`make-method-lambda` revisited

Irène Durand
Robert Strandh
irene.durand@u-bordeaux.fr
robert.strandh@u-bordeaux.fr
LaBRI, University of Bordeaux
Talence, France

ABSTRACT

The Common Lisp [1] metaobject protocol [3] specifies a generic function named `make-method-lambda` to be called at macro-expansion time of the macro `defmethod`. In an article by Costanza and Herzeel [2], a number of problems with this generic function are discussed, and a solution is proposed.

In this paper, we show that the alleged problems are due to the fact that existing implementations do not include proper compile-time processing of the associated macro `defgeneric`, and that with proper compile-time processing, the problems indicated in the paper by Costanza and Herzeel simply vanish.

The main characteristic of our proposed solution is for the compile-time side effects of `defgeneric` to include saving the name of the method class given as an option to that macro call. With this additional information, no difference exists between the behavior of direct evaluation and that of file compilation of a `defgeneric` form and a `defmethod` form mentioning the same name of the generic function.

CCS CONCEPTS

• **Software and its engineering → Abstraction, modeling and modularity**; **Software performance**; **Compilers**;

KEYWORDS

Common Lisp, Meta-Object Protocol

ACM Reference Format:
Irène Durand and Robert Strandh. 2019. `make-method-lambda` revisited. In *Proceedings of the 12th European Lisp Symposium (ELS'19)*. ACM, New York, NY, USA, 5 pages.

1 INTRODUCTION

In the definition of the Common Lisp [1] metaobject protocol in the book by Kiczales et al [3] (also known as the AMOP), the generic function `make-method-lambda` plays a role that is very different from most of the other generic functions that are part of the metaobject protocol.

ELS'19, April 1–2 2019, Genoa, Italy

ACM ISBN 978-2-9557474-2-1.

According to the book, the function has four parameters, all required:

(1) A generic function metaobject.
(2) A (possibly uninitialized) method metaobject.
(3) A lambda expression.
(4) An environment object.

The main difference between `make-method-lambda` and other generic functions defined by the metaobject protocol is that `make-method-lambda` is called as part of the expansion code for the `defmethod` macro, whereas other generic functions are called at execution time.

The AMOP book states that the generic function passed as the first argument may be different from the one that the method is ultimately going to be added to. This possibility seems to exist to handle the situation where a `defgeneric` form is followed by a `defmethod` form in the same file. In this situation, the Common Lisp standard clearly states that the file compiler does not create the generic function at compile time. Therefore, when the corresponding `defmethod` form is expanded (and therefore `make-method-lambda` is called), the generic function does not yet exist. It will be created only when the compiled file is loaded into the Common Lisp system.

The AMOP book also states that the method object passed as second argument may be uninitialized, suggesting that the *class prototype* of the method class to be instantiated may be passed as the second argument.

The third argument is a lambda expression corresponding to the body of the `defmethod` form. The purpose of `make-method-lambda` is to wrap this lambda expression in another lambda expression called the *method lambda* which is ultimately compiled in order to yield the *method function*.

The default method lambda returned by an invocation of `make-method-lambda` is a lambda expression with two parameters. The first parameter is a list of all the arguments to the generic function. The second parameter is a list of next methods that can be invoked using `call-next-method` from the body of the method. Therefore `make-method-lambda` also provides definitions of `call-next-method` and `next-method-p` that are lexically inside the lambda expression it returns.

It is important that the method lambda is returned as part of the expansion of the `defmethod` macro and that it is then processed in the same environment as that of the `defmethod` form itself, so that when the `defmethod` macro call is evaluated in an environment that is not the *null lexical environment*, that environment is taken into account when

the method lambda is processed. For example, code like this one:

```
(let ((x 10))
  (defmethod foo ((y integer))
    (+ x y)))
```

should work as expected.

Finally, the fourth argument to `make-method-lambda` is an environment object.

2 PREVIOUS WORK

In their article [2], Costanza and Herzeel give a simple example of this simple `defmethod` form:

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

and at the end of section 2.1, on page 3, they claim that the expansion of that form is "something like" the follow form:

```
(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))
```

except that we have formatted the code to fit the page, and we have added two missing closing parentheses at the end of the form.

This example is a slight variation on the code that is shown in section 5.5.1 of the AMOP book. However, in that section, it is not claimed that this code is the result of expanding a `defmethod` form. Rather, it is given as "an example of creating a generic function and a method metaobject, and then adding the method to the generic function".

Indeed, this expansion is not possible, at least not in a compiling implementation, which is the premise of both the paper by Costanza and Herzeel and this one. It has two fundamental problems:

(1) The call to `make-method-lambda` must be made at macro-expansion time, whereas in their example, the call is present in the expansion, so it will be made at load time.
(2) In their example, the resulting method lambda is compiled in the null lexical environment. However, compiling in the null lexical environment would violate the semantics of the Common Lisp standard, which requires that the body of the `defmethod` form be compiled in the lexical environment in which it appears.

In section 5.4.3 of the AMOP book, an example of an expansion is shown, and figure 5.4 clearly mentions that `make-method-lambda` is called during the macro-expansion phase. Furthermore, in figure 5.3, which shows the expansion of the `defmethod` macro, no call to `compile` is made. The result of calling `make-method-lambda`, i.e., the *method lambda* is simply present in the expanded code.

As Costanza and Herzeel point out, the `defmethod` macro does not allow the programmer to specify a class for the method to be created. That class must be determined by the generic function to which the method is ultimately going to be added. Therefore, in the case of a `defgeneric` form followed by a `defmethod` form, the method class must be the one indicated in the `defgeneric` form.

The conundrum, then, is that the file compiler does not create the generic function as a result of compiling the `defgeneric` form, so when a `defmethod` form with the same name is encountered later in the same file, the method class can not be taken from the generic function metaobject. Otherwise, the normal way of obtaining the method class would be to call the accessor `generic-function-method-class`, passing it the generic function metaobject with the name indicated in the `defmethod` form. If there is no way for the file compiler to determine the method class when the `defmethod` form is encountered, then clearly the only choice is to call `make-method-lambda` with the class prototype of the class named `standard-method` as the second argument. However, the analysis by Costanza and Herzeel is that this behavior is a result of the file compiler calling `ensure-generic-function` to obtain a generic-function metaobject and then querying that object to obtain the method class. A simple experiment shows that this is not the case in SBCL for instance.

When the following code is compiled with the SBCL file compiler:

```
(defclass hello (standard-method) ())

(defgeneric foo (x y)
  (:method-class hello))

(defmethod foo (x y)
  (+ x y))

(eval-when (:compile-toplevel)
  (print (fdefinition 'foo)))
```

the compilation fails when an attempt is made to find the definition of `foo` in the last top-level form. Thus, after the `defmethod` form has been compiled, the generic-function metaobject still does not exist in the compilation environment.

However, tracing `make-method-lambda` prior to compiling the code above in a fresh compilation environment reveals that `make-method-lambda` is indeed called as a result of compiling the `defmethod` form, and that the second argument passed to the call is an instance of `standard-method`.

This situation can lead to some problems in client code that are amply described in the paper by Costanza and Herzeel. The essence of the problem is that, when a `defgeneric` form with a non-standard `:method-class` option is followed by a `defmethod` form in the same file, the file compiler may generate an expansion of the `defmethod` form that creates an instance of `standard-method` when the compiled file is ultimately loaded, as opposed to an instance of the method class with the name that was explicitly mentioned in the `defgeneric` form.

Furthermore, this behavior is inconsistent with the behavior when the source file is processed using `load`. The reason is that, contrary to the file compiler, `load` completely processes and evaluates each top-level form in order. As a result, when `load` is used, the generic function metaobject *is* created as a result of evaluating the `defgeneric` form, so that it *does* exist when the `defmethod` form is ultimately evaluated. Clearly, such inconsistent behavior between directly loading a source file and loading the result of applying the file compiler to it first is highly undesirable.

Perhaps even worse, even when the file compiler is used consistently, if the file is recompiled after having been loaded previously, the existing generic-function metaobject is reinitialized to have the correct method class, and the code works as when `load` is used.

The ultimate conclusion by Costanza and Herzeel is that, in order for the behavior of the file compiler to be consistent with that of loading the source file directly, and indeed for that behavior to be correct, the file compiler *must* create the generic function metaobject at compile time, so that it can be queried for the desired method class when the `defmethod` form is encountered. In the next section, we propose an alternative solution to this conundrum.

To solve the perceived problems with `make-method-lambda`, Costanza and Herzeel first analyze what desirable features this function has, and conclude that the following two are essential:

(1) It can add new lexical definitions inside method bodies. This is the feature that is used to introduce definitions of `next-method-p` and `call-next-method`.
(2) It can create lambda expressions for method functions with parameters in addition to the usual two, namely one for holding the arguments to the generic function and another for holding a list of next methods.

With these essential features in mind, Costanza and Herzeel then propose an alternative to `make-method-lambda` that does not have the perceived problem that this function has.

Their proposed solution has two parts:

(1) They use custom method-defining macros. Such a macro would expand to a `defmethod` form, but this form can contain additional lexical definitions into the method body, introduced by the custom macro.
(2) They propose that method functions always be able to take additional parameters in the form of Common Lisp keyword parameters. Furthermore, the use of the lambda-list keyword `&allow-other-keys` would make it easier to combine method functions that accept different additional arguments.

While it is able to solve the problem of the inconsistent behavior between `compile-file` and `load`, this solution has two major disadvantages as pointed out by Costanza and Herzeel:

(1) With this solution, method functions have a lambda list that includes keyword parameters. Processing keyword arguments imposes a significant performance penalty on the invocation of method functions.
(2) Existing CLOS implementations that use a lambda list without any keyword parameter for method functions are incompatible with this solution.

In the next section, we propose a solution that has neither of these disadvantages.

3 OUR TECHNIQUE

As permitted by the Common Lisp standard, the `defgeneric` macro may store information provided in the `defgeneric` form so as to make better error reporting possible when subsequent forms are compiled. In particular, the standard mentions storing information about the lambda list given, so that subsequent calls to the generic function can be checked for correct argument count. This information is kept in an implementation-specific format that does not contain the full generic-function metaobject, as this object is created when the compiled code resulting from the file compilation is loaded.

However, just as it is possible to keep information about the lambda list at compile time, it is also possible to keep information about the `:method-class` option given, or, when no option was supplied, the fact that the method class is `standard-method`.

With this additional information, during the expansion of the `defmethod` macro, the name of the method class can be retrieved, then a class metaobject from the name, and finally a class prototype from the class metaobject.

While the first parameter of `make-method-lambda` is indicated by the AMOP book as a generic-function metaobject, it is not specifically indicated that this object might be uninitialized, contrary to the method object that must be passed as the second argument. It is, however, indicated that the generic-function object passed as the first argument may not be the generic-function object to which the new method will eventually be added. Therefore, there is not much information that `make-method-lambda` can make use of. The exception would be the exact class of the generic function and the exact method class. It would be awkward for a method on `make-method-lambda` to access this information explicitly, rather than as specializers of its parameters. For that reason, the first argument to `make-method-lambda` might as well be a class prototype, just as the second argument might be.

As an example of how to accomplish this additional information, we suggest a solution with two parts:

(1) The first part involves the possibility for the compiler to store information about a generic function in the

compilation environment, as the result of compiling a `defgeneric` form. Specifically, the name of the generic-function class and the name of the method class would need to be stored, and later retrieved.

(2) The second part requires a modification to the protocol used by the compiler to query the compilation environment in order to determine how a form is to be compiled.

For a solution to the first part, in Appendix A we show how additional information about a generic function can be stored and retrieved in the context of the protocol described in Strandh's paper on first-class global environments [5].

For the second part, recall that in section 8.5 of the second edition of Guy Steele's book on Common Lisp [4], the author describes a protocol for this kind of environment query. This protocol contains three functions for environment query, namely `function-information`, `variable-information` and `declaration-information`.

Not only are these functions inadequate for all the information that a compiler needs to determine about a function or a variable, but they are also hard to extend in a backward-compatible way. A modern version of this protocol would likely return standard objects as opposed to multiple values, thereby allowing for backward-compatible extensions on a per-implementation basis.

For the second part of our solution, the environment function `function-information`, when given the name that has previously been encountered in a `defgeneric` form, would have to return information about the name of the generic-function class and the method class. With this additional information, the expander for the macro `defmethod` would query the environment for this information, access the corresponding classes, and then the class prototypes, and finally call `make-method-lambda` with those prototypes as arguments.

While our solution is an improvement on the existing situation, it is clearly not perfect. For one thing, both the generic-function class and the method class mentioned in the `defgeneric` form must exist when the `defmethod` form is encountered, so that the class prototypes of these classes can be passed as arguments to `make-method-lambda`.

Also, when a custom generic-function class is used, it is possible that there exist custom methods on various generic functions for initializing instances of this custom class, and these custom methods could conceivably intercept and alter the method class in the generic-function metaobject thus created. In such a situation, our technique would then use incorrect information about the method class, and pass the wrong class prototype as the second argument to `make-method-lambda`.

4 CONCLUSIONS AND FUTURE WORK

We have defined a technique that alleviates a problem encountered in current Common Lisp implementations when a `defgeneric` form is followed by a `defmethod` form in the same compilation unit. When the `defgeneric` form mentions a method class other than `standard-method`, and the compilation unit is processed in a fresh compilation environment, current implementations do not propagate the information about the method class to the macro expander for `defmethod`, resulting in `make-method-lambda` being called with a `method` argument of the wrong class.

Our solution requires the compiler of the Common Lisp implementation to store a small amount of additional information about the generic-function class and the method class when the `defgeneric` form is encountered, and requires the macro expander for `defmethod` to retrieve this information by querying the compilation environment.

Contrary to the proposal by Costanza and Herzeel, our suggested solution does not introduce any incompatibilities that would render some existing code obsolete. Furthermore, our solution does not have the potential performance problem of the proposal by Costanza and Herzeel, i.e. the additional cost of processing keyword arguments to method functions.

However, there are still some situations where our technique does not work. In particular, when a custom generic-function class is used, and the initialization of instances of this class intercepts and alters the information about the method class as given in the `defgeneric` form. For cases like this, we suggest that the author of the custom generic function class also add a method on `add-method`, specialized to the custom generic-function class so as to verify that the class of the method being added is indeed correct, and signal an error otherwise.

Future work includes adding the functions defined in Appendix A to the SICL[1] protocol for first-class global environments.

The Cleavir compiler framework which is part of SICL defines a modern version of the protocol for environment query defined in the second edition of Guy Steele's book [4]. We plan to extend this protocol to include information about the name of the generic-function class and of the method class given (explicitly or implicitly) in a `defgeneric` form previously encountered in the current compilation environment. Since our existing protocol returns standard objects, no modifications to the existing Cleavir code will be required as a result of this extension. The extension will allow us to define the macro `defmethod` in SICL to query the environment, and to invoke `make-method-lambda` with appropriate arguments.

A PROTOCOL

In this appendix we present the additional generic functions making up the protocol for our first-class global environments.

In order for our definitions to fit in a column, we have abbreviated "Generic Function" as "GF".

`function-class-name` *fname env* [*GF*]

This generic function returns the name of the class of the function associated with *fname* in *env*.

If *fname* is not associated with an ordinary function or a generic function in *env*, then an error is signaled.

[1] See https://github.com/robert-strandh/SICL

`(setf function-class-name)` *class-name fname env* [*GF*]
This generic function is used to set the class name of the function associated with *fname* in *env* to *class-name*.

If *fname* is associated with a macro or a special operator in *env*, then an error is signaled.

`method-class-name` *fname env* [*GF*]
This generic function returns the name of the method class of the function associated with *fname* in *env*.

If *fname* is not associated with a generic function in *env*, then an error is signaled.

`(setf method-class-name)` *class-name fname env* [*GF*]
This generic function is used to set the class name for methods of the function associated with *fname* in *env* to *class-name*.

class-name must be a symbol naming a class.

If *fname* is not associated with a generic function in *env*, then an error is signaled.

REFERENCES

[1] *INCITS 226-1994[S2008] Information Technology, Programming Language, Common Lisp*. American National Standards Institute, 1994.

[2] Pascal Costanza and Charlotte Herzeel. `make-method-lambda` considered harmful. Technical report, Vrije Univrsiteit Brussels, Belgium, 2008.

[3] Gregor Kiczales and Jim Des Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991. ISBN 0262111586.

[4] Guy L. Steele, Jr. *Common LISP: The Language (2Nd Ed.)*. Digital Press, Newton, MA, USA, 1990. ISBN 1-55558-041-6.

[5] Robert Strandh. First-class global environments in common lisp. In *Proceedings of the 8th European Lisp Symposium*, ELS '15, pages 79 – 86, April 2015. URL http://www.european-lisp-symposium.org/editions/2015/ELS2015.pdf.