Representing method combinations

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

ABSTRACT

The Common Lisp standard has few requirements on method combinations, and so does the semi-standard metaobject protocol for Common Lisp. For that reason, there is great variety among different Common Lisp implementations regarding how method combinations are represented and handled. Some implementations allocate a new method-combination instance for each generic function, whereas others attempt to reuse existing instances as much as possible. Most implementations are able to verify the validity of method-combination options for the built-in method-combination types, but no free Common Lisp implementation can verify custom method-combination types using the long form of the macro `define-methodcombination` immediately when a generic function is created, nor when a method-combination type is redefined. Instead, incompatibilities between supplied options and the method-combination type are then verified only when an attempt is made to execute the resulting method-combination procedure in order to create an effective method.

We propose a technique that makes early detection of incompatible method-combination options possible even for custom long-form method-combination types. We augment the lambda list of the method-combination definition with `&aux` entries that verify restrictions, and we construct a function with the augmented lambda list that will fail whenever there is such an incompatibility. With this technique, when an incompatibility is detected, we are also able to signal more relevant errors than most existing free implementations are able to do.

CCS CONCEPTS

• **Software and its engineering → Incremental compilers**; **Runtime environments**;

KEYWORDS

Common Lisp, CLOS, Meta-Object Protocol, Method combinations

ACM Reference Format:
Robert Strandh. 2021. Representing method combinations. In *Proceedings of the 13th European Lisp Symposium (ELS'20)*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.3747553

1 INTRODUCTION

The Common Lisp standard [1] contains very little information about method combinations. The dictionary entry in the standard for the system class `method-combination` requires a *method combination object* to be an *indirect instance* of the system class named `method-combination`. The standard further requires such an object to contain information both about the *type* of method combination and the *arguments* used with that type.

The term *indirect instance*, as explained in the glossary, excludes the possibility of such a method combination object to be an immediate instance of the class `method-combination`. We can interpret this requirement as the need to create a subclass, say, `standard-method-combination` to parallel the situation for `method` vs `standard-method` and `generic-function` vs `standard-generic-function`, i.e., so as to allow the programmer to create very different objects from those that the `standard-` version can provide.

Clearly, the text of the dictionary entry means that when the macro `defgeneric` is used with the `:method-combination` option given, such a method combination object is what the generic function will contain. We can confirm this view by examining the description of the MOP generic function `generic-function-method-combination` (as described in [2]) which states that the return value is "a method combination metaobject".

However, the macro `define-method-combination` does *not* define a method combination object. The reason is of course that no method-combination options are supplied to this macro. The dictionary entry for this macro also clearly says that the macro is used to define new *types* of method combinations.

The main issue for the person implementing a Common Lisp system, then, is how to interpret the relation between a *method combination type* and a *method combination object.*

It is easy to draw the conclusion that a call to the macro `define-method-combination` creates a new *class*, as suggested by the use of the word *type* in the standard, and that method combination objects of that type are instances of the new class. However, this view creates several problems. In particular, one must then determine whether each use of the same combination of the type and the arguments in the `:method-combination` option to `defgeneric` creates a new instance of the class, or whether existing instances are somehow kept track of and reused. The first possibility would have the unfortunate consequence that two

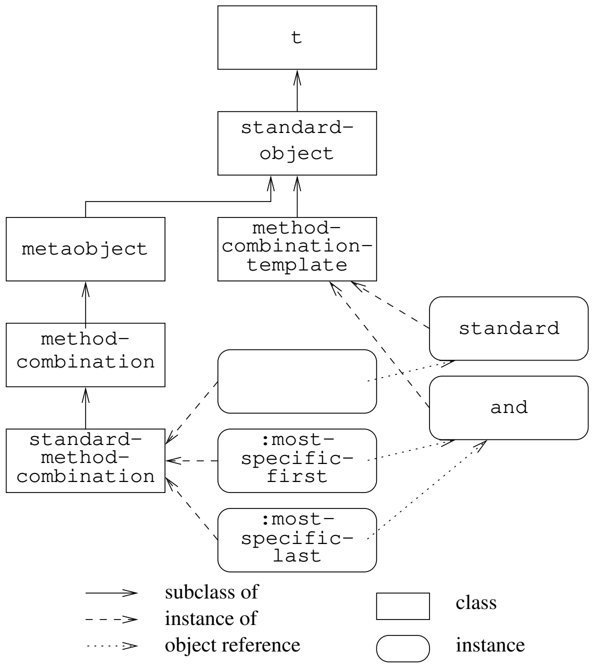

Figure 1: Representation of method combinations.

calls to `generic-function-method-combination` with different generic-function metaobjects would return two method combination objects that are not identical.

In this paper, we argue that a *method combination type* is itself an instance of a completely different class that we shall call `method-combination-template`, and that a *method combination object* is a *variant* of the template in that it contains a reference to the template as well as the values of the *options* that this particular method combination type allows. To conform to the standard, we obviously maintain that method combination objects are instances of `standard-method-combination`. This idea is illustrated in Figure 1, which shows two method combination templates (`standard` and `and`) and two variants of the `and` method-combination template; one variant with the option `:most-specific-first` and another variant with the option `:most-specific-last`.

A call to the macro `define-method-combination` results in a function that can be applied to a list of arguments which include at least a generic function and a list of applicable methods. This function becomes associated with the *name* of the method-combination *type* thus defined. The standard briefly uses the term *procedure* to refer to this resulting function. We adopt that convention in this paper, and refer to the resulting function as the *method-combination procedure*.

When a generic function is defined or redefined, it would be desirable to have the options of the `:method-combination` `defgeneric` option checked for validity immediately when the definition or redefinition occurs. For the built-in method combination types, most implementations also handle this check as a special case. However, all implementations we have investigated fail to check the options to a user-defined method-combination type defined by the long form of the macro `define-method-combination`. Instead, if the options are incompatible with the defined method-combination type, in the best case, an error is signaled when the method-combination procedure is applied to a list of applicable methods by the generic function `compute-effective-method`. Furthermore, the error being signaled can be hard to decipher, as it typically results from invalid arguments to a function with a particular lambda list.

In this paper, we propose a general mechanism for early detection of incompatible options to a particular method-combination type. This mechanism is available to the creator of custom method-combination types, and also used to verify the options to the built-in method-combination types.

The macro `define-method-combination` comes in two versions called the *long form* and the *short form* in the Common Lisp standard. The short form of the macro can be expressed in terms of the long form, but it may not be obvious how the options to the short form should be propagated to the long form.

Furthermore, in the description of the short form of the macro, the standard states that the method-combination procedure resulting from such a definition accepts an optional argument (named `order`) that can have two values, `:most-specific-first` and `:most-specific-last`, with the value `:most-specific-first` being the default. It is not obvious how this restriction can be expressed as a long-form definition of a similar procedure. A common solution to this problem is to define a subclass `short-method-combination` of the class `method-combination`, and to introduce special-purpose code for checking this restriction. The technique presented in this paper does not require such a subclass, as the long-form version of the short-form definition is able to check the restriction.

Throughout this paper, we assume that it is an error to attempt to create a generic function using a method-combination type that is not already created. Recall that the standard states that when a `define-method-combination` form appears at the top level, the compiler must recognize the name of that type as valid in subsequent `defgeneric` forms, but that the resulting method-combination procedure is not executed until the `define-method-combination` form itself is executed. In other words, since the method-combination type is not created at compile time, it may not exist when a `defgeneric` form using the name is encountered by the compiler. However, our assumption is still valid, since the compiler also does not create the generic function when a `defgeneric` form appears at the top level.

For example, assume that some source file contains a `define-method-combination` form, defining a method combination type with a new name, followed by a `defgeneric` form that refers to that method combination type in the `:method-combination` option of the `:defgeneric` form. When the compiler encounters the `define-method-combination` form, it registers its name as being valid for use in subsequent `defgeneric` forms, but the compiler does not *create* the method-combination type. Subsequently, when the compiler encounters the `defgeneric` form, it recognizes a valid name of a method-combination type, but since the compiler also does not create the generic function when the `defgeneric` form is encountered, there is no need for the method-combination type to have been created. When the compiled file is later loaded, the new method-combination type is first created. Subsequently, the generic function is created, referring to an existing method-combination type. In this paper, we do not address the mechanism by which the compile-time behavior required by the standard is implemented.

There are several scenarios that are discussed in this paper:

(1) The user correctly defines a custom method-combination type using `define-method-combination`. Subsequently, the user defines a generic function with that method-combination type, but makes a mistake in the list of options.
(2) The user defines a custom method-combination type using the long form of `define-method-combination`, but makes a mistake in the lambda list supplied to the macro, so that the options of the resulting method-combination procedure are not the ones that were intended. Subsequently, the user defines a generic function with a list of options that were intended to be acceptable.
(3) The user initially correctly defines a custom method combination type using `define-method-combination`, and then also correctly defines one or more generic functions with that method combination type. Then the user decides to make a change to the code of the method-combination type, so the `define-method-combination` form is re-executed, but the new lambda list is incompatible with the options given when the generic functions were created, either as a result of a mistake or of a deliberate decision.

To illustrate these scenarios, we can imagine a restricted form of the `and` method combination that does not admit any `:around` methods. This restriction means that the short form of `define-method-combination` can not be used.

An example of the first scenario would be the following code:

```
(define-method-combination simple-and
    (&optional (order :most-specific-first))
    (primary (and) :order order :required t)
 ...)

(defgeneric simple-and (...)
  (:method-combination simple-and :msot-specific-first))
```

Here, the user has a typo in the second form. An example of the second scenario would be the following code:

```
(define-method-combination simple-and
    (&optional (order :msot-specific-first))
    (primary (and) :order order :required t)
 ...)

(defgeneric simple-and (...)
  (:method-combination simple-and :most-specific-first))
```

Here, the user has a typo in the first form. Finally, an example of the third form would be the following code:

```
(define-method-combination simple-and
    (&optional (order :most-specific-first))
    (primary (and) :order order :required t)
 ...)

(defgeneric simple-and (...)
  (:method-combination simple-and :most-specific-first))
```

Here, there are no mistakes, but the user later decides to disallow the option , so the first form is altered to become:

```
(define-method-combination simple-and ()
    (primary (and) :order :most-specific-first
                   :required t)
 ...)
```

In the first two scenarios, the ideal consequence would be that a warning is initially signaled, stating that the options supplied to the creation of the generic function are incompatible with the type of the desired method combination. Any subsequent attempt to execute the generic function would result in an appropriate error being signaled. Once the incorrect definition has been corrected and the corresponding form has been re-executed, the generic function should be operational.

In the third scenario, the ideal consequence would be that a warning is signaled, giving a list of generic functions with a list of options that are now incompatible with the redefined method-combination type. Any subsequent attempt to execute one of these generic functions would result in an appropriate error being signaled. If a mistake was made, the re-execution of a corrected `define-method-combination` form should render the existing generic functions operational again. If the change was deliberate, the list of generic functions in the message can be used to determine which definitions to correct and re-execute.

The technique described in this paper handles all these scenarios, but it has been implemented only partially. We are currently working on incorporating the remaining elements of our technique into the SICL[1] code base.

2 PREVIOUS WORK

In this section, we give an overview of how different free Common Lisp implementations represent and handle method combinations. In particular, we compare the technique that each implementation uses with the three scenarios specified in Section 1. We do not include commercial Common Lisp implementations, simply because we can not know in detail how the code is written. Extensive experimentation might have given sufficient clues, but we prefer to limit ourselves to implementation where we can examine the source code.

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

2.1 PCL

Portable Common Loops[2], PCL for short, is a library that implements the functions defined in the book "The Art of the Metaobject Protocol" [2], and is meant as an add-on to pre-standard Common Lisp implementations, i.e., implementations without CLOS.

Most Common Lisp implementations that exist today were initially written before the standard was published, and many of those implementations chose to use PCL to incorporate CLOS functionality, though frequently, the code has since been adapted for each specific implementation. Much of the analysis in this section was also described in [4], although the description in that paper refers to the way SBCL handled method combinations at the time that article was written.

PCL unsurprisingly defines the class `method-combination` and then the class `standard-method-combination` as a subclass of the class named `method-combination`.

More surprisingly, it then defines two subclasses of the class `standard-method-combination`, namely `long-method-combination` and `short-method-combination`, each for use with the different forms (long and short) of the macro `define-method-combination`.

The class `standard-method-combination` contains slots for the method-combination type (i.e., a symbol), and the method-combination options.

The class `short-method-combination` adds two more slots: namely, the operator and a Boolean that indicates whether the operator, when given a single argument, is the identity function.

The short form of `define-method-combination` adds a method to the generic function `find-method-combination`. The second parameter of this method has an `eql` specializer with the name of the method-combination type being defined. The method function of this method first checks that the options given are valid for the short form of `define-method-combination`, and then it creates a fresh instance of the class `short-method-combination`. In other words, a fresh method combination is created whenever `find-method-combination` is called, which is typically whenever a generic function is created. As a result, with a method-combination type defined by the short form, the method combination of a generic function using this type is not updated as a result of redefining that method-combination type, which is undesirable.

Furthermore, `compute-effective-method` has a method specialized to the class `short-method-combination` that handles the case of the short method combination as a special case.

The long form of `define-method-combination` turns the body of the form into a method-combination procedure. This procedure has the same lambda list as `compute-effective-method`. The expansion of the macro stores this procedure in a global hash table, using the method-combination type as a key. There is a slot for this procedure in the class `long-method-combination`, but this slot is not used.

[2]https://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/lisp/oop/clos/pcl/0.html

Like the short form, the long form also creates a method on `find-method-combination`, also with an `eql` specializer for the second parameter. This method simply creates an instance of the class `long-method-combination`. The generic function `compute-effective-method` has a method specialized to the class `long-method-combination`. This method consults the hash table to find the method-combination procedure and applies that procedure to the generic function, the method combination, and the applicable methods.

Appendix B of [4] shows some very strange consequences of the use of the global hash table, combined with the fact that the effective-method caches of existing generic functions are not flushed when the method-combination type is redefined by the long form. A generic function may well end up with some effective methods computed *before* the redefinition and some computed *after* it. Needless to say, this behavior is very undesirable.

In summary then, the generic function named `find-method-combination` acts as a container for method-combination types, encoded as `eql`-specialized methods. Furthermore, there is no attempt to reuse existing method combinations. A new one is created whenever `find-method-combination` is called. Finally, while the validity of the options is verified for the built-in method combination types, no such verification is done for custom method-combination types defined by the long form of `define-method-combination`.

2.2 SBCL

The SBCL[3] Common Lisp implementation uses a heavily modified version of PCL (See Section 2.1). Prior to April of 2018, SBCL used the unmodified technique from PCL as described in section 2.1. The technique described in this section is a result of significant modifications to the code for handling method combinations. The article by Didier Verna [4] published at ELS in April of 2018 contained a detailed description of the technique used by SBCL at that time. The improvements to SBCL were likely a result of the descriptions in that article.

One aspect of the SBCL code that remains from the previous version is that the two subclasses of `method-combination` are still present.

An invocation of `define-method-combination` does not create any new class. Instead, an *info* structure is created, and stored in a hash table that uses the name of the method-combination type as a key. This info structure contains a *cache*, which is an association list. The key of an element of the association list is a list of options for the method combination, and the value of an element is the method-combination object. Initially, the cache is empty, except for the info structure associated with the `standard` method combination.

The function `find-method-combination` is given the name of the method combination and the desired options. It looks up the appropriate info structure, and searches the cache for an element corresponding to the options. If such an element is found, the method-combination object is returned. If no

[3]http://www.sbcl.org/

element is found, a new one is constructed, pushed on the cache, and returned. The new element is constructed by consing the list of options and the result of applying a *constructor function* to the list of options. This constructor function is stored in a slot in the info structure. As a result, existing method-combination instances are reused whenever possible.

When a generic function is defined with one of the built-in method combinations, or with a method combination defined using the short form, SBCL will check that the options given to the `:method-combination defgeneric` option are valid. This verification is done by special-purpose code. However, with a user-defined method combination using the long form, no verification is done. It is only when an attempt is made to invoke the generic function that the method-combination procedure is invoked, and the incompatible lambda lists are detected. Furthermore, the error message is very general and can be difficult to decipher by the programmer.

SBCL handles reevaluation of `define-method-combination` forms with the name of an existing info entry in the hash table. Every method-combination instance contains a list of back pointers to generic functions that use this method combination. The cache of the existing info entry is traversed, and for each method combination, the effective methods of its generic functions are invalidated. The problems indicated in Appendix B of [4] therefore no longer exist in recent versions of SBCL. When the method-combination type is redefined with a different form of `define-method-combination`, SBCL correctly changes the class of the method-combinations of the type in question, but it fails to verify that the existing options are compatible with the new definition, even when the redefinition is using the short form of `define-method-combination`. The reason for this failure is that the options are verified only as a result of a call to `find-method-combination`, and this function is not called when a method-combination type is redefined.

2.3 Clozure Common Lisp

The Clozure Common Lisp[4] implementation (CCL for short) defines the class `method-combination` and then three subclasses of that class:

- `standard-method-combination` with a single instance, namely the standard method combination. This class is used as a specializer in a method on the generic function `compute-effective-method` so as to handle the standard method combination as a special case.
- `short-method-combination` which is used for method combinations defined by the short form of the macro `define-method-combination`.
- `long-method-combination` which is used for method combinations defined by the long form of the macro `define-method-combination`.

The class `standard-method-combination` in CCL thus does not play the role of a general instantiable subclass of `method-combination`.

[4]https://ccl.clozure.com/

The generic function `compute-effective-method` has a method specialized to each of these subclasses. The method specialized to `standard-method-combination` uses special-purpose code in order to achieve the effect of the standard method combination. The standard method combination is thus not defined using `define-method-combination`. Similarly, the method specialized to `short-method-combination` uses special purpose code. Only the method specialized to `long-method-combination` invokes the method-combination procedure to achieve the desired effect.

In CCL, the macro `define-method-combination` does not define a method combination class. Instead it defines an *info* vector (disguised as a structure) that acts as a template for creating method combinations later. The info vector contains the following elements:

- The name of the method-combination class to be created which is either `short-method-combination` or `long-method-combination`.
- An element that contains the short-form options if the info vector was created as a result of the short form of `define-method-combination`, and the method-combination procedure (called the *expander function* in CCL) if the info vector was created as a result of the long form.
- A list of *instances*, i.e., method-combination objects that share the same info vector.
- A list of generic functions using method combinations of the type defined by the info vector.

This information is used in order to invalidate effective-method caches when a method-combination type is redefined. Therefore, CCL does not have the problem that PCL does, described in Section 2.1.

When a long method-combination type is redefined using the short form of `define-method-combination`, every generic function having a method combination of that type is accessed, and the method-combination options are checked so that they are valid for the short method combination, i.e., either there are no explicit options or the options consist of a singleton list containing either `:most-specific-first` or `:most-specific-last`. No analogous verification is made when a short method-combination type is redefined using the long form. However, in both cases, the method combination with the redefined type is passed to `change-class`, thereby making the redefinition effective in all generic functions with a method combination of that type.

2.4 ECL

The ECL[5] Common Lisp implementation defines the class `method-combination`, and method-combination metaobjects are direct instances of this class. Thus, in this respect, ECL is not conforming.

Unlike PCL, SBCL, and CCL, ECL does not define any subclasses of the instantiable class. Method-combination types defined by the short form are rewritten to the equivalent long form.

[5]https://common-lisp.net/project/ecl/

The macro `define-method-combination` does not define a new method-combination class. Instead it defines a method-combination procedure. This procedure computes the effective method of a generic function. The lambda list of the method-combination procedure consists of two required parameters: namely, a generic function and a list of applicable methods, followed by the lambda list given to `define-method-combination`. For most built-in method-combination types, that lambda list will contain an optional parameter named `order` with a default value of `:most-specific-first`. The resulting method-combination procedure is stored in a hash table with the name of the method-combination type as a key.

When a generic function is created, a new instance of the `method-combination` class is created. The new instance contains the method-combination procedure and a list of the options given after the method-combination name in the `:method-combination` option to `defgeneric`.

Redefining a method-combination type does not have any effect on existing generic functions having a method combination of that type. The hash table containing method-combination procedures is updated, but this update does not affect existing generic functions.

The `standard` method combination is not defined using the macro `define-method-combination`. Instead, it is defined using special-purpose code.

Incompatibilities between method-combination options given to `find-method-combination` and the lambda list of the method-combination procedure are detected when an effective method needs to be computed. Because there is no specific class for method combinations defined by the short form, this behavior is true also for method-combination types defined by the short form.

Because the short form is rewritten into the long form, and the body of the resulting form contains no verification that the option is either `:most-specific-first` or `:most-specific-last`, it is possible to give any object as an option to `find-method-combination`. Any object different from the keyword `:most-specific-last` will make the resulting method combination behave as if `:most-specific-first` had been given. We argue that this behavior is not conforming, since the description of the short form of `define-method-combination` states that this form "automatically includes error checking".

2.5 Clasp

Clasp [3] is a Common Lisp implementation based on ECL (See Section 2.4), although all the C code in ECL was rewritten in C++.

A large part of the Common Lisp code in Clasp is identical or near-identical to the corresponding code in ECL, and that includes the code for handling method combinations. As a result, Clasp handles method combinations in exactly the same way as ECL.

3 OUR TECHNIQUE

3.1 Representation of method combinations

We introduce a class named `method-combination-template`. An instance of this class represents all method combinations with the same *name*, independent of the options. There is a template for `standard`, a template for `and`, etc. Furthermore, in order to respect the restriction required by the standard, we introduce a class `standard-method-combination` which is a subclass of `method-combination`. All method-combination metaobjects are direct instances of this subclass. There are no subclasses of `standard-method-combination`, neither for specific method-combination types, nor for distinguishing between method combinations defined by the long and the short form of `define-method-combination`. In other words, a method combination is a *variant* of a method-combination template. The template contains a list of all its variants in use.

A method-combination instance contains the following slots:

- A reference to its template.
- The list of method-combination *options* to be given to `find-method-combination`, and that typically appear after the method-combination name of the `:method-combination defgeneric` option.
- The method-combination procedure. This procedure has two parameters, both required. The first parameter is a generic function for which an effective method is to be computed. The second parameter is a list of pairs. Each pair contains an applicable method, and a list of method *qualifiers* for that method. The result of applying the method-combination procedure is a form called the *effective method*. Notice that the method-combination procedure does *not* have the method-combination options in its lambda list.
- A list of generic functions that contain this method combination.

3.2 When `find-method-combination` is called

The expansion of the `defgeneric` macro contains a call to the ordinary function `ensure-generic-function`. If the `:method-combination` option is explicitly supplied to the call to `defgeneric`, then the call to `ensure-generic-function` contains an explicit keyword argument `:method-combination` with the value form being a call to the generic function `find-method-combination` with the generic function, the name of the method-combination type, and the options. If no `:method-combination` option is given in the `defgeneric` form, the `:method-combination` keyword argument to the call to `ensure-generic-function` is not supplied.

The call to `find-method-combination` either returns an existing method-combination instance corresponding to the type and the options given, or it creates and stores a new such instance. If the options are incompatible with the method-combination template, a warning is signaled, and the method-combination procedure is one that signals an error if invoked. The mechanism for detecting this incompatibility is described later in this section.

A call to `ensure-generic-function` results in a call to `ensure-generic-function-using-class` where the first argument is either an existing generic function or `nil` if no generic function with the given name exists. The method on `ensure-generic-function-using-class` specialized to the class `null` supplies the `standard` method-combination as a default value of the `:method-combination` when calling `make-instance` to create a new generic function.

To detect whether a list of method-combination options are invalid for a particular method-combination template, we analyze the *lambda-list* given in the long form of `define-method-combination`. The analysis consists of extracting all parameters that can be referenced in the method-combination procedure. We then construct a lambda expression as follows:

```
(lambda (...)
  (list v1 v2 ... vn))
```

which is then compiled so that a function is obtained. The lambda list of this function is the lambda list that appears in the `define-method-combination` form and `v1`, `v2`, ..., `vn` are the lexical variables resulting from our analysis of the lambda list. Applying this function to the options given to the `find-method-combination` function returns a list of objects. The lambda list typically contains `&aux` lambda list keywords, with forms that check the validity of the options supplied, and signal an error whenever an invalid option combination is detected. Thus, if either the lambda list is incompatible with the options given, or one of these `&aux` forms detects an invalid option combination, an error is signaled. We handle this error, turn it into a warning, and return a method-combination instance with a method-combination procedure that signals an error whenever invoked.

This technique for detecting incompatible or invalid options handles the first scenario described in Section 1. When the user corrects the incorrect form that created or reinitialized the generic function (typically a `defgeneric` form), the validation process is re-invoked and a method-combination with a viable method-combination procedure is assigned to the generic function. This technique also detects the second scenario described in Section 1. The way the user can correct the situation in this scenario is described below.

When the options given to `find-method-combination` are compatible and valid, a viable method-combination procedure is constructed as follows:

```
(lambda (generic-function method-qualifier-pairs)
  (let ((v1 ...) (v2 ...) ... (vn ...))
    <body>))
```

where `v1`, `v2`, ..., `vn` are again the lexical variables resulting from our analysis of the lambda list. The initialization forms for the variables are the values returned in the resulting list of our analysis function.

3.3 Redefining a method-combination type

When a `define-method-combination` form is re-evaluated, we locate the corresponding method-combination template. We then invoke the same analysis as before to every variant, i.e., to every existing method combination having this type name. If an analysis fails, we then signal a warning containing all generic functions using the now invalid method-combination, and we set the method-combination procedure of the invalid method combination to one that will signal an error when invoked. If the analysis succeeds, then the corresponding method combination is assigned a viable method-combination procedure.

3.4 Expanding the short form to the long form

As mentioned in Section 1, it is not obvious how to transform the short form of `define-method-combination` into the long form. Recall that the syntax of the short form is:

`(define-method-combination` *name [[short-form-options]]*`)`
where a *short-form-option* can be:

- `:documentation` *documentation*
- `:identity-with-one-argument` *identity-with-one-argument*
- `:operator` *operator*

Here, *documentation* is a string that is not evaluated. When the short form gets turned into the long form, it becomes an ordinary documentation string, preceding the forms of the body of the long form.

To illustrate where the remaining options end up in the long form, recall the following example from the dictionary entry for `define-method-combination`, where both the short form and the long form are used to define the built-in method-combination `and`. We have changed only the layout of the code so that it will fit on the page.
The short form is:

```
(define-method-combination and
  :identity-with-one-argument t)
```

The long form is;

```
(define-method-combination and
        (&optional (order :most-specific-first))
        ((around (:around))
         (primary (and) :order order :required t))
  (let ((form (if (rest primary)
                  `(and ,@(mapcar
                           #'(lambda (method)
                               `(call-method ,method))
                           primary))
                  `(call-method ,(first primary)))))
    (if around
        `(call-method ,(first around)
                      (,@(rest around)
                       (make-method ,form)))
        form)))
```

The option *identity-with-one-argument* is responsible for the form:

```
(if (rest primary)
    `(and ,@(mapcar
              #'(lambda (method)
                  `(call-method ,method))
              primary))
    `(call-method ,(first primary)))))
```

Had this option been `nil` or not present, the corresponding form would have looked like this instead:

```
`(and ,@(mapcar
          #'(lambda (method)
              `(call-method ,method))
          primary))
```

In order for our technique to work for the short form, when we express the short form in terms of the long form, we modify the lambda list of the long form compared to the example above as follows:

```
(&optional (order :most-specific-first)
 &aux (ignore (unless (member order
                              '(:most-specific-first
                                :most-specific-last))
                (error ....))))
```

Now, any attempt to call a function with this lambda list with a number of arguments other than exactly 1, or with one argument that is neither `:most-specific-first` nor `:most-specific-last` will fail.

4 CONCLUSIONS AND FUTURE WORK

We define a subclass `standard-method-combination` of the specified class `method-combination`. Method combinations created from a method-combination type defined by the macro `define-method-combination` are all instances of this subclass.

Our technique allows for early detection of mismatches between the method-combination options given when a method combination is created as a result of calling `find-method-combination` and the lambda list given to the invocation of `define-method-combination`. We detect such mismatches when a new method combination is created, but also when a method-combination type is redefined with a modified invocation of `define-method-combination` using the name of an existing method-combination type.

Furthermore, while a mismatch exists, our technique results in an error being signaled whenever an attempt is made to use the faulty method combination in order to create an effective method.

Future work includes incorporating our technique into the SICL code base. The technique described in this paper was developed after our initial implementation of method combinations in SICL. (Hence, this technique was not in SICL from the start.) Currently, SICL does not have any data structure allowing weak references, but such references would be desirable for the back pointer from a method combination to the generic functions using it. Otherwise, a memory leak would result from using `fmakunbound` or some other operator that makes the back pointer be the only reference to the generic function. In general, it is impossible to have such operators remove the back pointer, since there could be any number of references to the generic function in question.

5 ACKNOWLEDGMENTS

We would like to thank Yan Gajdoš and Cyrus Harmon for providing valuable feedback on early versions of this paper.

REFERENCES

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

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

[3] CE Schafmeister. Clasp–A Common Lisp that Interoperates with C++ and Uses the LLVM Backend. In *Proceedings of the 8th European Lisp Symposium*, pages 90–91, New York, NY, USA, 2015. ACM.

[4] Didier Verna. Method combinators. In *11th European Lisp Symposium*, Marbella, Spain, April 2018. ISBN 9782955747421. doi: 10.5281/zenodo.3247610.