Method Combinators

Didier Verna
EPITA
Research and Development Laboratory
Le Kremlin-Bicêtre, France
didier@lrde.epita.fr

ABSTRACT

In traditional object-oriented languages, the dynamic dispatch algorithm is hardwired: for every polymorphic call, only the most specific method is used. CLOS, the Common Lisp Object System, goes beyond the traditional approach by providing an abstraction known as *method combinations*: when several methods are applicable, it is possible to select several of them, decide in which order they will be called, and how to combine their results, essentially making the dynamic dispatch algorithm user-programmable.

Although a powerful abstraction, method combinations are under-specified in the Common Lisp standard, and the MOP, the Meta-Object Protocol underlying many implementations of CLOS, worsens the situation by either contradicting it or providing unclear protocols. As a consequence, too much freedom is granted to conforming implementations. The exact or intended behavior of method combinations is unclear and not necessarily coherent with the rest of CLOS.

In this paper, we provide a detailed analysis of the problems posed by method combinations, the consequences of their lack of proper specification in one particular implementation, and a MOP-based extension called *method combinators*, aiming at correcting these problems and possibly offer new functionality.

CCS CONCEPTS

• **Software and its engineering → Object oriented languages**; **Extensible languages**; **Polymorphism**; *Inheritance*; *Classes and objects*; *Object oriented architectures*; *Abstraction, modeling and modularity*.

KEYWORDS

Object-Oriented Programming, Common Lisp Object System, Meta-Object Protocol, Generic Functions, Dynamic Dispatch, Polymorphism, Multi-Methods, Method Combinations, Orthogonality

ACM Reference Format:
Didier Verna. 2019. Method Combinators. In *Proceedings of the 11th European Lisp Symposium (ELS'18).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.5281/zenodo.3247610

ELS'18, April 16–17 2018, Marbella, Spain

ACM ISBN 978-2-9557474-2-1.
https://doi.org/10.5281/zenodo.3247610

1 INTRODUCTION

Common Lisp was the first programming language equipped with an object-oriented (OO) layer to be standardized [16]. Although in the lineage of traditional class-based OO languages such as Smalltalk and later C++ and Java, CLOS, the Common Lisp Object System [2, 5, 7, 9], departs from those in several important ways.

First of all, CLOS offers native support for multiple dispatch [3, 4]. Multiple dispatch is a generalization of single dispatch *a.k.a.* inclusion polymorphism [12]. In the classic message-passing style of single dispatch, the appropriate method is selected according to the type of the receiver (the object through which the method is called). In multiple dispatch however, the method selection algorithm may use as many arguments as requested in the generic call. Because this kind of polymorphism doesn't grant any object argument a particular status (message receiver), methods (herein called *multi-methods*) are naturally decoupled from classes and generic function calls look like ordinary function calls. The existence of multi-methods thus pushes dynamic dispatch one step further in the direction of separation of concerns: polymorphism and inheritance are clearly separated.

Next, CLOS itself is written on top of a meta-object protocol, the CLOS MOP [10, 13]. Although not part of the ANSI specification, the CLOS MOP is a *de facto* standard well supported by many implementations. In supporting implementations, the MOP layer not only allows for CLOS to be implemented in itself (classes being instances of their meta-classes *etc.*), but also lets the programmer extend or modify its very semantics, hence providing a form of homogeneous behavioral reflection [11, 14, 15].

Yet another improvement over the classical OO approach lies in the concept of *method combination*. In the traditional approach, the dynamic dispatch algorithm is hardwired: every polymorphic call ends up executing the most specific method available (applicable) and using other, less specific ones requires explicit calls to them. In CLOS however, a generic function can be programmed to implicitly call several applicable methods (possibly all of them), not necessarily by order of specificity, and combine their results (not necessarily all of them) in a particular way. Along with multiple dispatch, method combinations constitute one more step towards *orthogonality* [8, chapter 8]: a generic function can now be seen as a 2D concept: 1. a set of methods and 2. a specific way of combining them. As usual with this language, method combinations are also fully programmable, essentially turning the dynamic dispatch algorithm into a user-level facility.

Richard P. Gabriel reports[1] that at the time Common Lisp was standardized, the standardization committee didn't believe that

[1] in a private conversation

```
(defgeneric details (human)
  (:method-combination append :most-specific-last))
(defmethod details append ((human human)) ...)
(defmethod details append ((employee employee)) ...))
```

Figure 1: Short Method Combination Usage Example

method combinations were mature enough to make people implement them in one particular way (the only industrial-strength implementation available back then was in Flavors on Lisp Machines). Consequently, they intentionally under-specified them in order to leave room for experimentation. At the time, the Mop was not ready either, and only added later, sometimes with unclear or contradictory protocols. The purpose of this paper is to provide a detailed analysis of the current status of method combinations, and also to offer possible improvements over them.

Section 2 provides a detailed analysis of the specification for method combinations (both Clos and the Mop) and points out its caveats. Section 3 describes how Sbcl[2] implements method combinations, and exhibits some of their inconsistent or unfortunate (although conformant) behavior. Sections 4 and 5 provide an extension to method combinations, called *method combinators*, aimed at fixing the problems previously described. Finally, Section 6 demonstrates an additional feature made possible with method combinators, which increases yet again the orthogonality of generic functions in Common Lisp.

2 METHOD COMBINATIONS ISSUES

In this section, we provide an analysis of how method combinations are specified and point out a set of important caveats. This analysis is not only based on what the Common Lisp standard claims, but also on the additional requirements imposed by the Clos Mop. In the remainder of this paper, some basic knowledge on method combinations is expected, notably on how to define them in both short and long forms. The reader unfamiliar with `define-method-combination` is invited to look at the examples provided in the Common Lisp standard first[3].

2.1 Lack of Orthogonality

As already mentioned, method combinations help increase the separation of concerns in Common Lisp's view on generic functions. The orthogonality of the concept goes only so far however, and even seems to be hindered by the standard itself occasionally. This is particularly true in the case of method combinations defined in short form (or built-in ones, which obey the same semantics).

Figure 1 demonstrates the use of the `append` built-in combination, concatenating the results of all applicable methods. In this particular example, and given that employees are humans, calling `details` on an employee would collect the results of both methods. Short combinations require methods to have exactly *one* qualifier: either the combination's name for primary methods (`append` in our example), or the `:around` tag[4]. This means that one cannot change a generic function's (short) method combination in a practical way, as it would basically render every primary method unusable (the standard also mandates that an error be signaled if methods without a qualifier, or a different one are found). Hence, method combinations are not completely orthogonal to generic functions. On the other hand, `:around` methods remain valid after a combination change, a behavior inconsistent with that of primary methods.

Perhaps the original intent was to improve readability or safety: when adding a new method to a generic function using a short method combination, it may be nice to be reminded of the combination's name, or make sure that the programmer remembers that it's a non-standard one. If such is the case, it also fails to do so in a consistent way. Indeed, short method combinations support an option affecting the order in which the methods are called, and passed to the `:method-combination` option of a `defgeneric` call (`:most-specific-first/last`, also illustrated in Figure 1). Thus, if one is bound to restate the combination's name anyway, why not restate the potential option as well? Finally, one may also wonder why short method combinations didn't get support for `:before` and `:after` methods as well as `:around` ones.

Because short method combinations were added to enshrine common, simple cases in a shorter definition form, orthogonality was not really a concern. Fortunately, short method combinations can easily be implemented as long ones, without the limitations exhibited in this section (see Appendix A).

2.2 Lack of Structure

The Common Lisp standard provides a number of concepts related to object-orientation, such as objects, classes, generic functions, and methods. Such concepts are usually gracefully integrated into the type system through a set of classes called *system classes*. Generic functions, classes, and methods are equipped with two classes: a class named *C* serving as the root for the whole concept hierarchy, and a class named `standard-`*C* serving as the default class for objects created programmatically. In every case, the standard explicitly names the APIs leading to the creation of objects of such standard classes. For example, `standard-method` is a subclass of `method` and is "the default class of methods defined by the `defmethod` and `defgeneric` forms"[5].

Method combinations, on the other hand, only get one standardized class, the `method-combination` class. The Mop further states that this class should be abstract (not meant to be instantiated), and also explicitly states that it "does not specify the structure of method combination metaobjects"[10, p. 140]. Yet, because the standard also requires that method combination objects be "indirect instances" of the `method-combination` class[6], it is mandatory that subclasses are provided by conforming implementations (although no provisions are made for a `standard-method-combination` class for instance). Although this design may seem inconsistent with the rest of Clos, the idea, again, was to leave room for experimentation. For example, knowing that method combinations come in two forms, *short* and *long*, and that short combinations may be implemented as long ones, implementations can choose whether to represent short and long combinations in a single or as separate hierarchies. The unfortunate consequence, however, is that it is

[2] http://www.sbcl.org
[3] http://www.lispworks.com/documentation/lw70/CLHS/Body/m_defi_4.htm
[4] http://www.lispworks.com/documentation/lw70/CLHS/Body/07_ffd.htm
[5] http://www.lispworks.com/documentation/lw70/CLHS/Body/t_std_me.htm
[6] http://www.lispworks.com/documentation/lw70/CLHS/Body/t_meth_1.htm

impossible to specialize method combinations in a portable way, because implementation-dependent knowledge of the exact method combination classes is needed in order to subclass them.

Yet another unfortunate consequence of this under-specification lies in whether method combinations should be objects or classes to be instantiated, although the original intent was to consider them as some kind of macros involved in method definition. The Common Lisp standard consistently talks of "method combination types", and in particular, this is what is supposed to be created by `define-method-combination`[7]. This seems to suggest the creation of classes. On the other hand, method combinations can be parametrized when they are used. The long form allows a full ordinary lambda-list to be used when generic functions are created. The short form supports one option called `:identity-with-one-argument`, influencing the combination's behavior at creation-time (originally out of a concern for efficiency), and another one, the optional `order` argument, to be used by generic functions themselves. The long form also has several creation-time options for method groups such as `:order` and `:required`, but it turns out that these options can also be set at use-time, through the lambda-list.

2.3 Unclear Protocols

The third and final issue we see with method combinations is that the Mop, instead of clarifying the situation, worsens it by providing unclear or inconsistent protocols.

2.3.1 `find-method-combination`. In Common Lisp, most global objects can be retrieved by name one way or another. For example, `symbol-function` and `symbol-value` give you access to the Lisp-2 namespaces [6], and other operators perform a similar task for other categories of objects (`compiler-macro-function` being an example). The Common Lisp standard defines a number of `find-*` operators for retrieving objects. Amongst those are `find-method` and `find-class` which belong to the Clos part of the standard, but there is no equivalent for method combinations.

The Mop, on the other hand, provides a generic function called `find-method-combination` [10, p. 191]. However, this protocol only adds to the confusion. First of all, the arguments to this function are a generic function, a method combination type name, and some method combination options. From this prototype, we can deduce that contrary to `find-class` for example, it is not meant to retrieve a globally defined method combination by name. Indeed, the description of this function says that it is "called to determine the method combination object used by a generic function". Exactly *who* calls it and *when* is unspecified however, and if the purpose is to retrieve the method combination used by a generic function, then one can wonder what the second and third arguments (method combination type and options) are for, and what happens if the requested type is *not* the type actually used by the generic function. In fact, the Mop already provides a more straightforward way of inquiring a generic function about its method combination. `generic-function-method-combination` is an accessor doing just that.

2.3.2 `compute-effective-method`. Another oddity of method combinations lies in the design of the generic function invocation protocol. This protocol is more or less a two steps process. The first step consists in determining the set of applicable methods for a particular call, based on the arguments (or their classes). The Common Lisp standard specifies a function (which the Mop later refines), `compute-applicable-methods`, which unsurprisingly accepts two arguments: a generic function and its arguments for this specific call. The second step consists in computing (and then calling) the *effective method*, that is, the combination of applicable methods, precisely combined in a manner specified by the generic function's method combination. While the Common Lisp standard doesn't specify how this is done, the Mop does, via a function called `compute-effective-method`. Unsurprisingly again, this function accepts two arguments: a generic function and a set of (applicable) methods that should be combined together. More surprisingly however, it takes a method combination as a third (middle) argument. One can't help but wonder why such an argument exists, as the generic function's method combination can be retrieved through its accessor which, as we saw earlier, is standardized. Here again, we may be facing a aborted attempt at more orthogonality. Indeed, this protocol makes it possible to compute an effective method for *any* method combination, not just the one currently in use by the generic function (note also that the Mop explicitly mentions that `compute-effective-method` may be called by the user [10, p. 176]). However, the rest of Clos or the Mop doesn't support using `compute-effective-method` in this extended way. It is, however, an incentive for more functionality (see Section 6).

2.3.3 Memoization. One final remark in the area of protocols is about the care they take for performance. The Mop describes precisely how and when a discriminating function is allowed to cache lists of applicable methods [10, p. 175]. Note that nothing is said about the location of such a cache however (within the discriminating function, in a lexical closure over it, globally for every generic function *etc.*), but it doesn't really matter. On the other hand, the Mop says nothing about caching of effective methods. This means that conforming implementations are free to do what they want (provided that the semantics of Clos is preserved). In particular, if caching of effective methods is done, whether such a cache is maintained once for every generic function, or once for every generic function/method combination pair is unspecified. This is rather unfortunate, both for separation of concerns, and also for the extension that we propose in Section 6.

3 THE CASE OF SBCL

In this section, we analyse Sbcl's implementation of Clos, and specifically the consequences of the issues described in the previous section. Note that with one exception, the analysis below also stands for Cmucl[8] from which Sbcl is derived, and which in turn derives its implementation of Clos from Pcl [1].

3.1 Classes

The Sbcl method combination classes hierarchy is depicted in Figure 2. It provides the `standard-method-combination` class that was missing from the standard (see Section 2.2), although this class

[7] http://www.lispworks.com/documentation/lw70/CLHS/Body/m_defi_4.htm

[8] https://www.cons.org/cmucl/

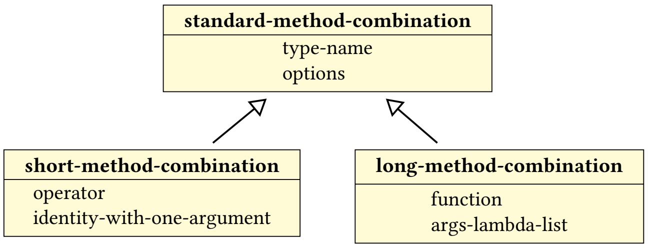

Figure 2: SBCL Method Combination Classes Hierarchy

doesn't serve as the default implementation for method combinations, as two subclasses are provided for that, one for each combination form. The `options` slot in the base class stores the use-time options (the ones passed to the `:method-combination` option to a `defgeneric` call). New method combination definitions are not represented by new classes; only by instances of either the `short` or `long-method-combination` ones. As a result, method combinations retrieved later on are objects containing a mix of definition-time and use-time options.

3.2 Short Method Combinations

Investigating how short method combinations are created in SBCL uncovers a very peculiar behavior. `define-method-combination` expands to a call to `load-short-defcombin`, which in turn creates a method on `find-method-combination`, `eql`-specialized on the combination's name and ignoring its first argument (the generic function). This method is encapsulated in a closure containing the method combination's parameters, and recreates and returns a *new* method combination object on the fly *every time it is called.*

This has at least three important consequences.

(1) Short method combinations never actually globally exist *per se* (they don't have a namespace proper). Indeed, what is defined is not a method combination object (not even a class), but a means to create one on-demand. In particular, every generic function gets its own brand new object representing its method combination.
(2) `find-method-combination` neither does what its name suggests, nor what the MOP seems to imply. Indeed, because the generic function argument is ignored, it doesn't "determine the method combination object used by a generic function", but just creates whatever method combination instance you wish, of whichever known type and use-time option you like.
(3) It also turns out that redefining a short method combination (for example by calling `define-method-combination` again) doesn't affect the existing generic functions using it (each have a local object representing it). This is in contradiction with how every other CLOS component behaves (class changes are propagated to live instances, method redefinitions update their respective generic functions *etc.*).

3.3 Long Combinations

The case of long method combinations is very similar, although with one additional oddity. Originally in PCL (and still the case in CMUCL), long method combinations are compiled into so-called *combination functions*, which are in turn called in order to compute effective methods. In both PCL and CMUCL, these functions are stored in the `function` slot of the long method combination objects (see Figure 2). In SBCL however, this slot is not used anymore. Instead, SBCL stores those functions in a *global* hash table named `*long-method-combination-functions*` (the hash keys being the combination names). The result is that long method combinations are represented half-locally (local objects in generic functions), half-globally with this hash table.

Now suppose that one particular long method combination is redefined while some generic functions are using it. As for the short ones, this redefinition will not (immediately) affect the generic functions in question, because each one has its own local object representing it. However, the combination function in the global hash table *will* be updated. As a result, if any concerned generic function ever needs to recompute its effective method(s) (for instance, if some methods are added or removed, if the set of applicable methods changes from one call to another, or simply if the generic function needs to be reinitialized), then the updated hash table entry will be used and the generic function's behavior will indeed change according to the updated method combination. With effective methods caching (as is the case in SBCL) and a little (bad) luck, one may even end up with a generic function using different method combinations for different calls at the same time (see Appendix B).

4 METHOD COMBINATORS

In this section, we propose an extension to method combinations called *method combinators*, aiming at correcting the problems described in Sections 2 and 3. Most importantly, method combinators have a global namespace and generic functions using them are sensitive to their modification. Method combinators come with a set of new protocols inspired from what already exists in CLOS, thus making them more consistent with it. As an *extension* to method combinations, they are designed to work on top of them, in a non-intrusive way (regular method combinations continue to work as before). Finally, their implementation tries to be as portable as possible (although, as we have already seen, some vendor-specific bits are unavoidable).

4.1 Method Combinator Classes

Figure 3 depicts the implementation of method combinators in SBCL. We provide two classes, `short/long-method-combinator`, themselves subclasses of their corresponding, implementation-dependent, method combination classes. A `method-combinator-mixin` is also added as a superclass for both, maintaining additional information (the `clients` slot will be explained in Section 5.3) and serving as a means to specialize on both kinds of method combinators at the same time.

4.2 Method Combinators Namespace

Method combinators are stored globally in a hash table and accessed by name. This hash table is manipulated through an accessor called `find-method-combinator` (and its accompanying `setf` function). This accessor can be seen as the equivalent of `find-class` for

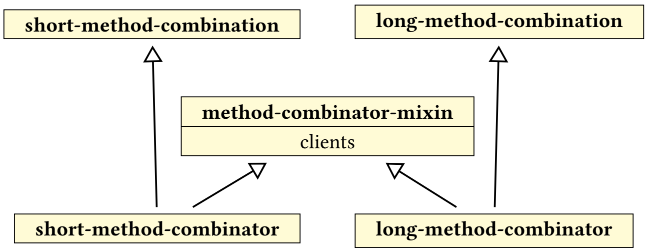

Figure 3: Method Combinator Classes Hierarchy

method combinators, and has the same prototype. It thus considerably departs from the original MOP protocol, but is much more consistent with CLOS itself.

4.3 Method Combinators Management

4.3.1 Protocols. The short method combinator protocol is designed in the same layered fashion as the rest of the MOP. First, we provide a macro called `define-short-method-combinator` behaving as the short form of `define-method-combination`, and mostly taking care of quoting. This macro expands to a call to `ensure-short-method-combinator`. In turn, this (regular) function calls the `ensure-short-method-combinator-using-class` generic function. Unsurprisingly, this generic function takes a method combinator as its first argument, either `null` when a new combinator is created, or an existing one in case of a redefinition. Note that the MOP is not always clear or consistent with its `ensure-*` family of functions, and their relation to the macro layer. In method combinators, we adopt a simple policy: while the functional layer may default some optional or keyword arguments, the macro layer only passes down those arguments which have been explicitly given in the macro call.

The same protocol is put in place for long method combinators. Note that it is currently undecided whether we want to keep distinct interfaces and protocols for short and long forms. The current choice of separation simply comes from the fact that PCL implements them separately. Another yet undecided feature is how to handle definition-time *vs.* use-time options. Currently, in order to simplify the matter as a proof of concept, the (normally) use-time option `:most-specific-first/last` is handled when a short combinator is defined rather than when it is used, and the lambda-list for long forms is deactivated. In other words, use-time options are not supported. Note that this is of little consequence in practice: instead of using the same combination with different use-time arguments, one would just need to define different (yet similar) combinations with those arguments hard-wired in the code.

4.3.2 Creation. A new method combinator is created in 3 steps.

(1) `define-method-combination` is bypassed. Because regular method combinations do not have any other protocol specified, we use SBCL's internal functions directly. Recall that the effect of these functions is to add a new method to `find-method-combination`.
(2) We subsequently call this new method in order to retrieve an actual combination object, and upgrade it to a combinator by calling `change-class`.

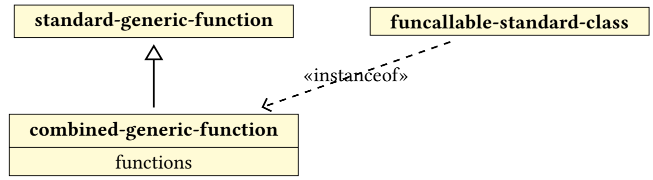

Figure 4: Combined Generic Functions

(3) Finally, this upgraded object is stored in the global combinators hash table by calling `(setf find-method-combinator)`.

All of this is done in layer 3 of the protocols, except that in the case of long combinators, the combination function is computed at the macro level (this is how SBCL does it). Additionally, as CMUCL still does, but contrary to SBCL, we update the `function` slot in the long combinator objects.

The advantage of this process is that defining a combinator also inserts a regular method combination in the system. Regular generic functions may thus use the new combination without any of the combinator extensions.

4.3.3 Modification. An existing method combinator may be updated by the user via the first two protocol layers (the `define-*` macro layer or the `ensure-*` functional one). The updating process is quite simple: it merely involves a call to `reinitialize-instance` or to `change-class` if we are switching combinator forms. The definition change is also propagated to the regular combination layer, and in the case of the long form, care is taken to update not only the `function` slot of the combinator object, but SBCL's `*long-method-combination-functions*` hash table as well.

4.3.4 Built-in Combinators. Finally, we provide new versions of the `standard` and built-in method combinations as combinators. These combinators are named with keywords, so as to both co-exist gracefully with the original Common Lisp ones, and still be easily accessible by name. On top of that, the built-in method combinators are defined in long forms, so as to provide support for `:before` and `:after` methods, and also avoid requiring the combinator's name as a qualifier to primary methods. In fact, a user-level macro called `define-long-short-method-combinator` is provided for defining such "pseudo-short" combinators easily.

5 COMBINED GENERIC FUNCTIONS

At that point, generic functions can seamlessly use method combinators as regular combinations, although with not much benefit (apart from the extended versions of the built-in ones). Our next goal is to ensure that the global method combinator namespace is functioning properly.

5.1 Generic Functions Subclassing

As usual, in order to remain unobtrusive with standard CLOS, we specialize the behavior of generic functions with a subclass handling method combinators in the desired way. This class, called `combined-generic-function`, is depicted in Figure 4 (an explanation for the `functions` slot will be provided in Section 6.2.2). For convenience, a macro called `defcombined` is provided as a wrapper around `defgeneric`. This macro takes care of setting the generic function class to `combined-generic-function` (unless otherwise specified). Also for convenience, a new `:method-combinator` option is provided to replace the regular `:method-combination` one, but ultimately transformed back into

Finally, the (not portable) `method-combination` slot of generic functions is extended to recognize a `:method-combinator` initarg, and a `method-combinator` accessor.

5.2 Method Combinator Management

5.2.1 Initialization. In the absence of an explicit method combinator option, new combined generic functions should use the `:standard` one. This is easily done by providing a default initarg for `:method-combinator` to the `combined-generic-function` class, with a value of `(find-method-combinator :standard)`.

The case of a provided method combinator name is more interesting. Normally, we would wrap `ensure-generic-function/-using-class` with specialized versions to look up a combinator instead of a combination. However, at the expense of portability (a necessity anyway), we can do a little simpler. As it turns out, Sbcl initializes a generic function's method combination by calling `find-method-combination` on the generic function's class prototype. Consequently, we can simply specialize this function with an `eql` specializer on the `combined-generic-function` class prototype, and look up for the appropriate global method combinator object there. Note that in order to specialize on a class prototype, the class needs to have been finalized already. Because of that, we need to call `finalize-inheritance` explicitly and very early on the class `combined-generic-function`.

5.2.2 Sanitation. This is also a good opportunity for us to sanitize the `find-method-combination` protocol for combined generic functions. A new method specialized on such functions is provided. Contrary to the default behavior, this method ensures that the requested method combinator is indeed the one in use by the function, and then returns it (recall that this is a global object). Otherwise, an error is signaled.

5.2.3 Updating. In order to change a combined generic function's method combinator, we provide a convenience function called `change-method-combinator`. This function accepts a combined generic function (to be modified) and a method combinator *designator* (either a name, or directly an object) which it canonicalizes. In the ideal case, this function should be able to only invalidate the generic function's effective method cache. Unfortunately, this cannot be done in a portable way. Thus, the only thing we can do portably is to call `reinitialize-instance` with the new method combinator.

5.3 Client Maintenance

The last thing we need to do is make sure that method combinator updates are correctly propagated to relevant combined generic functions. A combined generic function using a method combinator is called its *client*. Every method combinator maintains a list of clients, thanks to the the `clients` slot of the mixin (see Figure 3).

5.3.1 Registration. Registering a combined generic function as a method combinator client is implemented via two methods. One, on `initialize-instance`, adds a new combined generic function to its method combinator's `clients` slot. The other one, on `reinitialize-instance`, checks whether an existing combined generic function's combinator has changed, and performs the updating accordingly (recall that reinitializing the instance is the only portable way to change a generic function's method combination).

Note that while the Common Lisp standard allows a generic function's class to change, provided that both classes are "compatible" (a term which remains undefined)[9], the Mop seems to imply that meta-classes are only compatible with themselves (it is forbidden to change a generic function's meta-class [10, p. 187]). This restriction makes the client registration process simpler, as a regular generic function cannot become a combined one, or *vice versa*.

5.3.2 Updating. When a method combinator is redefined, it can either remain in the same form, or switch from short to long and *vice versa*. These two situations can be easily detected by specializing `reinitialize-instance` and `u-i-f-d-c`[10] (we could also use `shared-initialize`). Two such `:after` methods are provided, which trigger updating of all the method combinator's clients.

Client updating is implemented thanks to a new protocol inspired from the instance updating one: we provide a generic function called `make-clients-obsolete`, which starts the updating process. During updating, the generic function `u-c-g-f-f-r-m-c`[11] is called on every client. As mentioned previously, there is no portable way to invalidate an effective methods cache in the Clos Mop, so the only thing we can do safely is to completely reinitialize the generic function.

The problem we have here is that while the method combinator has been redefined, the object identity is preserved. Still, we need to trick the implementation into believing that the generic function's method combinator object has changed. In order to do that, we first set the combined generic function's `method-combination` slot to `nil` manually (and directly; bypassing all official protocols), and then call `reinitialize-instance` with a `:method-combinator` option pointing to the same combinator as before. The implementation then mistakenly thinks that the combinator has changed, and effectively reinitializes the instance, invalidating previously cached effective methods.

6 ALTERNATIVE COMBINATORS

In Section 4.3.4, we provided new versions of the built-in method combinations allowing primary methods to remain unqualified. In Section 5.2.3 we offered a convenience function to change the method combinator of a combined generic function more easily (hence the use for unqualified methods). In the spirit of increasing the separation of concerns yet again, the question of *alternative combinators* follows naturally: what about calling a generic function with a different, temporary method combinator, or even maintaining several combinators at once in the same generic function?

In the current state of things, we can already change the method combinator temporarily, call the generic function, and then switch the combinator back to its original value. Of course, the cost of doing it this way is prohibitive, as the generic function would need

[9] http://www.lispworks.com/documentation/lw70/CLHS/Body/f_ensure.htm
[10] `update-instance-for-different-class`
[11] `update-combined-generic-function-for-redefined-method-combinator`

to be reinitialized as many times as one changes its combinator. There is however, a way to do it more efficiently. While highly experimental, it has been tested and seems to work properly in SBCL.

6.1 Protocols

At the lowest level lies a function called `call-with-combinator`. This function takes a combinator object, a combined generic function object and a `&rest` of arguments. Its purpose is to call the generic function on the provided arguments, only with the temporary combinator instead of the original one. On top of this function, we provide a macro called `call/cb` (pun intended) accepting designators (*e.g.* names) for the combinator and generic function arguments, instead of actual objects. Finally, it is not difficult to extend the Lisp syntax with a reader macro to denote alternative generic calls in a concise way. For demonstration purposes, a `#!` dispatching macro character may be installed and used like this:

```
#!combinator(func arg1 arg2 ...)
```

This syntax is transformed into the following macro call:

```
(call/cb combinator func arg1 arg2 ...)
```

In turn, this is finally expanded into:

```
(call-with-combinator
    (find-method-combinator 'combinator)
  #'func arg1 arg2 ...)
```

6.2 Implementation

Method combinations normally only affect the computation of effective methods. Unfortunately, we have already seen that the CLOS MOP doesn't specify how or when effective methods may be cached. Consequently, the only portable way of changing them is to reinitialize a generic function with a different combination. Although effective methods cannot be portably accessed, the generic function's discriminating function can, at least in a half-portable fashion. This gives us an incentive towards a possible implementation.

6.2.1 Discriminating Functions / Funcallable Instances. A generic function is an instance of a *funcallable* class (see Figure 4), which means that generic function objects may be used where functional values are expected. When a generic function (object) is "called", its discriminating function is actually called. The MOP specifies that discriminating functions are installed by the (regular) function `set-funcallable-instance-function`. This strongly suggests that the discriminating function is stored somewhere in the generic function object. Unfortunately, the MOP doesn't specify a reader for that potential slot, although every implementation will need one (this is why we said earlier that discriminating functions could be accessed in a half-portable way). In SBCL, it is called `funcallable-instance-fun`.

6.2.2 Discriminating Function Caches. The idea underlying our implementation of alternative combinators is thus the following. Every combined generic function maintains a cache of discriminating functions, one per alternative combinator used (this is the `functions` slot seen in Figure 4). When an alternative combinator is used for the first time (via a call to `call-with-combinator`), the generic function is reinitialized with this temporary combinator, called, and the new discriminating function is memoized. The function is then reinitialized back to its original combinator, and the values from the call are returned. It is important to actually execute the call *before* retrieving the new discriminating function, because it may not have been calculated before that.

If the alternative combinator was already used before with this generic function, then the appropriate discriminating function is retrieved from the cache and called directly. Of course, care is also taken to call the generic function directly if the alternative combinator is in fact the generic function's default one.

6.2.3 Client Maintenance. Alternative combinators complicate client maintenance (see Section 5.3), but the complication is not insurmountable. When an alternative combinator is used for the first time, the corresponding generic function is registered as one of its clients. The client updating protocol (see Section 5.3.2) is extended so that if the modified combinator is not the generic function's original one, then the generic function is *not* reinitialized. Instead, only the memoized discriminating function corresponding to this combinator is invalidated.

6.2.4 Disclaimer. Generic functions were never meant to work with multiple combinations in parallel, so there is no guarantee on how or where applicable and effective method caches, if any, are maintained. Our implementation of alternative combinators can only work if each discriminating function gets its own set of caches, for example by closing over them. According to both the result of experimentation and some bits of documentation[12], it appears to be the case in SBCL. If, on the other hand, an implementation maintains a cache of effective methods *outside* the discriminating functions (for instance, directly in the generic function object), then, this implementation is guaranteed to *never* work.

7 PERFORMANCE

Because method combinators are implemented in terms of regular combinations, the cost of a (combined) generic call shouldn't be impacted. In SBCL, only the `standard` combination is special-cased for bootstrapping and performance reasons, so some loss could be noticeable with the `:standard` combinator. Method combinator updates or changes do have a cost, as clients need to be reinitialized, but this is not different from updating a regular generic function for a new method combination. Again, the only portable way to do so is also to completely reinitialize the generic function.

Alternative combinators, on the other hand, do come at a cost, and it is up to the programmer to decide whether the additional expressiveness is worth it. Using an alternative combinator for the first time is very costly, as the generic function will be reinitialized twice (hence affecting the next regular call to it as well) and client maintenance will be triggered. Once an alternative discriminating function has been memoized, an "alternative call" will essentially require looking it up in a hash table (twice if `find-method-combinator` is involved in the call) before calling it.

In order to both confirm and illustrate those points, some rough performance measurements have been conducted and are reported in Figure 5. The first batch of timings involve a generic function with

[12] http://www.sbcl.org/sbcl-internals/Discriminating-Functions.html#Discriminating-Functions

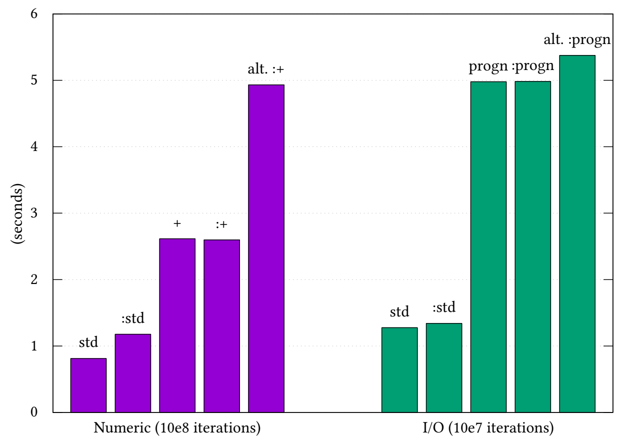

Figure 5: Benchmarks

4 methods simply returning their (numerical) argument. The second one involves a generic function with 4 methods printing their argument on a stream with `format`. The idea is that the methods in the numerical case are extremely short, while the ones performing I/O take much longer to execute. The timings are presented in seconds, for 10^8 and 10^7 consecutive iterations respectively.

The first two bars show the timings for a regular generic function with the standard method combination, and an equivalent combined generic function with the `:standard` combinator. In the numerical case, we observe a 45% performance loss, while in the I/O case, the difference is of 5%. This is due to SBCL optimizing the `standard` method combination but not the `:standard` combinator.

The next two bars show the timings for a built-in method combination compared to its equivalent combinator (`+` for the numerical case, `progn` for the I/O one). Recall that short combinators are in fact implemented as long ones, so the comparison is not necessarily fair. Nevertheless, the difference in either case is not measurable. Again, this is due to the fact that method combinators are implemented in terms of regular combinations.

Finally, the last bars show the timings involved in calling a generic function with an alternative combinator. Recall that this simply means calling a memoized discriminating function (hence taking the time displayed by the 4^{th} bars) after having looked it up in a hash table. The large number of iterations measured ensures that the overhead of first-time memoization is cushioned). In the first (numerical) case, the overhead of using the `:+` combinator as an alternative instead of as the original one is of 90%. The methods being very short, the impact of an additional has table lookup is important. In the (longer) I/O case and for the `:progn` combinator however, this impact is amortized and falls down to 8%.

8 RELATED WORK

Greg Pfeil has put up a set of useful method combination utilities on Github[13]. These utilities include functions and macros frequently used in the development of new combinations or helping in the debugging of their expansion, and also some pre-made ones.

His library addresses some of the orthogonality concerns raised in Section 2.1. In particular, the `append/nconc` combination allows one to switch between the `append` and `nconc` operators without the need for requalifying all the primary methods (they still need to be qualified as `append/nconc` though, so are short forms defined with the `basic` combination).

Greg Pfeil's library does not attempt to address the primary concern of this paper, namely the overall consistence of the design of method combinations, and more specifically their namespace behavior. In one particular case, it even takes the opposite direction. The `basic` combination implements an interesting idea: it serves as a unique short form, making the operator a use-time value. In this way, it is not necessary anymore to define short combinations globally before using them. Every short combination essentially becomes local to one generic function.

Note that even though we attempted to do the exact opposite with method combinators, it is also possible to use them locally. Indeed, one can always break the link from a name to a combinator by calling `(setf (find-method-combinator name) nil)`. After this, the combinator will only be shared by combined generic functions already using it. Again, this behavior is similar to that of `find-class`[14].

Finally, the `basic` combination also addresses some of the concerns raised in Section 2.2. On top of allowing `:before` and `:after` methods in short forms, the distinction between definition-time and use-time options is removed. Indeed, since the operator has become a use-time option itself, the same holds for the option `:identity-with-one-argument`. What we have done, on the contrary, is to turn the `order` option into a definition-time one (see Section 4.3.1).

9 CONCLUSION

Method combinations are one of the very powerful features of CLOS, perhaps not used as much as they deserve, due to their apparent complexity and the terse documentation that the standard provides. The additional expressiveness and orthogonality they aim at providing is also hindered by several weaknesses in their design.

In this paper, we have provided a detailed analysis of these problems, and the consequences on their implementation in SBCL. Basically, the under-specification or inconsistency of the associated protocols can lead to non-portable, obscure, or even surprising, yet conforming behavior.

We have also proposed an extension called *method combinators* designed to correct the situation. Method combinators work on top of regular combinations in a non-intrusive way and behave in a more consistent fashion, thanks to a set of additional protocols following some usual patterns in the CLOS MOP. The full code is available on Github[15]. It has been successfully tested on SBCL.

10 PERSPECTIVES

Method combinators are currently provided as a proof of concept. They still require some work and also raise a number of new issues. First of all, it is our intention to properly package them and provide them as an actual ASDF system library. Next, we plan on investigating their implementation for vendors other than SBCL,

[13] https://github.com/sellout/method-combination-utilities

[14] http://www.lispworks.com/documentation/lw70/CLHS/Issues/iss304_w.htm

[15] https://github.com/didierverna/ELS2018-method-combinators

and in particular figuring out whether alternative combinators are possible or not. As of this writing, the code is in fact already ported to CMUCL, but surprisingly enough, it doesn't work as it is. Most of the tests fail or even trigger crashes of the Lisp engine. It seems that CMUCL suffers from many bugs in its implementation of PCL, and it is our hope that fixing those bugs would suffice to get combinators working.

One still undecided issue is whether to keep long and short forms implemented separately (as in PCL), or unify everything under the long form. We prefer to defer that decision until more information on how other vendors implement combinations is acquired. The second issue is on the status of the long form's lambda-list (currently deactivated) and consequently whether new combinators should be represented by new classes or only instances of the general one (see Section 4.3.1).

As we have seen, the lack of specification makes it impossible to implement method combinators in a completely portable way, and having to resort to `reinitialize-instance` is overkill in many situations, at least in theory. Getting insight on how exactly the different vendors handle applicable and effective methods caches could give us hints on how to implement method combinators more efficiently, alternative combinators in particular.

Apart from the additional functionality, several aspects of method combinators and their protocols only fill gaps left open in the MOP. Ultimately, these protocols (generic function updating notably) should belong in the MOP itself, although a revised version of it is quite unlikely to see the day. It is our hope, however, that this paper would be an incentive for vendors to refine their implementations of method combinations with our propositions in mind.

Finally, one more step towards full orthogonality in the generic function design can still be taken. The Common Lisp standard forbids methods to belong to several generic functions simultaneously. By relaxing this constraint, we could reach full 3D separation of concerns. Method combinators exist as global objects, so would "floating" methods, and generic functions simply become mutable sets of shareable methods, independent from the way(s) their methods are combined.

ACKNOWLEDGMENTS

Richard P. Gabriel provided some important feedback on the history of method combinations, Christophe Rhodes some documentation on SBCL's internals, and Pascal Costanza and Martin Simmons some valuable insight or opinions on several aspects of the MOP.

REFERENCES

[1] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. Commonloops: Merging lisp and object-oriented programming. *SIGPLAN Notices*, 21(11):17–29, June 1986. ISSN 0362-1340. doi: 10.1145/960112.28700. URL http://doi.acm.org/10.1145/960112.28700.

[2] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common lisp object system specification. *ACM SIGPLAN Notices*, 23(SI):1–142, 1988. ISSN 0362-1340.

[3] Giuseppe Castagna. *Object-Oriented Programming, A Unified Foundation*. Progress in Theoretical Computer Science. Birkhäuser Boston, 2012. ISBN 9781461241386.

[4] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *SIGPLAN Lisp Pointers*, 5(1):182–192, January 1992. ISSN 1045-3563. doi: 10.1145/141478.141537. URL http://doi.acm.org/10.1145/141478.141537.

[5] Linda G. DeMichiel and Richard P. Gabriel. The common lisp object system: An overview. In *European Conference on Object Oriented Programming*, pages 151–170, 1987.

[6] Richard P. Gabriel and Kent M. Pitman. Technical issues of separation in function cells and value cells. *Lisp and Symbolic Computation*, 1(1):81–101, June 1988. ISSN 1573-0557. doi: 10.1007/BF01806178. URL https://doi.org/10.1007/BF01806178.

[7] Richard P. Gabriel, Jon L. White, and Daniel G. Bobrow. Clos: integrating object-oriented and functional programming. *Communications of the ACM*, 34(9):29–38, 1991. ISSN 0001-0782.

[8] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-61622-X.

[9] Sonja E. Keene. *Object-Oriented Programming in Common Lisp: a Programmer's Guide to CLOS*. Addison-Wesley, 1989. ISBN 0-20117-589-4.

[10] Gregor J. Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

[11] Patty Maes. Concepts and experiments in computational reflection. In *OOPSLA*. ACM, December 1987.

[12] Kristen Nygaard and Ole-Johan Dahl. The development of the SIMULA languages. *SIGPLAN Notices*, 13(8):245–272, August 1978. ISSN 0362-1340. doi: 10.1145/960118.808391. URL http://doi.acm.org/10.1145/960118.808391.

[13] Andreas Paepcke. User-level language crafting – introducing the CLOS metaobject protocol. In Andreas Paepcke, editor, *Object-Oriented Programming: The CLOS Perspective*, chapter 3, pages 65–99. MIT Press, 1993. Downloadable version at http://infolab.stanford.edu/~paepcke/shared-documents/mopintro.ps.

[14] Tim Sheard. Accomplishments and research challenges in meta-programming. In Walid Taha, editor, *Semantics, Applications, and Implementation of Program Generation*, volume 2196 of *Lecture Notes in Computer Science*, pages 2–44. Springer Berlin / Heidelberg, 2001. ISBN 978-3-540-42558-8.

[15] Brian C. Smith. Reflection and semantics in Lisp. In *Symposium on Principles of Programming Languages*, pages 23–35. ACM, 1984.

[16] ANSI. American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999), 1994.

A LONG SHORT METHOD COMBINATIONS

The Common Lisp standard provides several examples of built-in method combinations, and their equivalent definition in long form[16]. In a similar vein, the macro proposed in Figure 6 defines method combinations similar to those created with the short form, only with the following differences:

(1) the primary methods must not be qualified,
(2) `:before` and `:after` methods are available.

As in the original short form of `define-method-combination`, `identity-with-one-argument` is available as an optimization avoiding the call to the operator when a single method is invoked. The long form's lambda-list is used to define the `order` optional argument, directly passed along as the value of the `:order` keyword to the `primary` method group.

B LONG METHOD COMBINATION WOES

This section demonstrates an inconsistent behavior of generic functions using long method combinations in SBCL, when the combination is redefined. First, we define a `progn`-like long method combination, ordering the methods in the default, most specific first way.

```
(define-method-combination my-progn ()
  ((primary () :order :most-specific-first :required t))
  `(progn ,@(mapcar (lambda (method)
                      `(call-method ,method))
                    primary)))
```

Next, we define a generic function using it with two methods.

```
(defgeneric test (i) (:method-combination my-progn)
  (:method ((i number)) (print 'number))
  (:method ((i fixnum)) (print 'fixnum)))
```

[16] http://www.lispworks.com/documentation/lw70/CLHS/Body/m_defi_4.htm

```
(defmacro define-long-short-method-combination
    (name &key documentation identity-with-one-argument (operator name))
  "Define NAME as a long-short method combination.
OPERATOR will be used to define a combination resembling a short method
combination, with the following differences:
- the primary methods must not be qualified,
- :before and :after methods are available."
  (let ((documentation (when documentation (list documentation)))
        (single-method-call (if identity-with-one-argument
                              '`(call-method ,(first primary))
                              ``(,',operator (call-method ,(first primary))))))
    `(define-method-combination ,name (&optional (order :most-specific-first))
       ((around (:around))
        (before (:before)) ;; :before methods provided
        (primary (#| combination name removed |#) :order order :required t)
        (after (:after)))  ;; :after methods provided
       ,@documentation
       (flet ((call-methods (methods)
                (mapcar (lambda (method) `(call-method ,method)) methods)))
         (let* ((primary-form (if (rest primary)
                                `(,',operator ,@(call-methods primary))
                                ,single-method-call))
                (form (if (or before after)
                        `(multiple-value-prog1
                             (progn ,@(call-methods before) ,primary-form)
                           ,@(call-methods (reverse after)))
                        primary-form)))
           (if around
             `(call-method
               ,(first around) (,@(rest around) (make-method ,form)))
             form))))))
```

Figure 6: Long Short Method Combinations

Calling it on a `fixnum` will execute the two methods from most to least specific.

```
CL-USER> (test 1)
FIXNUM
NUMBER
```

Next, we redefine the combination to reverse the ordering of the methods.

```
(define-method-combination my-progn ()
  ((primary () :order :most-specific-last :required t))
  `(progn ,@(mapcar (lambda (method)
                      `(call-method ,method))
                    primary)))
```

This does not (yet) affect the generic function.

```
CL-USER> (test 1)
FIXNUM
NUMBER
```

We now add a new method on `float`, which normally reinitializes the generic function.

```
(defmethod test ((i float)) (print 'float))
```

However, a `fixnum` call is not affected, indicating that some caching of the previous behavior is still going on.

```
CL-USER> (test 1)
FIXNUM
NUMBER
```

A first `float` call, however, will notice the new combination function.

```
CL-USER> (test 1.5)
NUMBER
FLOAT
```

Meanwhile, `fixnum` calls continue to use the old one.

```
CL-USER> (test 1)
FIXNUM
NUMBER
```