`make-method-lambda` considered harmful

Pascal Costanza, Charlotte Herzeel
Vrije Universiteit Brussel, Belgium

June 11, 2008

Abstract

The CLOS Metaobject Protocol (CLOS MOP) is a specification of how major building blocks of CLOS are implemented in terms of CLOS itself. This enables programmers to subclass meta-level classes and define meta-level state and behavior in an incremental fashion. The benefits of such a meta-level architecture for object systems in general and CLOS in particular are well documented. However, some parts of the CLOS MOP are underspecified or impractical to use. We discuss a particular dark corner of the CLOS MOP, the meta-level function `make-method-lambda`, whose purpose is to influence the expansion, and thus the semantics, of `defmethod` forms. We also make concrete suggestions for an alternative design for achieving the functionality that `make-method-lambda` provides, without any of its drawbacks.

1 Introduction: The CLOS MOP

ANSI Common Lisp specifies the Common Lisp Object System (CLOS), an object-oriented extension of Common Lisp that supports classes with multiple inheritance, generic functions as containers for methods, multiple dispatch, default and user-defined method combinations, and support for redefining many aspects of class definitions, method definitions and instances at runtime [1]. The CLOS Metaobject Protocol (CLOS MOP) is a specification of how major building blocks of CLOS are implemented in terms of CLOS itself. This enables programmers to subclass meta-level classes and define meta-level state and behavior in an incremental fashion [5].

The benefits of such a meta-level architecture for object systems in general and CLOS in particular are well documented. Examples include: dynamic slots [7], persistence frameworks [6], extensions for context-oriented programming [2], and so on. However, some parts of the CLOS MOP are underspecified or impractical to use. The reason for this is that the CLOS MOP specification was work in progress at the time of its publication in 1991: Major parts were already well understood and used in practice, while the use of other parts remained vague and unclear. This paper discusses a particular dark corner of the CLOS MOP, the meta-level function `make-method-lambda`, whose purpose is to influence the expansion, and thus the semantics, of `defmethod` forms.

2 How to make method lambdas

2.1 The role of `make-method-lambda`

The CLOS MOP is broadly divided into two parts: One part specifies how the user interface macros `defclass`, `defgeneric` and `defmethod` are expanded into calls of the underlying functional abstractions of the CLOS MOP, and the other part specifies the various subprotocols that provide hooks for influencing the semantics of CLOS. While the expansion for `defclass` and `defgeneric` are relatively straightforward mappings to invocations of `ensure-class` and `ensure-generic-function`, the expansion of `defmethod` forms is more involved. Especially, it is specified that method bodies have to be passed to, and preprocessed by, `make-method-lambda`. The role of `make-method-lambda` is to ensure that arguments passed to a generic function are made available to the method body, and to insert further local definitions into method bodies. For example, local definitions for `call-next-method` and `next-method-p` are inserted by default, whose implementations fetch the internal list of next methods.

In order to be able to get access to the different parameters to method bodies, which are not necessarily only the arguments passed to the corresponding generic function, a method body lambda expression is therefore converted into a modified lambda expression that can take additional parameters. For example, assume that the following method is defined.

```
(defmethod foo ((x integer) (y integer))
  (do-something x y))
```

The corresponding method body is an unspecialized lambda form.

```
(lambda (x y)
  (do-something x y))
```

This lambda form (the s-expression, not the closure it would evaluate to!) is passed to `make-method-lambda` and converted into roughly something like this.

```
(lambda (args next-methods)
  (let ((x (car args)) (y (cadr args)))
    (flet ((call-next-method ()
             (funcall (method-function (car next-methods))
                      args (cdr next-methods)))
           (next-method-p () (not (null next-methods))))
      (do-something x y))))
```

Industrial-strength CLOS implementations perform a number of optimizations and thus yield different lambda forms that vary in a number of details, but the overall idea remains the same.

The function `make-method-lambda` is a generic function and is specified to take the following parameters: A generic function metaobject for which a method is being defined, a possibly uninitialized method metaobject (as, for example, returned by `class-prototype` on some method metaobject class), a method body lambda expression and an environment object. Methods on `make-method-lambda` return a new lambda expression and possibly a number of initialization arguments for creating a new method object. The full expansion of the above `defmethod` corresponds to something like this.[1]

[1] The environment parameter is not further explained here.

```
(let ((gf (ensure-generic-function 'foo)))
  (multiple-value-bind
      (lambda-expression extra-initargs)
      (make-method-lambda
        gf (class-prototype (generic-function-method-class gf))
        '(lambda (x y) (do-something x y))
        lexical-environment-of-defmethod-form)
    (add-method gf (apply #'make-instance
                    (generic-function-method-class gf)
                    :qualifiers '()
                    :lambda-list '(x y)
                    :specializers (list (find-class 'integer)
                                        (find-class 'integer))
                    :function (compile nil lambda-expression)
                    extra-initargs))
```

The function `make-method-lambda` is quite powerful: User-defined methods on `make-method-lambda` can insert their own lexical definitions into method bodies (like `call-previous-method` or `current-method` [5]), and can make method functions accept more parameters. The CLOS MOP specifies a corresponding extension to user-defined method combinations, where `call-method` forms can pass additional arguments to method functions. If a user-defined method combination and a method on `make-method-lambda` are paired correctly, they can cooperate to take advantage of such additional arguments and actually make extensions like `call-previous-method` and `current-method` work. However, there is a major drawback in this approach, which is discussed in the next subsection.

2.2 Dependency on generic functions

A user can specialize `make-method-lambda` on its generic function or method parameter. Specializations on the lambda expression parameter and the environment are not useful for practical purposes, since the lambda expression should always be a cons cell that makes up an s-expression, and the environment parameter is an environment object, whose class is left unspecified in ANSI Common Lisp. We are not aware of any Common Lisp implementation that supports subclassing of cons cells or environments, so we are left with specializing `make-method-lambda` on the other two parameters.

Next, ANSI Common Lisp does not specify how to associate method classes with `defmethod` forms. Only default method classes can be specified for generic functions as the `:method-class` option in `defgeneric` forms (or as the corresponding keyword parameter to `ensure-generic-function`), which specify the method class which is used by default for any method defined on such a generic function. So for most practical purposes, which `make-method-lambda` method gets to preprocess method bodies depends only on the generic function to which a method is supposed to added.[2]

This implies two problems: The generic function metaobjects have to be present at macroexpansion time, and the coupling between generic functions and methods is too tight.

[2]The lack of a possible `:method-class` option in `defmethod` forms is arguably an omission.

2.3 Required presence of generic function metaobjects

As an example, assume the following definitions.

```
(defgeneric foo (x y)
  (:method-class my-method))

(defmethod foo ((x integer) (y integer))
  (do-something x y))
```

Executing these definitions has two different results, depending on whether the code is evaluated directly, or is preprocessed using a file compiler. With direct evaluation, the first form gets expanded and immediately executed to create a generic function metaobject with an associated special method class. Then, the second form gets expanded, where the expansion is potentially modified by a user-defined method on `make-method-lambda` specialized on `my-method` (for example, for inserting additional lexical definitions into the method body). After expansion, the second form gets executed to create a method metaobject and add it to the generic function metaobject yielded by the first form.

When the code is processed by a file compiler, things will be different: There, the first form gets expanded but not immediately executed. Instead, execution of the expanded code is delayed until load time, as is always the case for file-compiled code in ANSI Common Lisp. This implies that the corresponding generic function metaobject is not created yet. Thus, when the second form gets expanded, a user-defined method on `make-method-lambda` specialized on `my-method` will *not* be executed: Recall from above that the corresponding generic function metaobject is obtained by `ensure-generic-function`, which is specified to either return an existing generic function with the given name, or to create a new generic function with default settings in case it does not exist yet. One of the default settings is that the default method class should be `standard-method`, and thus a `make-method-lambda` on `my-method` will not be taken into consideration in this case. This situation is further complicated in that after a load of the thus compiled code, the corresponding generic function metaobject `foo` will have a method class option `my-method`, and after a subsequent recompilation, the correct `make-method-lambda` method will be executed. Taken together, this means that the processing of such code does not only lead to surprising, unexpected bugs, but that it is also unnecessarily hard to debug such code, because on recompilation, bugs triggered by this surprising behavior will simply disappear.[3]

A workaround in practice is to embed `defgeneric` forms in appropriate `eval-when` forms that ensure that the generic functions are already created at compile time. For example, a correct version of the code above looks like this.

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (defgeneric foo (x y)
    (:method-class my-method)))

(defmethod foo ((x integer) (y integer))
  (do-something x y))
```

[3] We have painful experience with long debugging sessions caused by this problem.

Here, `:compile-toplevel` indicates that the `defgeneric` form is already executed at compile time, `:load-toplevel` indicates that it is also executed at load time (when we actually want it in the first place), and `:execute` indicates that we also want it executed when directly evaluated.

Another workaround is to place the `defgeneric` and the `defmethod` forms in separate files and instruct a system definition utility like `asdf` or `mk-defsystem` to load the compiled file with the `defgeneric` form before the other one is compiled. However, this can be unnatural, for example when the corresponding methods are default methods for the corresponding generic function, or should be part of the same source file for some other reasons.

2.4 Coupling between generic functions and methods

Conceptually, methods are entities independent of particular generic functions. For example, it should be possible from a conceptual point of view to remove a particular method from one generic function and add it to another one. While this is rare, it can be quite useful, for example to programmatically generate methods on generic functions at runtime. Consider the following example.

```
(defun add-reader-filter (gf filter)
  (let ((method (defmethod dummy-reader :around (object)
                  (funcall filter (call-next-method)))))
    (remove-method (fdefinition 'dummy-reader) method)
    (add-method gf method)))
```

Here, a method is created on an intermediate generic function, and then immediately removed from it and added to another one. However, problems can occur if the generic function to which the filter method should be added requires a modification of the method body that deviates from the one specified for `dummy-reader`. This effectively means that this technique for programmatically creating methods at runtime is feasible only in very controlled cirumstances, where the classes of the generic functions involved and their specializations of `make-method-lambda` are known in advance.

2.5 State of `make-method-lambda`

One could object that programmatic method creation should not depend on `defmethod`, but should use the corresponding subprotocols of the CLOS MOP, so a call to `make-method-lambda`, a `compile` on its result, a creation of the method class of the corresponding generic function, and the addition of the thus created method to that generic function, as already sketched above.

However, due to the problems discussed above, the corresponding subprotocols of the CLOS MOP and especially `make-method-lambda` are not widely supported in Common Lisp implementations. Currently, only SBCL supports it as specified, and LispWorks supports a slight variation of `make-method-lambda`. Additionally, Allegro Common Lisp, Clozure Common Lisp / Macintosh Common Lisp / OpenMCL, and LispWorks do not take processed parameters that allow passing of additional parameters, like a list of next methods, but only the original parameters of the corresponding `defmethod` form. This has some advantages in programmatic method creation, as shown in the following hypothetical variation of `add-reader-filter`.

```
(defun add-reader-filter (gf filter)
  (add-method gf (make-instance 'standard-method
                   :qualifiers '()
                   :lambda-list '(object)
                   :specializers (list (find-class 't))
                   :function (lambda (object)
                               (funcall filter (call-next-method))))))
```

Note that the method body is not processed anymore but can actually be a 'real' closure, whereas in the approach proposed by the CLOS MOP, method bodies must be s-expressions that are processed by `make-method-lambda` and then explicitly compiled into method functions. An implication is that such method functions cannot close over local lexical environments, which can be useful, as this example shows. On the other hand, however, the example uses `call-next-method` which would need to be inserted by `make-method-lambda`, so this way of expressing `add-reader-filter` is not correct either. So implementations that do not take processed parameters in method functions have the advantage of being able to use lexical closures as method functions in a straightforward way, but cannot take advantage of additional lexical definitions in method bodies.

Because of these incompatibilities and problems with `make-method-lambda` across several Common Lisp implementations, this way of modifying method bodies is unlikely to be widely used. Instead, new user-defined 'kinds' of method bodies can be introduced in a much more straightforward way by defining one's own method-defining macros, as in the following example.

```
(defmacro define-filter-method (name filter)
  `(defmethod ,name :around (.object.)
     (funcall ,filter (call-next-method))))
```

This way of introducing a filter method does not allow for programmatic creation of filter methods, but avoids any of the problems mentioned above. Especially the differences between evaluated and file-compiled code are avoided because by default, macros do not depend on the presence or absence of objects in the compilation environment. While this seems to work well in practice more often than not, it misses the ability to pass extra arguments to method bodies, like lists of next or previous methods as discussed above.

3 An alternative design

Instead of proposing fixes to `make-method-lambda`, let us first take a look at the functionalities we expect from it. As we have seen above, `make-method-lambda` serves two purposes: One can add new lexical definitions inside method bodies, and one can ensure that method functions can receive additional parameters. Introducing one's own method-defining forms can serve the first purpose, but they are not expressive enough to serve the second purpose. We propose an alternative way of achieving the same functionality and expressiveness while avoiding the problems of `make-method-lambda`.

Our proposal consists of two elements.

1. We base our approach on the current practice of introducing one's own method-defining forms in order to avoid any ambiguities of evaluated versus file-compiled code.

2. Additionally, we change the signature of method functions such that they can *always* receive additional arguments.

Let us focus on the second element: In fact, Common Lisp already provides a mechanism to receive arbitrary parameters in the form of keyword arguments. Consider the creation of a method function that expects an additional current method parameter. Such a method function can look like this.

```
(make-instance 'standard-method
  :qualifiers '()
  :lambda-list '(object)
  :specializers (list (find-class 't))
  :function (lambda (args next-methods
                     &key current-method
                     &allow-other-keys) ...))
```

A corresponding user-defined method combination that provides the current parameter as an extra argument to the method body can look like this.

```
(define-method-combination standard/current ()
  ((around (:around))
   (before (:before))
   (primary () :required t)
   (after (:after)))
  (flet ((call-methods (methods)
           (loop for method in methods
                 collect `(call-method
                           ,method ()
                           :current-method ,method))))
    (let ((form (if (or before after (rest primary))
                    `(multiple-value-prog1
                       (progn ,@(call-methods before)
                              (call-method
                                ,(first primary) ,(rest primary)
                                :current-method ,(first primary)))
                       ,@(call-methods (reverse after)))
                    `(call-method ,(first primary) ()
                       :current-method ,(first primary)))))
      (if around
          `(call-method
             ,(first around) (,@(rest around) (make-method ,form))
             :current-method ,(first method))
          form))))
```

Note that this is exactly like the definition for the default method combination, as given as an example in ANSI Common Lisp, but with every `call-method` form extended by an argument for the keyword parameter `:current-method`.

Our approach provides the following advantages.

- There is no need for providing hooks in the CLOS MOP for modifying the expansion of method bodies anymore. Especially, any ambiguity with regard to a distinction between evaluated and file-compiled code is avoided. Since `make-method-lambda` is the only such hook in the CLOS MOP specification, the CLOS MOP would avoid any such hooks for modifying macroexpansions, providing CLOS implementors more freedom in handling and optimizing them.

- Users can still introduce their own method-defining macros as before, which is currently the only reliable way to achieve modifications of method bodies anyway.

- Additionally, users can rely on a unified protocol for passing extended parameters to method functions. Methods can thus now, in principle, be added to arbitrary generic functions, because the processing of the method bodies and the handling of extra parameters does not depend on generic functions anymore. There are still dependencies on generic functions, though, in that method bodies may depend on whether specific extra parameters are passed by the generic function or not in the first place. However, methods now have better ways to gracefully deal with missing extra parameters: Keyword arguments can have default values, can be checked whether they have been explicitly passed or not, and `&allow-other-keys` can indicate whether unrecognized keyword arguments are acceptable or not, as is all usual for keyword argument processing in Common Lisp itself.

- Finally, method functions can be closures, which is not the case with the current design proposed by the CLOS MOP. With our proposed change, `add-filter-method` can be implemented like this.

```
(defun add-reader-filter (gf filter)
  (add-method gf (make-instance 'standard-method
                   :qualifiers '()
                   :lambda-list '(object)
                   :specializers (list (find-class 't))
                   :function (lambda (args next-methods
                                      &rest more-args)
                               (funcall filter
                                 (apply (method-function
                                           (first next-methods))
                                        args (rest next-methods)
                                        more-args))))))
```

Our approach also has some disadvantages:

- There may be a performance overhead when processing keyword parameters: As is already the case in Common Lisp, keyword arguments are passed as lists, which need to be traversed to fetch the correct arguments. However, user-defined extensions of handling generic functions and methods already imply an overhead since CLOS implementations cannot exploit certain assumptions about their semantics anymore. We believe that

the increased expressive power and the removal of semantic ambiguities and hard-to-find bugs justifies the price to pay. CLOS implementations should still be able to apply their usual optimizations in non-user extended code. For example, their method functions may simply reject additional keyword parameters and thus do not need to incur any overhead for processing them.[4]

- Current CLOS implementations that use the unchanged argument list from the method definitions instead of the extended ones that can accept additional parameters are incompatible with our proposed approach. Adopting our approach would thus render some existing CLOS MOP user code invalid. However, presumably they use the original paramter lists because this enables the use of 'real' closures as method functions. Our approach combines the expressiveness of all the approaches in use so far, including support for 'real' closures, so we are convinced that it should be worthwhile to adopt it in spite of incompatibilities with old code. (It should also be possible to provide an API that selectively provides backward compatibility with any of the old approaches.)

4 Discussion

The current CLOS MOP specification is a result of an evolution of several design alternatives for its various subprotocols. In [8], a previous design for processing method bodies is described that does not rely on influencing their (macro)expansion, but rather on influencing their interpretation. So instead of defining methods on `make-method-lambda`, one would define methods on `apply-method`. For example, one could add extra arguments to method functions like this.

```
(defmethod apply-method ((gf my-generic-function)
                         (method my-method)
                         args next-methods)
  (funcall (method-function method)
           args next-methods method))
```

By relying on hooking into the interpretation of method bodies, the ambiguities between evaluated and file-compiled code are avoided. However, this older design has the problem that method bodies can, in the general case, not be fully compiled and executed without meta-level intercession at runtime anymore, but instead method execution always has to go through `apply-method`. By switching to `make-method-lambda`, a more efficient implementation is achieved that completely removes any runtime intercession.

Nevertheless, `make-method-lambda` still has serious drawbacks which are inherited from the `apply-method` approach. Essentially, one cannot tell by looking at a `defmethod` form what its eventual behavior will be, because a (potentially) arbitrary method on `apply-method` / `make-method-lambda` can modify or even completely replace the method body in question. The advantage of one's own method-defining macros is that it is syntactically always clear

[4] This may actually lead to a situation where both `make-method-lambda` and our suggested approach can coexist. We are currently investigating this option.

from the `define-...-method` call which other part of the code influences the expansion of the method body.

In our own work on reconstructing 3-Lisp [4], a reflective language which heavily influenced the design of the CLOS MOP, we have found similar problems and similar fixes. This paper is a practical outcome of our findings by proposing what we think of as a concrete improvement of the CLOS MOP based on that experience.

We are not the first ones to discuss the drawbacks of `make-method-lambda`. For example, Bruno Haible discussed them before using a concrete example in a Usenet posting [3]. Our contribution are a more detailed discussion and a concrete suggestion for an alternative design.

Acknowledgments We thank the anonymous reviewers for their fruitful comments which greatly improved this paper.

References

[1] ANSI/INCITS X3.226-1994. *American National Standard for Information Systems - Programming Language - Common Lisp*, 1994.

[2] Pascal Costanza, Robert Hirschfeld, Wolfgang De Meuter. Efficient Layer Activation for Switching Context-dependent Behavior. *Joint Modular Languages Conference 2006*, Proceedings, Springer LNCS.

[3] Bruno Haible. make-method-lambda ill-designed. Usenet posting, `d3gmvb$1ri$1@laposte.ilog.fr`.

[4] Charlotte Herzeel, Pascal Costanza, Theo D'Hondt. Reflection for the Masses. Workshop on Self-Sustaining Systems (S3), Potsdam, Germany, May 15-16, 2008. Springer LNCS (to appear).

[5] Gregor Kiczales, Jim des Rivières, Daniel Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[6] Heiko Kirschke. *Persistenz in objekt-orientierten Programmiersprachen*. Logos Verlag, Berlin, 1997.

[7] Andreas Paepcke. User-Level Language Crafting – Introducing the CLOS Metaobject Protocol. In: Andreas Paepcke (ed.), *Object-Oriented Programming: The CLOS Perspective*, MIT Press, 1993.

[8] Jim des Rivières. The Secret Tower of CLOS. 1990 OOPSLA/ECOOP Workshop on Reflection.