Generic Function Parameter and Method Parameter Metaobjects: A Proposed Enhancement to the CLOS Metaobject Protocol

Eric L. Peterson*
Artificial Intelligence Technologies Center
MITRE Corporation
1820 Dolley Madison Blvd.
McLean, VA 22102-3481
e-mail: eric@ai.mitre.org
URL: http://www.cs.umd.edu/users/ericp
phone: (703) 883-6116
FAX: (703) 883-6435

April 25, 1996

Abstract

Common Lisp's Metaobject Protocol defines classes of objects which embody various aspects of the object oriented programming environment. Current such metaobjects represent classes, slots, generic functions, methods, as well as other object oriented entities. Not currently included, however, are objects that represent generic function and method parameters. This paper first gives class definitions declaring the behavior of these proposed parameter metaobjects and then proceeds to argue for for their inclusion in the expected addition of the Metaobjects Protocol to the ANSI Common Lisp Standard.

1 Introduction

The Common Lisp Object System's (CLOS) Metaobject Protocol (MOP), as put forth in *The Art of the Metaobject Protocol* (AMOP) [2], provides metaobjects representing many aspects of the state of the dynamic object oriented program.[1] It is widely embraced as a de facto standard.[2] Perhaps the largest aspect of the object oriented program still untamed by the MOP is the rendering of the actual method code as a network of objects. This paper, however, is concerned with the taming of a simpler aspect.

2 The Proposal

This paper argues for the incorporation of five new instantiable metaobject classes as well as their ancestors into the CLOS MOP prior to the MOP's incorporation into the Common Lisp ANSI standard. These metaobjects represent generic function (GF) and method parameters. Their instantiable classes are as follows:

- *standard-generic-function-parameter*
- *standard-method-parameter*
- *standard-optional-parameter*
- *standard-rest-parameter*

[0] Eric Peterson is a member of the Machine Translation Group at the Artificial Intelligence Technologies Center at the MITRE Corporation. Thanks go to Lawrence Mayka and David Terebessy for review comments and to Lawrence for the CLIM example/argument.

[1] The term MOP, in this paper will be understood to mean the AMOP MOP.

[2] Slight differences in metaobject protocols exist between Common Lisp implementations.

- *standard-keyword-parameter*

The first two parameters represent the required parameters from generic function and method lambda lists respectively. The remaining parameter metaobjects represent non-required parameter information and are to be used in either generic function or method lambda lists containing non-required parameters.[3] Please see Figure 1 for the definitions of these classes and for the descriptive information found in their CLOS documentation strings.

This proposed family of metaobjects would require a new *defmethod*-like macro to allow for the input of metadata beyond what is found in a standard *defmethod* macro. See Figure 2 for an example use of such a macro. The conventional *defmethod* macro would still be available, but would serve to load conventional parameter information into metaobjects of the kind described in this paper. It would still serve ordinary object oriented programming needs in the traditional fashion. The new macro would be for those who wish to document their parameters in a run-time retrievable fashion, as well as for those who which to associate additional information with those parameters. To some, it may appear overly verbose, yet it should be noted that it is no more verbose than the CLOS slot specifications within the *defclass* macro. In both cases, terse representations would not allow for seamless addition of non-standard attributes as is done by the subclassing of slot definition metaobjects.

Both *standard-generic-function* and *standard-method*, two existing MOP metaobjects, would receive a new slot called *parameters*.[4] The lists would not contain the traditional *&optional*, *&rest*, *&key*, and *&allow-other-keys* delimiter tokens. None would be needed, because the types of the metaobjects would be immediately determinable. An additional sibling slot to *parameters* would be called *allow-other-keys* and would take the place of the lambda list token of the same name and would serve as a boolean indication as to whether other keys would be allowed.

The following five new GF's would be added to the MOP:

- *generic-function-parameter-class*
- *method-parameter-class*
- *optional-parameter-class*
- *rest-parameter-class*
- *keyword-parameter-class*

They would be analogous in purpose to the *effective-slot-definition-class* GF. They would allow the CLOS user/extender to communicate the presence of parameter metaobject subclasses to the CLOS/MOP environment.f

It may be argued that *supplied-p* does not belong in a parameter metaobjects because it has nothing to do with the specification of the GF/method interface. Although it is clearly a piece of utility information for the benefit of the method(s)' lexical environment, this proposed change, like current lambda lists, makes no distinction between *interface specification*, and *utility* information. It is proposed that any information that is logically associated with such a parameter, is not only welcome, but encouraged to become part of a parameter metaobject.

&aux parameters would simply be treated as if macro expanded into *let* expressions, as is presently done. They would have no metaobjects.[5]

Caching of parameter metadata would, as allowed by the MOP, be permitted. Therefore, implementations would be free to create and cache conventional lambda lists in GF and method metaobjects. Implementations could thus obtain runtime performance identical to their present implementations for generic dispatch not involving additional parameter metadata from user enhanced parameter metaobjects.

[3] The term non-required is used to refer to *&optional*, *&rest*, and *&keyword* parameters. The term *optional* would, of course, have been ambiguous.

[4] Both the use of the term *slot* and the use of class definitions with slots are for expository convenience. Precisely speaking, such behavior could be implemented without the use of slots. Assume that the behavior is *as if* defined as shown.

[5] Future work alluded to later in this paper will require metaobjects for program variables in order to house variable related metadata. This suggests the possibility that metaobjects for auxiliary variables would be needed as well.

```
(defclass parameter (clos:metaobject)
  ((parameter-name
    :type symbol
    :documentation "The name of a lambda list parameter"
    :accessor parameter-name)
   (parameter-documentation
    :type string
    :documentation "User supplied documentation of parameter"
    :accessor parameter-documentation
    :initarg :documentation))
  (:documentation "Information pertaining to a given lambda list parameter"))

(defclass standard-generic-function-parameter (parameter) ()
  (:documentation "Information pertaining to a generic function parameter"))

(defclass standard-method-parameter (parameter)
  ((method-parameter-specializer
    :type (or symbol list)
    :documentation "Parameter specializer for generic dispatch"
    :accessor method-parameter-specializer
    :initarg :specializer
    :initform t))
  (:documentation "Information pertaining to a given method parameter"))

(defclass optional-parameter (parameter)
  ((optional-parameter-default
    :documentation "Default value for &optional parameters"
    :accessor optional-parameter-default
    :initarg :default
    :initform nil)
   (optional-parameter-supplied-p
    :documentation "An indication of whether the calling function
     supplied the argument"
    :accessor optional-parameter-supplied-p
    :initarg :supplied-p
    :initform nil)
   (optional-parameter-type
    :type (or symbol list)
    :documentation "Non-specializing type information"
    :accessor optional-parameter-type
    :initarg :type
    :initform t))
  (:documentation "Information pertaining to a given &optional parameter"))

(defclass standard-optional-parameter (optional-parameter) ()
  (:documentation "Information pertaining to a given parameter"))

(defclass standard-rest-parameter (parameter) ()
  (:documentation "Information pertaining to a given &rest parameter"))

(defclass standard-keyword-parameter (optional-parameter)
  ((keyword-parameter-keyword
    :documentation "For when the keyword is different from the parameter name"
    :accessor :keyword-parameter-keyword
    :initarg :keyword))
  (:documentation "Information pertaining to a given &key parameter"))
```

Figure 1: Generic function and method parameter metaobjects would behave as if specified by these class definitions. Metaobject classes whose names begin with *standard* are instantiable.

```
(defmethod+ foo ((bar-param
                  :specializer float
                  :documentation "bar parameter documentation"
                  :unit-of-measure :kilopascals)
                 &optional
                 (baz-param
                  :type float
                  :default bar-default-value
                  :supplied-p  bar-supplied-p-value
                  :documentation "bar documentation"
                  :unit-of-measure :kilopascals)
                 &key
                 (blat-param
                  :type float
                  :keyword :blat-actual-keyword
                  :default blat-default-value
                  :supplied-p blat-supplied-p-value
                  :documentation "blat documentation"
                  :unit-of-measure :joules))
  ...)
```

Figure 2: An example use of the proposed *defmethod*-like macro. Parameter attributes are specified in CLOS *defclass* slot definition fashion. CLOS users not needing or wishing this much representational power would simply use *defmethod* in the traditional manner.

3 Motivations

3.1 Adding Parameter Metadata

3.1.1 Parameter Documentation

Perhaps the most commonly felt desire for more parameter representational power is the desire to be able to add runtime accessible documentation to GF and method parameters. Although such information could be placed somewhere in a modification of the existing lambda list, if the parameters were presently implemented as objects, these objects would be a convenient place for the parameter documentation to reside. Although this first point is admittedly not a powerful argument, it should be noted that having one more piece of parameter related information makes the notion of a more powerful data structure such as an object more palatable.

3.1.2 Metaobjects and Open Implementation

The advantages of using metaobjects revolve chiefly around their provided ability to (i) intuitively navigate, query, and augment a runtime semantic net of CLOS program meta-information, and to (ii) override or augment any advertised default internal GF interface behavior of a CLOS program. This is true provided that the GF is specialized on at least one metaobject. It is accomplished by the subclassing of metaobjects and the *advising* of advertised GF's with methods specialized on the new metaclass subclasses. The reader is referred to the AMOP [2] and open implementation work by Kiczales [5] for a defense of merits of the type of object based open implementation found in the MOP. Because of the strict manner in which parameter metaobjects adhere to both the form and the spirit of the MOP, the merits of parameter metadata rise or fall with the overall merits of the MOP.

An example of where the use of parameter-metaobject-based open implementation philosophy could be put to good use is in case of CLIM. Had parameter metaobjects existed, they may have influenced the design of CLIM. Had this been the case, CLIM could have been written to include GF's that specialize on parameter metaobjects for virtually any internal functionality that deals with parameter metadata. By taking advantage of such a use of open implementation philosophy and parameter metaobjects, would-be CLIM extenders could add new parameter constraints such as Harlequin's new :VIEW constraint. This would be accomplished by (i) subclassing the appropriate parameter metaobject and adding the :VIEW constraint as a slot to the new subclass; then by (ii) adding extending or overriding methods to advertised

CLIM internal GF's to get the expected :VIEW behavior. As the open implementation camp stresses, CLIM user/extenders could greatly augment and/or extend CLIM's behavior without possessing source code.

3.2 Enhancing Power in Generic Dispatch

3.2.1 Passing Metadata Around in a CLOS Environment

This paper's final argument for parameter metaobjects lies in the ability to provide more sophisticated forms of generic dispatch. The paper proceeds to lay out a brief description of what will be referred to as the metadata-centric environment. Then, with this foundation, it sets forth parameter metaobjects as one means of directly enhancing the power of this environment.

3.2.2 Basic Metadata Manipulation

Multiple options presently exist for maintaining objects with slot-instance-specific metadata such as the unit of measure of the slot's contents [3] [4]. An object's slot can be queried for this metadata and in the case of unit of measure data, a program could use this metadata to convert the slot's contents to another unit of measure. This sort of knowledge flow can be exemplified by the following simple step by step scenario:

- The value of a slot named *length* is accessed and stored in a lexical variable.
- The associated unit of measure value of slot *length* is accessed and likewise stored in a lexical variable.
- Some arithmetic manipulation is performed on the *length* value.
- This new *length* related value originally obtained from the slot is stored in a second slot.
- The associated unit of measure value is associated with the second slot.

As the scenario suggests, the function or method involved must manage both the data and the metadata item. Furthermore, a slot may also have several other pieces of metadata such as slot contents reliability, former slot states, or the source of the slot information. Transferring this metadata can clearly become an onerous task for the application programmer.

3.2.3 Passing Metadata Through Generic Dispatch

Perhaps an even more cumbersome task arises when attempting to pass this slot data and accompanying metadata to a generic function. Two options present themselves - either (i) the creation of extra generic function parameters for the metadata or (ii) the creation of objects to consolidate the value and its metadata. The latter option allows the simple passing of objects to the generic function in place of the values, while creating extra parameters would quickly muddy up a clear, concise generic function interface. Consolidated objects, on the other hand, would deny the application programmer the ability to easily utilize Common Lisp functions. For example, the user would have to first (*i*) extract the value from the consolidated object, (*ii*) hand the value off to the Common Lisp function, and (*iii*) perhaps return the value back to the consolidated object. Simply stated, a consolidated object representing an integer could not be passed to the Common Lisp *inc* function. While the consolidated object solution would solve the parameter passing problem, the use of the values embodied in these consolidated objects would be needlessly difficult. It should been pointed out that if Common Lisp used GF interfaces for its functions, this problem largely would not exist. An extending or overriding method on a Common Lisp GF could extract the information from the consolidated object.

In the absence of a more object oriented Common Lisp, the author chose to solve the latter loading/unloading problem by hiding variable attributes, such as units of measure in program variable property lists. In this system, generic functions could be invoked as before with variables as arguments, but their associated metadata would be seamlessly and invisibly passed through and associated with the called function's parameters. This metadata would therefore be available inside of the called function. It would be likewise hidden in the property lists of the formal parameters. Generic dispatch would, as always, specialize on the Lisp objects being passed. In short, metadata would unobtrusively accompany their respective data items throughout the program.

3.2.4 The Parameter Metaobject Tie-in

Rather that attempting to illicit agreement on any of the particulars of a metadata-centric environment or explaining it in any great detail, this paper is simply attempting to suggest that passing metadata through generic dispatch is a desirable goal. Assuming the presence of such a means for passing values and their metadata to a generic function, the opportunity arises to perform non-standard forms of generic dispatch by means of accessing this metadata. For example, an application programmer may wish for a form of generic dispatch that would first convert the argument's unit of measure to that of the generic function, and then proceeding with standard generic dispatch. Other more exotic future forms of generic dispatch may insist that arguments must be above a certain accuracy threshold or that the data had been gathered by a certain reputable type of measuring device.[6] Presently the MOP provides no place to house such constraint information. Since such constraints are clearly associated with the individual parameters, the parameter metaobjects, if they existed, would provide a structured means of associating metadata such as unit of measure with the parameter method metaobject. Application programmers could then specify unit of measure information or any other desired metadata for parameters in all cases where extended parameter-constraint-based generic dispatch was wanted.

4 Parameter Metaobject Alternatives

The CLIM implementations stand as living proof that it is non only possible, but feasible to successfully implement CLOS systems with an abundance of complex parameter related information without the benefit of MOP-based parameter metaobjects. These implementations used their own mechanisms for associating parameter metadata with parameters. This *other available means* argument could, be used against parameters metaobjects, but it could just as easily be employed against every metaobject of the existing MOP. Although pre-MOP CLOS implementations also survived without metaobjects, this paper, nevertheless, has argued that metaobjects offer an improved *quality of life* beyond the minimal *survival* of closed implementation or open implementation lacking the mentioned benefits of exposure of key internal user subclassable classes.

5 Conclusion

Although the majority of CLOS users would probably be quite happy without parameter metaobjects, the same could certainly be said of the rest of the standard metaobjects. Metaobjects are there to act a queryable state-keepers for the CLOS environment and to allow CLOS augmentations and extensions. This paper simply asserts that parameter metaobjects belong in the future ANSI MOP. Their presence can be easily ignored by the majority of CLOS users and richly exploited by those who wish to define and utilize the additional power of parameter metadata.

References

[1] Steele, G. *Common Lisp the Language (Second Edition)*, Digital Press, U.S., 1990.

[2] Kiczales, G., Rivieres, J., Bobrow, D., *The Art of the Metaobject Protocol*, The MIT Press, Cambridge, MA, 1993.

[3] Mato Mira F., "ECLOS: An Extended CLOS" in *Object-Oriented Programming in Lisp: Languages and Applications Workshop ECOOP 93*, Kaiserslautern, Germany, July 1993.

[4] Peterson, E., "Dynamic Persistent Metadata: A Metaobject Protocol Based Approach to Increasing Power in Knowledge Representation" in *1995 Association of Lisp Users Workshop Proceedings*, Cambridge, MA., 1995.

[5] G. Kiczales, "Towards a new model of abstraction in software engineering" in *Proceedings of the International Workshop on Reflection and Meta-Level Architecture*, November 1992.

[6] At present, there is no *method-combination-class* GF for returning a method combination metaobject given a quasi-*standard-class* metaobject. This GF and other open implementation method combination GF's would be necessary for such exotic generic dispatch implementations to be consistent with other parts of the MOP.