-standard subclasses of `mop:specializer` and enumerate other, open problems in section 3, and finish by detailing one particular avenue for possible further experimentation along with our conclusions.

1.1 Specializers in the CLOS MOP

Closer is an evolving project, and does not at present test all aspects of the Metaobject Protocol – concentrating on the portions which are most clearly described. One aspect which has received relatively little attention at the time of writing is the extensible nature of the `mop:specializer` metaobject class, whose instances represent a description of the set of objects in one argument position to which a method with that specializer is applicable.

For `defmethod`, there is only a surface syntax defined by the Common Lisp standard, with *parameter specializer names* being class names for matching a class, and `(eql` *`form`*`)` for specializers to match the value of *form* in the lexical environment. However, for `find-method`, the ANSI standard defines a *parameter specializer* as either a class or a list `(eql` *`value`*`)`.

The Art of the Metaobject Protocol defines the objects backing the surface syntax as being `class`es themselves and objects of class `mop:eql-specializer-object` obtained by calling `mop:intern-eql-specializer`; both `class` and `mop:eql-specializer-object` are subclasses of `mop:specializer`. There are discussions on the CLOS MOP mailing list between Gregor Kiczales and David Moon about whether the nature of the `mop:eql-specializer-object` is strictly incompatible with the language which was eventually standardized by ANSI[3]: Moon specifically points out the need for a compatibility translation for elements of the the *specializers* argument to `find-method`.

AMOP leaves open (does not disallow explicitly) user-defined subclasses of `mop:specializer` (but standard methods on *e.g.* `compute-applicable-methods` signal error on non-standard specializer classes; we shall return to this point later). We can examine the historical record of the development of CLOS by inspecting the source code of Portable Common Loops (PCL), modified

[1] `http://common-lisp.net/project/closer/`

[2] At the time of writing, the MOP implementation of Steel Bank Common Lisp (Newman et al., 2000) fails none of the Closer MOP test suite.

[3] Thanks to Pascal Costanza for bringing this exchange to the author's attention.

versions of which are used in a number of Common Lisp implementations: PCL internally defines and uses a `pcl:class-eq` specializer, which is applicable only to objects whose class is `eq` to the specializer's class object (as opposed to a normal `class` specializer, which is also applicable to objects whose class is a subclass of the specializer).

Additionally, there is evidence that PCL developers were interested in developing more esoteric specializers, as there are remnants of what appears to be an experiment in implementing prototype-based dispatch: in the source code supporting `mop:compute-applicable-methods-using-classes`, there is incomplete support for of `pcl:prototype` specializers, such as

```
(defun saut-prototype[4] (specl type)
  (declare (ignore specl type))
  (values nil nil)) ; fix this someday
```

1.2 Related Work

Predicate dispatching – a dispatching system in which a method's applicability is determined by calling an arbitrary predicate – has been discussed in Ucko (2001), where the solution discussed was to extend method qualifiers (arbitrary predicates not being associated with any particular argument, and methods being distinguished from each other only on the basis of qualifiers and specializers). The author notes portability difficulties with this approach, which would likely still be present today: for example, some implementations will only accept non-standard qualifiers if the generic function has a non-standard method combination, while a strict implementation of `define-method-combination` will lead to errors if methods are placed in the same method group with the same specializers (irrespective of how their qualifiers later affect dispatch).

2 Worked Examples

2.1 Disjunction specializer

The disjunction specializer we present here is applicable to arguments that are subclasses of any of the classes of the specializer. While a disjunction specializer is not well-motivated from the point of view of object-orientation – classes whose instances share behaviour should probably have that shared behaviour modelled by a common superclass – the implementation details of such a specializer are illustrative of some points.

Firstly, such a disjunction specializer has a natural place within the standard ordering of specializers: such a specializer is as specific as a specializer representing a common direct superclass of the classes in question. If such a class in fact exists, then the tie can be broken in an arbitrary (but consistent) way by choice of a suitable convention.

The major point to note is that, despite this natural and unambiguous ordering of the specializers, the entirety of `compute-applicable-methods` and `mop:compute-applicable-methods-using-classes` need to be overridden as the standard methods on those functions are specified to signal an error if they encounter a specializer which is neither a `class` nor an `mop:eql-specializer`. Because the standard methods are otherwise opaque, there is no way of informing the system about

[4] `saut` here stands for `specializer-applicable-using-type`, a function internal to PCL for which this is one among several helper routines.

```
(defclass class1 () ())
(defclass class2 () ())
(defclass class3 () ())
(defclass class4 (class1) ())

(defgeneric foo (x)
  (:generic-function-class gf-with-or))

(let ((specializer (ensure-or-specializer 'class1 'class2)))
  (eval `(defmethod foo ((x ,specializer)) t)))

(assert (foo (make-instance 'class1)))
(assert (foo (make-instance 'class2)))
(assert (raises-error? (foo (make-instance 'class3))))
(assert (foo (make-instance 'class4)))
```

Figure 1: Example code illustrating the semantics of a specializer on the disjunction of multiple classes.

a 'natural' ordering for the non-standard specializers, and so a large amount of complex code needs to be written by the user of non-standard specializers.

Additionally, some notion of specializer equality needs to be present, so that method redefinitions behave as expected (removing an existing method with the semantically same specializer). In implementing the disjunction specializer, we act conservatively by canonising unions to the same, `eql`, specializer object: this is again a non-trivial operation, but it is necessary not only for `find-method` to work as expected but also for relatively simple implementations of `mop:specializer-direct-methods` and `mop:specializer-direct-generic-functions`.

2.2 Pattern specializer

A pattern specializer is perhaps more applicable in a well-founded way to programming, though its connection to object-orientation is tenuous. However, the protocols exposed in the CLOS MOP provide a convenient way to express functions using pattern-matching for dispatch (as in ML or Haskell), without sacrificing the ability to program in a dynamic fashion typically afforded by CLOS.[5]

An example of how we might wish to use a pattern-matching specializer is shown in figure 2. The application example is the simplification module of a hypothetical computer algebra system, and we suggest that it is reasonable to place methods for simplifying expressions with given operators near the definition of other methods relating to those operators, such as their semantics or possibly their graphical representation, so that the addition of a new operator need not involve editing of existing functions.

[5]We could of course implement pattern functions simply using the same funcallable instances on which CLOS is typically implemented. The benefits of using the existing CLOS MOP protocols are simplicity and parsimony; however, it is possible that an implementation of pattern-matching functions might wish not to document that it is implemented in this manner.

```
(defgeneric simplify (x)
  (:generic-function-class pattern-gf/1))
(defmethod simplify ((y _)) y)

;;; near the implementation of the * operator
(defmethod simplify ((x (* _ 0))) 0)
(defmethod simplify ((x (* 0 _))) 0)
(defmethod simplify ((x (* _ 1))) (simplify (second x)))
(defmethod simplify ((x (* 1 _))) (simplify (third x)))

;;; near the implementation of the + operator
(defmethod simplify ((x (+ _ 0))) (simplify (second x)))
(defmethod simplify ((x (+ 0 _))) (simplify (third x)))
```

Figure 2: An example for how a pattern-matching specializer might improve code locality, by encouraging the implementation of relevant portions of overarching functionality to be located near related definitions. We have intentionally kept the pattern-matching language simple for presentational purposes.

In this example, there is no strongly-defined unambiguous ordering for the specializers: of the patterns `(_ . x)` and `(y . _)`, it is not clear which should follow the other. Note that in our example application for simplification, the same results are obtained whichever of the specializers is treated as most specific (*e.g.* for an argument of `(* 1 1)` the return value will be `1`, whichever of the applicable `*` patterns is judged most specific.

The metaobject classes and support functions allowing the code in figure 2 to run are shown in figure 3. These, along with the overriding of `mop:compute-discriminating-function` in figure 4, provide enough support for SBCL's CLOS implementation to function as expected. Note in particular that we again canonicalise specializers so that specializers with the same semantics are `eql`, and also note the use of `pcl:make-method-specializers-form` to generate code that creates the right specializer when evaluated.

Although we have assumed in figure 2 that the pattern matching specializers match list structure, it would be straightforward to adapt the code in figure 4 to match instead patterns in a richer domain-specific data structure, while keeping the lists as surface specializer syntax.

The code in figure 4 is an interpreter for our pattern language; we can of course implement a compiler for that language and use that compiler to generate more efficient discriminating functions. A simple compiler (without any optimizations based on static analysis of the set of patterns) is shown in figure 5; more complex compilation strategies (see *e.g.* Le Fessant and Maranget (2001) and references therein) would improve performance here.

We note that the method dispatch protocol means that we cannot straightforwardly get full micro-efficiency with these pattern methods: we must destructure the argument to check whether a given method is applicable, but we must invoke the method with a list of the (undestructured) arguments. It is possible that a suitable override of `mop:make-method-lambda` might enable us to recover this loss of efficiency, and also possibly to refer to pattern variables in method bodies.

```
(defclass pattern-specializer (specializer)
  ((pattern :initarg pattern :reader pattern)
   (direct-methods :initform nil :reader specializer-direct-methods)))
(defvar *pattern-specializer-table* (make-hash-table :test 'equal))
(defun ensure-pattern-specializer (pattern)
  (or (gethash pattern *pattern-specializer-table*)
      (setf (gethash pattern *pattern-specializer-table*)
            (make-instance 'pattern-specializer 'pattern pattern))))
(defclass pattern-gf/1 (standard-generic-function) ()
  (:metaclass funcallable-standard-class)
  (:default-initargs :method-class (find-class 'pattern-method)))
(defclass pattern-method (standard-method)
  ((lambda-expr :initarg :lambda-expr :reader pattern-method-lambda-expr)))
(defmethod sb-pcl:make-method-specializers-form
    ((gf pattern-gf/1) method snames env)
  `(list ,@(mapcar (lambda (s) `(ensure-pattern-specializer ',s)) snames)))
```

Figure 3: Implementation classes and methods for one-argument generic functions with methods specialized on patterns.

```
(defun matchesp (arg pattern)
  (cond
    ((or (null pattern) (eq pattern '_)) t)
    ((atom pattern) (eql arg pattern))
    (t (and (matchesp (car arg) (car pattern)) (matchesp (cdr arg) (cdr pattern))))))
(defun method-interpreting-function (methods gf)
  (lambda (arg)
    (dolist (method methods (no-applicable-method gf (list arg)))
      (when (matchesp arg (pattern (car (method-specializers method))))
        (return (funcall (method-function method) (list arg) nil))))))
(defmethod compute-discriminating-function ((gf pattern-gf/1))
  (method-interpreting-function (generic-function-methods gf) gf))
```

Figure 4: Implementation of a discriminating function for one-argument pattern-matching method dispatch. Note that the result is dependent on the order of method definition, but also that we are able to cache the generic function methods (as the discriminating function is recomputed if methods are added or removed).

```
(defun compile-matcher (arg pattern success fail)
  (cond
    ((or (null pattern) (eq pattern '_)) success)
    ((atom pattern) `(if (eql ,arg ',pattern) ,success ,fail))
    (t (let ((car-name (gensym "CAR")) (cdr-name (gensym "CDR")))
         `(if (consp ,arg)
              (let ((,car-name (car ,arg)) (,cdr-name (cdr ,arg)))
                (declare (ignorable ,car-name ,cdr-name))
                ,(compile-matcher car-name (car pattern)
                                  (compile-matcher cdr-name (cdr pattern)
                                                   success fail)
                                  fail))
              ,fail)))))
```

Figure 5: A simple compiler for the pattern language of figure 4, taking an argument name, pattern, and success and failure forms, and returning code to perform the discrimination.

3 Future Work

For some of the problems noted in this paper, we can propose solutions that we believe are compatible in spirit with the *de facto* standard Metaobject Protocol, though there may be difficulties in incorporating those solutions into existing implementations of Common Lisp; we have validated those solutions to the point of testing them with SBCL's implementation of the Metaobject Protocol. There are additionally some issues to note that we have not attempted to address in our implementation.

3.1 Solved problems

To support use of `find-method` with non-standard specializers, we suggest that there should be a generic function equivalent to `pcl:parse-specializer-using-class`, dispatching on the generic function's class and returning a specializer object – and that for introspective purposes, it is also reasonable to provide `pcl:unparse-specializer-using-class`.

However, `pcl:parse-specializer-using-class` is insufficient to support the implementation of specializer names in `defmethod`, as the issue of running in the correct lexical environment is important. In

```
(defmethod foo ((x integer) (y (eql bar))) ...)
```

the specializers are treated in a similar fashion to the method body, in that some of the portions are evaluated in the lexical environment: specifically, while `integer` and `eql` are syntactic markers, it is the value of `bar` when the `defmethod` form is executed that is the object being specialized on, rather than the symbol `bar`. To support this (and arbitrary other ways of handling specializers in `defmethod` forms) we suggest a new operator `pcl:make-method-specializers-form`, analogous to `mop:make-method-lambda`, which should return a form which (when evaluated in the right lexical environment) evaluates to a list of specializers[6]. Note that this function suffers from one of the same

[6]Although Common Lisp and the CLOS MOP do not define any qualifiers as having portions evaluated in the

design quirks as `mop:make-method-lambda`, in that specializing it requires the generic function to be extant and as an instance of its final class at the point when the `defmethod` form is macroexpanded.

Individual implementations will need to take care over various codepaths; certainly PCL as used in SBCL implements certain optimizations which can be invalidated by user-defined specializers, but because of the lack of attempts at using this facility, the optimizations were written insufficiently defensively. For example, in `mop:make-method-lambda`, the system method attempted to insert type declarations into the lambda corresponding to system class specializers, which allows for efficient method functions with no extra user-provided declarations. Another case is that if the second return value from `mop:compute-applicable-method-using-classes` is null while the first is not, the system treats the first value as a list of possibly-applicable methods which it should build a discrimination net to distinguish between. Both of these cases caused problems to the initial implementation of user-defined specializers.

3.1.1 Dictionary

Generic Function `parse-specializer-using-class`
Syntax:
`parse-specializer-using-class` *generic-function specializer-name*

This generic function returns an instance of `mop:specializer`, representing the specializer named by *specializer-name* in the context of *generic-function*.

Primary Method `parse-specializer-using-class` (*gf* `standard-generic-function`) (*name* `t`)

This method applies the standard parsing rules for consistency with the specified behaviour of `find-method`.

Generic Function `unparse-specializer-using-class`
Syntax:
`unparse-specializer-using-class` *generic-function specializer*

This generic function returns the name of *specializer* for generic functions with class the same as *generic-function*

Primary Method `unparse-specializer-using-class`
(*gf* `standard-generic-function`) (*specializer* `specializer`)

This method applies the standard unparsing rules for consistency with the specified behaviour of `find-method`.

Generic Function `make-method-specializers-form`
Syntax:
`make-method-specializers-form` *generic-function method specializer-names env*

This function is called with (maybe uninitialized, as with the analogous arguments to `mop:make-method-lambda`) *generic-function* and *method*, and a list of specializer names (being the parameter specializer names from a `defmethod` form, or the symbol `t` if unsupplied), and returns a form which evaluates to a list of specializer objects in the lexical environment of the `defmethod` form.

Primary Method `make-method-specializers-form`
(*gf* `standard-generic-function`) (*method* `standard-method`) *names env*

This method implements the standard behaviour for parameter specializer names.

lexical environment of the `defmethod` form, if a user were to need this, then a similar operator `pcl:make-method-qualifiers-form` could be implemented.

3.2 Open problems

The greatest problem in opening specializers to user definition lies in method ordering and method combination. In the general case, there is no single unambiguous ordering of a set of applicable methods, even given particular arguments. This is largely a problem for the implementor of a specializer, as `compute-applicable-methods` and `mop:compute-applicable-methods-using-classes` must be overriden in any case, and it is the return value of those functions that determines the order of specificity.

However, this does imply that the entirety of `compute-applicable-methods` and `mop:compute-applicable-methods-using-classes` must be reimplemented for every new generic function class accepting extended specializers – a non-trivial amount of work, as implementing those functions (even with just the standard behaviour) comes to hundreds of lines of code. This is probably not optimal, and suggests that there is scope for experimentation with a protocol to replace the applicable method computation performed by `compute-applicable-methods`.

A sketch of such a protocol might include calling a function `pcl:specializer-of` (taking two arguments: an object and the generic function being called, and returning the most specific specializer applicable to the object for which cacheing of applicable methods is suitable) instead of `pcl:class-of`, so that methods on `pcl:specializer-of` specialized on a generic function class could be written to return an application-specific specializer for a given argument object; then a function `pcl:compute-applicable-methods-using-specializers` could replace the computation and cacheing functionality of `mop:compute-applicable-methods-using-classes`.

Another ingredient of such a protocol might be a predicate for determining whether a specializer is more or less specific than another (say `pcl:specializer-lessp`, along with `eql` to test for equal specificity), and defining how this is called from within the method ordering protocol functions. We suggest that experimentation along the lines of this sketch, using a defined generic function class, is a way to begin exploring the issues involved in the interoperability of user-defined specializers with the standard specializers and with each other.

In practice, the major impediment to the use of `mop:specializer` subclasses is that support for them is patchy to nonexistent. Aside from recent work in SBCL, which could be straightforwardly ported to PCL-based CLOS implementations such as CMUCL and GCL, there are architectural hurdles to clear: Allegro Common Lisp and GNU CLISP do not implement `mop:make-method-lambda`, and do not allow subclasses of `mop:specializer` to be used (even as literals) as arguments to `defmethod`. Thus, there is no way to create a method with non-standard specializers in those implementations. Lispworks Common Lisp does not include the `mop:specializer` metaobject class at all, representing `eql` specializers as conses – which makes it rather difficult to subclass `mop:specializer` in the first place. MCL-derived Lisps have an entirely different generic function calling protocol, so much of the discussion above does not apply to them, while most of the newer Common Lisp implementations do not purport to support the Metaobject Protocol.

4 Conclusions

We have presented our work allowing the MOP programmer to define subclasses of `mop:specializer` which integrate cleanly with the rest of CLOS, including the need for a protocol operator to act at macroexpansion time in a similar fashion to `mop:make-method-lambda`. We have additionally enumerated some open problems with user-extensible specializers, and suggested a path for experimenting with possible protocol resolutions to these open problems.

Acknowledgments

The author thanks Paul Khuong[7] and the workshop reviewers for valuable discussions and helpful feedback.

References

Bobrow, D. G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., and Zdybel, F. (1986). Common Loops: Merging Lisp and Object-Oriented Programming. In *OOPSLA'86 Proceedings*, pages 17–29.

Bradshaw, T. and de Lacaze, R. (2000). A Survey of Current CLOS MOP Implementations. In *Japan Lisp Users Group Meeting*.

Kiczales, G., des Rivières, J., and Bobrow, D. G. (1991). *The Art of the Metaobject Protocol*. MIT Press.

Le Fessant, F. and Maranget, L. (2001). Optimizing Pattern Matching. In *ICFP'01 Proceedings*, pages 26–37.

Moon, D. (1986). Object Oriented Programming with *Flavors*. In *OOPSLA'86 Proceedings*, pages 1–8.

Newman, W. H. et al. (2000). SBCL User Manual. `http://www.sbcl.org/manual/`.

Pitman, K. and Chapman, K., editors (1994). *Information Technology – Programming Language – Common Lisp*. Number 226–1994 in INCITS. ANSI.

Steele, Jr, G. L. (1990). *Common Lisp: The Language*. Digital Press, second edition.

Ucko, A. M. (2001). Predicate dispatching in the Common Lisp Object System. Technical Report AITR-2001-006, MIT AI Lab, Cambridge, MA. MEng thesis.

[7] `khuongpv@iro.umontreal.ca`