A Mop-Based Implementation for Method Combinations

Method Combinators Revisited

Didier Verna
EPITA, LRE
Le Kremlin-Bicêtre, France
didier@lrde.epita.fr

ABSTRACT

In traditional object-oriented languages, the dynamic dispatch algorithm is hardwired to select and execute the most specific method in a polymorphic call. In Clos, the Common Lisp Object System, an abstraction known as *method combinations* allows the programmer to define their own dispatch scheme. When Common Lisp was standardized, method combinations were not mature enough to be fully specified.

In 2018, using Sbcl as a research vehicle, we analyzed the unfortunate consequences of this under-specification and proposed a layer on top of method combinations designed to both correct a number of observed behavioral inconsistencies, and propose an extension called "alternative combinators". Following this work, Sbcl underwent a number of internal changes that fixed the reported inconsistencies, although in a way that hindered further experimentation.

In this paper, we analyze Sbcl's new method combinations implementation and we propose an alternative design. Our solution is standard-compliant so any Lisp implementation can potentially use it. It is also based on the Mop, meaning that it is extensible, which restores the opportunity for further experimentation. In particular, we revisit our former "alternative combinators" extension, broken after 2018, and demonstrate that provided with this new infrastructure, it can be re-implemented in a much simpler and non-intrusive way.

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, Multiple Dispatch, Method Combinations, Orthogonality

ACM Reference Format:
Didier Verna. 2023. A Mop-Based Implementation for Method Combinations: Method Combinators Revisited. In *Proceedings of the 16th European Lisp Symposium (ELS'23).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.5281/zenodo.7818680

ELS'23, April 24–25 2023, Amsterdam, Netherlands

ACM ISBN 978-2-9557474-7-6.
https://doi.org/10.5281/zenodo.7818680

1 INTRODUCTION

Common Lisp was the first programming language equipped with an object-oriented (OO) layer to be standardized [14]. 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, 6, 8], departs from those in several important ways.

First of all, Clos offers native support for multiple dispatch [3, 4]. The existence of multi-methods pushes the dynamic dispatch one step further in the direction of separation of concerns: polymorphism and inheritance are clearly separated. Next, when implemented on top of the Mop [9, 11], the very semantics of Clos itself can be extended or modified, hence providing a form of homogeneous behavioral reflection [10, 12, 13].

Yet another improvement over classical OO 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, not necessarily by order of specificity, and combine their results in a particular way. Along with multiple dispatch, method combinations constitute one more step towards *orthogonality* [7, 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.

In a private conversation, Richard P. Gabriel reported that at the time Common Lisp was standardized, the committee didn't believe that 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.

In 2018 [15], using Sbcl[1] as a research vehicle, we analyzed the unfortunate consequences of this under-specification and exhibited a number of oddities in the design and behavior of method combinations. In particular, it turned out that method combinations weren't required to have a global name-space, meaning that every generic function could end up with its own method combination object,

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

completely disconnected from the original definition, and hence unaffected by subsequent modifications to it. It is worth mentioning that although counter-intuitive, this behavior does *not* contradict the standard. We proposed an extension to method combinations, called "method combinators", designed, amongst other things to establish proper dependencies between global method combination definitions and the associated generic functions. We were able to implement that extension in a non-intrusive, semi-portable way. This means that no modifications to SBCL's internals were needed; only a couple of calls to internal functions here and there. In addition to that, method combinators allowed us to develop an additional feature, *alternative combinators*, namely, the ability to call the same generic function with different method combinations at the same time, and at the minimum performance cost, that is, without the need to reinitialize the function every single time.

After this work, SBCL underwent a number of internal changes that fixed the reported inconsistencies. In particular, in its current state, the dependencies between generic functions and their method combinations are handled in a more intuitive fashion: generic functions are updated if their original method combination is redefined globally. Unfortunately for us, the new dependency management code is buried deep down into SBCL's internals and doesn't go through any of the official or even just suggested protocols. As a consequence, those changes broke our implementation of alternative combinators, and made it impossible to re-implement them as before, in a non-intrusive way.

In this paper, we propose yet another iteration over a possible implementation of method combinations. The paper is organized as follows. Section 2 provides an analysis of the current implementation in SBCL, emphasizing on how the dependencies between method combinations and generic functions are handled. Section 3 proposes an alternative, MOP-based implementation. This implementation conforms to the standard, so it could very well be used not only by SBCL but by all interested Lisp implementations. The design we propose also focuses on extensibility and experimentation, which was in fact the original motivation for leaving the method combinations area under-specified in the standard. Section 4 describes some additional refinements aimed at extensibility, and illustrates the benefits with a couple of examples. In particular, we revisit our former alternative combinators extension, and demonstrate that this time, it can be re-implemented in a much simpler and non-intrusive way. Finally, Section 5 provides some feedback on the performance of the proposed design.

2 METHOD COMBINATIONS IN SBCL

In this section, we analyze how post-2018 SBCL implements method combinations, and how it handles the dependencies between them and the generic functions in the system.

2.1 Method Combinations Hierarchy

The SBCL method combination classes hierarchy is depicted in Figure 1.

2.1.1 *Description*

The `method-combination` class is the only one mandated by the standard. The existence of a sub-hierarchy is nevertheless also a requirement, as the standard stipulates that method combination

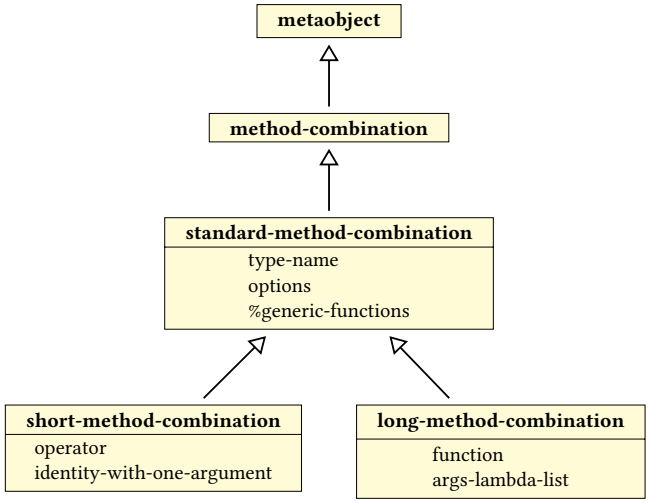

Figure 1: SBCL Method Combination Classes Hierarchy

objects be "indirect instances" of the `method-combination` class[2]; something that the MOP itself confirms by saying that this class should be "abstract".

Although, again, the standard does not require it, there is a `standard-method-combination` class, which is in fact a natural thing to provide. Indeed, it aligns the design of method combinations with the key components of CLOS which do have such a standardized equivalent: `standard-class`, `standard-generic-function`, and `standard-method` notably.

Apart from the standard method combination, every other one (that is, either built-in or user-defined) will be an instance of either the `short-method-combination`, or `long-method-combination` class.

2.1.2 *Analysis*

Already the case in 2018, a notable aspect of this implementation is the mixture of define-time and use-time attributes to method combinations.

The `type-name`, `operator`, `identity-with-one-argument`, and `args-lambda-list` slots represent information passed to `define-method-combination`. The `options` slot, on the other hand, holds specific sets of options passed to the `:method-combination` option in calls to `defgeneric`. As a consequence, different instances created from the same original method combination will only differ by their `options` slot.

One particular excerpt from the standard may explain this design, which is in fact that of PCL [1] rather than of SBCL itself. The following sentence appears in the description of the `method-combination` class[3].

> *A method combination object contains information about both the type of method combination and the arguments being used with that type.*

In PCL, the ability for a method combination instance to access information related to its original *type* is necessary anyway. Indeed,

[2] http://clhs.lisp.se/Body/t_meth_1.htm
[3] http://clhs.lisp.se/Body/t_meth_1.htm

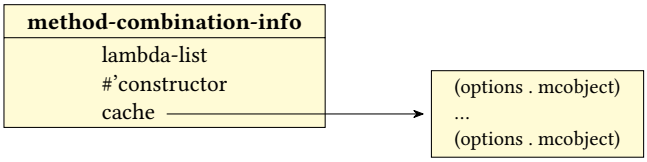

Figure 2: Sbcl Method Combination Info Structure

that information is used by the code computing effective methods when a generic function is called.

Note that even with this particular design, the information related to the method combination *type* is not exhaustive. The method combination's lambda-list is missing (it is in fact stored somewhere else), and so is the potential `:generic-function` option's value for long method combinations.

Finally, let us also mention that the `function` slot of long method combinations exists for historical reasons, but is not used anymore in Sbcl. Instead, Sbcl uses a global hash table mapping method combination names to such functions (stored in the `*long-method-combination-functions*` global variable). The functions in question are each method combination type's specific version of `compute-effective-method`, so there is indeed only one per method combination *type* (they are parameterized by the contents of the `options` slot).

2.1.3 Summary

From this analysis, it turns out that Sbcl's method combinations hierarchy, only slightly divergent from that of Pcl's, contains a mixture of information specific to every instance and information related to a method combination *definition* (in which case that information is duplicated). Two bits of information related to a method combination definition are also stored elsewhere, outside this hierarchy (the method combination's lambda list and function), and the `long-method-combination` class retains one obsolete, unused slot.

2.2 Dependency Management

One notable change in post-2018 Sbcl is a more natural handling of the dependencies between generic functions and method combinations. More specifically, method combinations have regained global name-space semantics, which means that should one of them be redefined, the generic functions using it would be notified. We now explain how this is done.

Each defined method combination is represented by an instance of a structure called `method-combination-info`, which is depicted in Figure 2. A global variable named `**method-combinations**` maintains a hash table mapping method combination names to such instances. The `lambda-list` slot stores the method combination's lambda-list. It is the one that was noted as missing from the hierarchy in Figure 1.

2.2.1 Instantiation

Every method combination info maintains a `cache` of method combination objects. When a generic function is defined to use a specific method combination with a specific set of `options`, the standard Mop function `find-method-combination` is called. If a method combination object associated with those particular options is found in the cache, it is simply returned. Otherwise, a new method combination object is created by calling the `constructor` function, and the cache is populated accordingly. Depending on the context, method combination objects will be instances of either the `short-` or `long-method-combination` classes.

2.2.2 Redefinition

Note, in Figure 1, the existence of a new slot (added post-2018 to Sbcl) named `%generic-functions` in the `standard-method-combination` class. This is how every method combination object keeps track of the generic functions using it.

When a method combination is redefined (by calling `define-method-combination` again), Sbcl updates the concerned info structure, and then traverses its cache, calling `change-class` on every method combination object. Also, for each "client" generic function in each method combination object's `%generic-functions` slot, Sbcl flushes the effective method cache and reinitializes the function by calling `reinitialize-instance`.

Note that this whole redefinition process is done in Sbcl's internals, without going through any standard (there is, in fact, none) or even just public protocols.

2.2.3 Updating

In a similar vein, when a generic function is created or updated, care is taken to add or remove it, to or from the `%generic-functions` slots in the concerned method combination objects. This time, the updating is done through a public protocol, namely, `[re]initialize-instance`.

2.2.4 Summary

Post-2018 Sbcl now handles the dependencies between method combinations and generic functions in a more intuitive way. Unfortunately, half of the dependency management code is buried in the implementation, without going through public protocols.

Note also that with the addition of the `method-combination-info` data structure, the global variable `*long-method-combination-functions*` has become superfluous. Indeed it could be replaced with an additional `function` slot in said structure, although that slot would be unused for short method combinations.

3 METHOD COMBINATIONS REVISITED

The lack of dependency management was our biggest concern in [15]. At the time, we were able to address it in a non-intrusive and extensible way. Although Sbcl's current solution works, it is buried deep down into the internals and doesn't go through any well-defined or public protocols. As a consequence, it is now impossible to continue experimenting with method combinations or providing extensions on top of them, without having to modify the language's implementation.

In this section, we suggest an alternative implementation for method combinations. In addition to proper dependency management, our implementation has the following properties.

- It remains standard-compliant.
- It retains Pcl's method combinations hierarchy (modulo some variations in the classes definitions).

- It clearly separates define-time and use-time method combination properties.
- It is grounded into the MOP. This means that it remains extensible and allows further experimentation, as was the original intent behind their under-specification, and as is, in general, the intent behind any MOP-based implementation of CLOS.

3.1 Overview

In the PCL implementation, and with the exception of the `standard` one, method combination objects are instances of one of the two built-in classes `short-` or `long-method-combination` (Figure 1). Yet, the standard consistently talks of method combination *types*[4][5], which seems to suggest that `define-method-combination` should create new *classes* of method combinations.

In addition to that, recall that `define-method-combination` comes in two forms, which means that there are in fact two *types of types of* method combinations. And so have we naturally entered the world of meta-objects.

The design we propose is thus as follows. We provide a hierarchy of method combination *types*, to distinguish between short and long ones. These are, in fact, meta-classes. `define-method-combination` is made to create a new method combination *class*, which is injected in PCL's method combinations hierarchy, and at the same time implemented as either a short or long method combination *type*. In other words, new method combination classes are sub-classes of either `short-` or `long-method-combination` as before, but are also *instances* of either `short-` or `long-method-combination-type`.

Further details are provided in the following sections. In the new hierarchies presented below, slots beginning with a percent sign contain information that is required for implementation purposes but are considered internal. Other slots are made publicly readable.

3.2 Method Combinations

Figure 3 depicts the updated method combinations hierarchy. It departs from PCL's in a number of ways.

First of all, the `standard-method-combination` class will not represent the built-in "standard method combination" anymore (there is, in fact, an ambiguity in the term). Rather, it exists as an intermediate implementation class similar to `standard-class`, `standard-generic-function`, or `standard-method`.

Also, this updated hierarchy doesn't hold any information related to the method combination *type* in use (information common to all instances). Instead, we only retain two slots: `options` (the options passed to the `:method-combination` option in calls to `defgeneric`), and `%generic-functions` (the cache of functions using this particular method combination object). As a consequence, the `short-` and `long-method-combination` classes are empty, and still exist only for specialization purposes.

3.3 Method Combination Types

Figure 4 depicts the added method combination types hierarchy; in other words, the hierarchy of method combination *meta-classes*. It

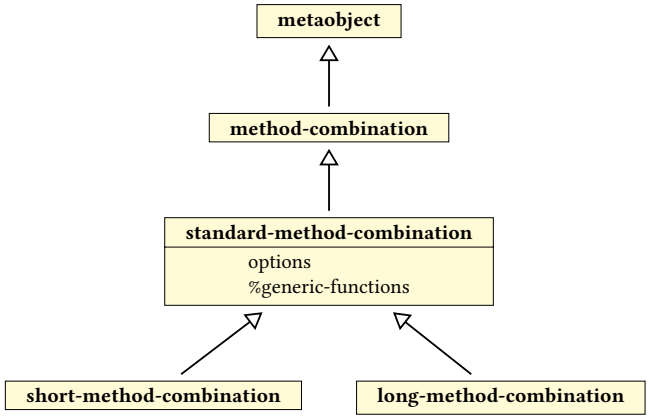

Figure 3: Method Combinations Hierarchy

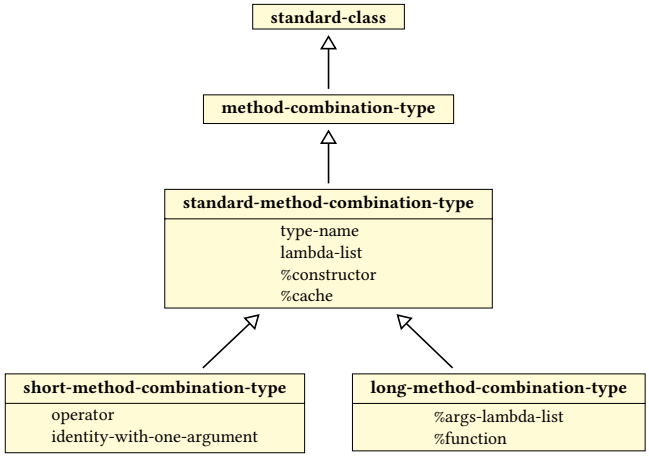

Figure 4: Method Combination Types Hierarchy

essentially serves as a replacement for SBCL's `method-combination-info` structure.

The `standard-method-combination-type` class holds the same information as the former info structure, with the addition of the method combination type's name. The former contents of the `short-` and `long-method-combination` classes, which was indeed common to all instances, is hence moved here, in the `short-` and `long-method-combination-type` classes. Note that in this new implementation, the `%function` slot will actually be used.

3.4 Standard Method Combination

As a first example of how those two hierarchies work together, let us now recreate the standard method combination. This is depicted in Figure 5.

We don't want to treat the standard method combination as an "exception" of any kind, and as mentioned before, we also want to remove any ambiguity around the term "standard" in this particular context. Because of that, the standard method combination *type*

[4] http://clhs.lisp.se/Body/m_defi_4.htm
[5] http://clhs.lisp.se/Body/m_defgen.htm

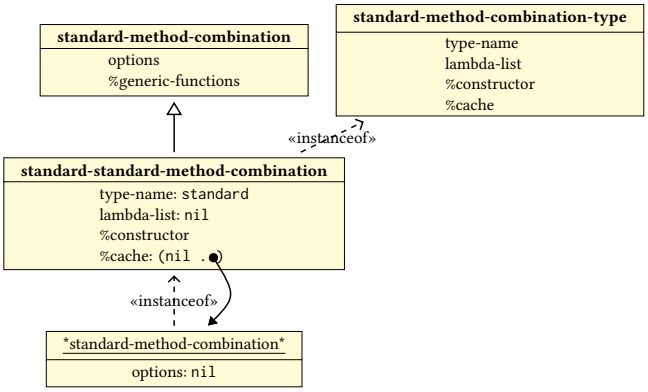

Figure 5: The Standard Method Combination

will be represented by a specific class (like any other method combination type). However, there will only ever be one standard method combination *object*, so the class in question will be a singleton one.

The standard method combination *type* is hence materialized by the singleton class `standard-standard-method-combination`. Because it is neither short nor long, it is a direct subclass of `standard-method-combination`, and it is directly implemented as a `standard-method-combination-type` for which the type name is `standard`, and the lambda-list is `nil`.

The standard method combination *object* (created by the `%constructor` function) is the only instance of that class, for which the options are also `nil`. Sbcl also happens to store that object in the global variable `*standard-method-combination*` for optimization purposes.

Finally, the `%cache` of method combination objects associates the options `nil` with the aforementioned single instance.

3.5 Built-In Method Combinations

Here we demonstrate how the built-in method combinations work as a second example. In Pcl, the built-in method combinations types are created using the short form of `define-method-combination`. Note that the creation of long method combination types works in exactly the same way. Figure 6 illustrates the effect of calling:

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

A new subclass of `short-method-combination` is created and implemented as a `short-method-combination-type`. The type name is `progn` (so is the `operator`), the lambda list is that of short method combinations and it falls back to identity with one argument, as specified in the call to `define-method-combination`.

Suppose now that two generic functions are created with:

```
(defgeneric gf1 (...)
  (:method-combination progn))
(defgeneric gf2 (...)
  (:method-combination progn :most-specific-last))
```

The `%constructor` function is called twice, resulting in the creation of two instances of the `progn` method combination type (`mc1` and `mc2`), each with the corresponding options. The method combination's `%cache` is populated accordingly. Finally, each method

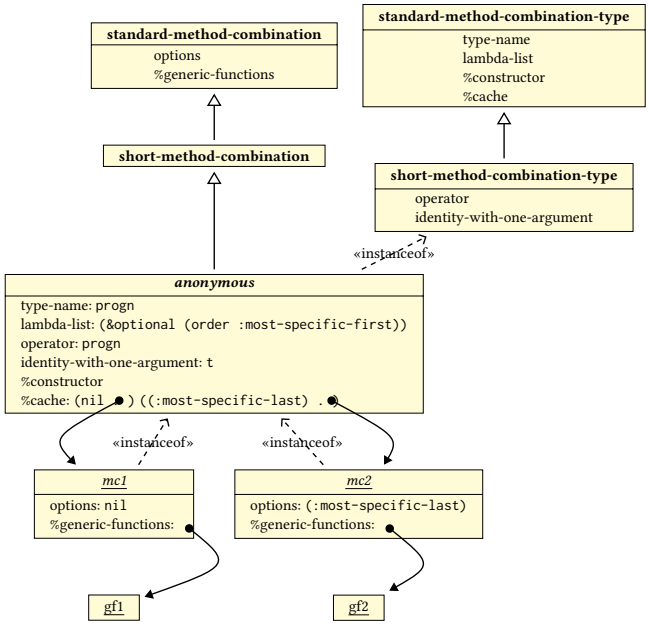

Figure 6: The `progn` Method Combination

combination object registers the concerned generic function as one if its "clients".

The reader may wonder why the `progn` method combination class is *anonymous* (especially since the standard method combination one isn't). The reason is that those classes are created automatically by the system, and as such, are not meant to be visible (even less so manipulable) by the programmer. We don't want them to "pollute" the global class name-space either. The fact that we provide a global name for `standard-standard-method-combination` is merely to facilitate the implementation. Sbcl provides three specializations on `compute-effective-method`; one on `short-method-combination`, one on `long-method-combination`, and one for the standard method combination type. Each user-defined method combination type will thus inherit automatically from one of the first two such methods. In the case of the standard method combination type, naming it explicitly (and statically) allows us to remain in the Mop's first layer (macro layer):

```
(defmethod compute-effective-method
    ((gf generic-function)
     (mc standard-standard-method-combination)
     applicable-methods)
...)
```

3.6 Implementation

We have implemented this approach in Sbcl. The resulting implementation is publicly available on Github, in a specific branch of our own Sbcl fork[6].

The implementation is in fact pretty straightforward, with the exception of one difficulty related to the bootstrapping of Clos. During that phase of the build, the Clos/Mop infrastructure is not

[6] https://github.com/didierverna/sbcl/tree/method-combination-types

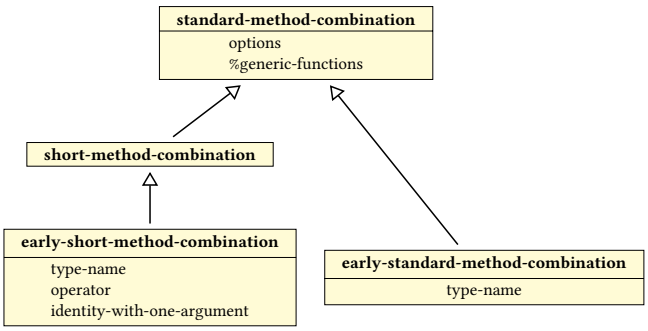

Figure 7: Early Method Combination Classes

fully available, and SBCL creates two "early" method combination objects: the `standard` and the `or` one. The difficulty is that a lot of the early CLOS code relies on those objects having the characteristics of the old infrastructure, which we have modified. In order to be as little intrusive as possible in the bootstrap, we use the following solution.

3.6.1 *Bootstrap*

We defer the creation of the method combination *types* hierarchy until *after* bootstrap. On the other hand, the updated method combinations hierarchy (Figure 3), which *is* available during bootstrap, is extended with two additional classes created specifically for the two initial method combination objects. These are `early-standard-method-combination` and `early-short-method-combination`, as depicted in Figure 7. As you can see, those two classes re-introduce the old slots that we moved around. With the appropriate accessors in place, the bootstrap code doesn't see the difference, and thus the required modifications are minimal: we just need to instantiate those "early" classes instead of the regular ones.

3.6.2 *Injection*

After bootstrap, the method combination types hierarchy is installed and the `standard` and built-in short method combination types are created. At that point, the complete new infrastructure is in place, but it is empty: we still have the two early method combination objects dangling around, and early generic functions using them.

These objects are in fact stored in two global variables named `*standard-method-combination*` and `*or-method-combination*`. Thus, it is easy for us to update the system: we transfer the generic function caches from these objects to the new ones, we update the existing generic functions to point to the new method combination objects, and we eventually re-assign the global variables to these new objects as well.

3.6.3 *Additions*

With that infrastructure in place, a number of suggestions already made in [15] can be re-implemented. In particular, we provide the following function which accesses the global method combination type name-space (analog to what `find-class` does).

```
find-method-combination-type
  (name &optional (errorp t))
"Find a NAMEd method combination type.
If ERRORP (the default), throw an error if no such
method combination type is found.
Otherwise, return NIL."
```

Also previously noted ([15, Section 2.3.1]) is the confusing nature of `find-method-combination`, which may be called with a method combination name and options (2^{nd} and 3^{rd} arguments) that do not correspond to the method combination actually in use by the generic function (1^{st} argument). Since the generic function argument to this protocol is in fact unused, it is easy to provide an alternative convenience function as follows.

```
find-method-combination*
  (name &optional options (errorp t))
"Find a method combination object for NAME and OPTIONS.
If ERRORP (the default), throw an error if no NAMEd
method combination type is found.
Otherwise, return NIL.

Note that when a NAMEd method combination type exists,
asking for a new set of (conformant) OPTIONS will
always instantiate the combination again, regardless
of the value of ERRORP."
```

4 EXTENSIBILITY

Establishing a clear distinction between the properties of method combination types and those of method combination objects certainly is a good thing from a software engineering point of view. On the other hand, it may seem overkill to introduce a meta-class hierarchy to do so. Indeed, the existence of duplicated information in the original hierarchy is not a critical problem; a simpler alternative to avoid duplication could have been the use of `:allocation :class` slots, *etc.*

What the proposed design gives back, however, is something quite valuable, especially in the general context of CLOS and the MOP, and that is *extensibility*. Recall that one of the original reasons for the general fuzziness around method combinations in the standard was to leave room for experimentation. The CLOS MOP is notoriously good at that when it provides (meta-)class hierarchies to extend, and protocols to specialize.

With the proposed design, it becomes possible to push experimentation with method combinations further, and in a less intrusive way, by extending the method combination and/or method combination types hierarchies separately.

4.1 Protocol refinements

In addition to the hierarchies proposed in the previous section, a number of refinements can be made to the current implementation to ease experimentation.

4.1.1 *Method Combination Types Redefinition*

When a method combination type is redefined, the current implementation in `sb-pcl` calls `change-class` on every concerned instance, and then reinitializes every (dependent) generic function listed in the instances caches. Here, we provide two small refinements for extensibility.

First, the code dealing with generic function reinitialization is installed in an `:after` method on `update-instance-for-different-class`. This allows potential extensions to get notified if a method combination type has changed.

Next, the code in question is wrapped in a new protocol named `u-g-f-f-r-m-c`[7], a protocol that was already proposed in [15]. This, in turn, allows potential extensions to generic functions to be notified when their method combination is updated.

4.1.2 *Method Combination Types Definition*

Finally, there is a simple way to allow extensions to seamlessly plug sub-classes of method combination (types) into the system. According to the Common Lisp standard (in particular Section 1.6 Language Extensions[8]), it is permissible to add new keyword arguments to functions or macros, provided that "they do not alter the behavior of conforming code and provided they are not explicitly prohibited [...]". Consequently, we can extend `define-method-combination` in the following manner (granted, this is just a macro so it wouldn't be difficult to provide a different one instead).

The short form is made to understand two additional options, the meaning of which should be self-explanatory.

```
:method-combination-class name
:method-combination-type-class name
```

or

```
:method-combination-type-class (name initargs*)
```

Similarly, the long form is made to recognize those as well, provided that they appear in that order, and only *after* the `:arguments` and `:generic-function` options when present.

```
(:method-combination-class name)
(:method-combination-type-class name initargs*)
```

Of course, care is taken to verify that when provided, the alternative classes make sense in the present context, and with each other (*e.g.* only sub-classes of `short-method-combination[-type]` are authorized in the short form, *etc.*).

We now provide two examples making use of this kind of extensibility. The complete code is available on Github[9] and requires the aforementioned fork of Sbcl to work.

4.2 Medium Method Combinations

We want to define "medium" method combinations, that is, method combinations behaving like short ones, but also equipped with `:before` and `:after` methods, and which do not request the qualification of primary methods. Of course, these may be defined as regular long method combinations (*all* method combinations can). However, we may want to keep the short-style `operator` and `identity-with-one-argument` properties around, for information purposes (*e.g.* specializing `print-object` or producing detailed reference manuals with Declt[10]).

We hence provide a new method combination type class, as depicted in Figure 8. A new convenience macro could be used as below:

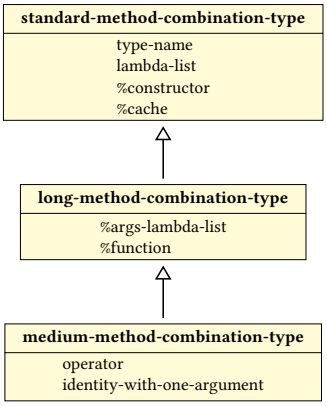

Figure 8: The Medium Method Combination Type Class

```
(define-medium-method-combination myprogn
  :operator progn :identity-with-one-argument t)
```

which, in turn, will expand to this:

```
(define-method-combination myprogn
    (&optional (order :most-specific-first))
  ((around (:around))
   (before (:before))
   (primary () :order order :required t)
   (after (:after)))
   (:method-combination-type-class
       medium-method-combination-type
     :operator progn :identity-with-one-argument t)
  ...)
```

Because this new method combination type is fully integrated into the original hierarchy, nothing else is required for it to work. In particular, Sbcl's original specialization on `compute-effective-method` for long method combinations remains applicable here.

4.3 Alternative Method Combinations

Alternative method combinations have been proposed and described in [15, Section 6]. In short, the idea is to be able to call the same generic function with different method combinations efficiently (meaning, without having to reinitialize it at every call), simply by maintaining a cache of discriminating functions.

Just as a quick reminder of a potential use-case, assume that access to alternative calls is provided through a reader-macro such as this one: `#!combination(func arg1 arg2 ...)`. It may be convenient, depending on the context, to vary the calls to a function at minimal cost like this:

```
#!append(func arg1 arg2 ...)
#!nconc(func arg1 arg2 ...)
```

Given the new method combination architecture proposed in Section 3, the implementation of this idea is not only straightforward, but also much simpler than that of 2018. In fact, we don't even need to extend the method combination type hierarchy anymore; only the generic functions one.

[7] `update-generic-function-for-redefined-method-combination`
[8] http://clhs.lisp.se/Body/01_f.htm
[9] https://github.com/didierverna/ELS2023-method-combinations
[10] https://www.lrde.epita.fr/~didier/software/lisp/typesetting.php#Declt

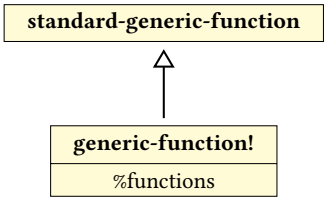

Figure 9: Extended Generic Functions

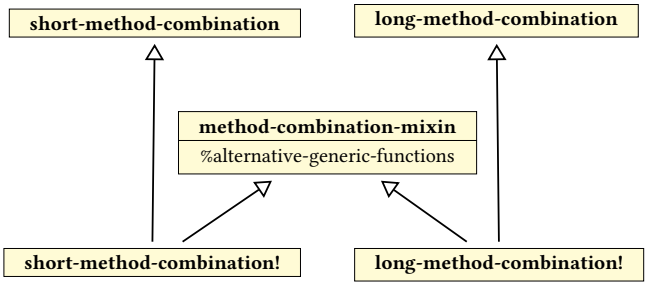

Figure 10: Extended Method Combinations

4.3.1 *Alternative Calls*

We provide a new class of generic functions, as depicted in Figure 9. The `%functions` slot implements the discriminating functions cache. It is a hash table mapping method combination objects to discriminating functions. When a generic function is called with an alternative method combination, it is reinitialized with that method combination, called, and the resulting discriminating function is cached. The function is then switched back to its original method combination.

Note that we also arrange for the two method combination objects (the original and the alternative one) to reference the generic function in their respective cache. This is why we don't need to extend the method combinations hierarchy anymore, but this means that the caches in question contain a mixture of generic functions using the method combination as their "primary" one, and others using it as an alternative one.

Another possible implementation would be to maintain separate caches for primary and alternative generic functions, in which case the method combinations hierarchy (*not* the method combination types one) would need to be extended as depicted in Figure 10.

4.3.2 *Generic Function Modification*

New `:after` methods are installed on `add-method` and `remove-method` to clear the discriminating functions cache in case the generic function is modified.

4.3.3 *Method Combination Change*

An `:around` method on `reinitialize-instance` is installed in order to intercept a method combination change to the generic function. On top of the normal behavior, if the new method combination was previously used as an alternative one for this generic function, both the generic function's discriminating functions cache, and the method combination's generic functions cache are updated.

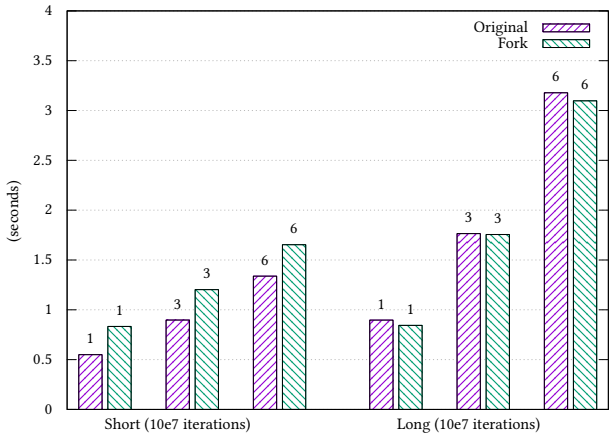

Figure 11: `compute-effective-method` Performance

4.3.4 *Method Combination Redefinition*

Finally, a new method on `u-g-f-f-r-m-c`[11], one of our newly proposed protocols, is installed. This method simply detects the redefinition of a method combination that was used as an alternative one, and invalidates the cached discriminating function associated with it.

5 PERFORMANCE

In this section, we study the impact of our proposed architectural changes on the performance of the system. We are *not* interested in benchmarking the creation or modification of method combinations, since that is bound to happen very rarely. Rather, the impact of the method combinations implementation is likely to be visible where they are *used*, that is, when effective methods are computed. In other words, we are interested in the performance of `compute-effective-method`.

In Sbcl, there are three different cases.

(1) The case of the standard method combination is completely hardwired, so regardless of its implementation, the performance will be exactly the same.
(2) The case of short method combinations is handled by a *single* function, `short-compute-effective-method`, but this function accesses properties specific to the method combination *types* every time it is called (type name, operator, identity with one argument).
(3) Finally, the case of long method combinations is handled by calling a function that is specific to each method combination type.

In order to roughly evaluate the consequences of our proposed architecture in terms of performance, we timed the execution of `compute-effective-method` (ten million calls in a row) on one generic function using a short method combination, and another one using a long method combination, each time with one, three, and six applicable methods. Those tests were run on a regular Sbcl as well as on our forked version. The code is available in the aforementioned Github repository, and the results are presented in

[11] `update-generic-function-for-redefined-method-combination`

Figure 11. The number of applicable methods is indicated on top of each bar.

5.1 Short Method Combinations

In the case of short method combinations, we observe a degradation ranging from 50% to 22% (the degradation decreasing as the number of applicable methods increases). This may be explained as follows.

In the original version of Sbcl, `short-compute-effective-method` retrieves three method combination properties (type name, operator, and identity with one argument) through regular accessors to the slots depicted in Figure 1.

In our new architecture, those properties belong to the method combination *type* rather than to the method combination *object*. As visible in Figure 6, accessing those properties from a method combination object hence requires an additional call to `class-of`. In other words, we are comparing `(accessor mc-object)` with `(accessor (class-of mc-object))`. Also, because the number of such accesses remains constant (exactly three), it is not surprising that the impact on performance decreases when the number of applicable methods increases: it just means that it takes longer to execute the function with more applicable methods.

Even though such a degradation may seem important, we still don't think it matters that much, in the sense that effective methods are not computed very frequently (thanks to caching); only when the set of applicable methods varies. And if it does matter, it is always possible to duplicate that information back into the method combination objects themselves, without sacrificing the new architecture.

5.2 Long Method Combinations

In the case of long method combinations on the other hand, we observe an *improvement* ranging from 7% to 3% (also less important as the number of applicable methods increases), which may or may not be considered significant. Again, this may be explained as follows.

In the original version of Sbcl, `compute-effective-method` retrieves the method combination "function" (in charge of actually computing the effective method in a way specific to that particular method combination type) from a hash table (see Section 2.1.2).

In our new architecture (see Figure 4), that function is stored in the method combination *type* itself, that is, in the *implementation* of the method combination object. So here this time, we are comparing a hash table lookup with `(accessor (class-of mc-object))`.

6 CONCLUSION AND PERSPECTIVES

Method combinations are an extremely powerful, yet somewhat obscure part of Clos. The arguable complexity of `define-method-combination`'s long form is a probable obstacle to a more widespread use, and their under-specification doesn't facilitate experimentation, as there is almost no official protocol that Lisp implementations need to conform to.

In this paper, we have proposed a Mop-based implementation for method combinations. We believe that this implementation presents a number of advantages compared to vendor-specific solutions. First, it reifies the notion of method combination *type*, which, in fact, seems quite a natural thing to do upon careful reading of the Common Lisp standard. It is also standard-compliant, which gives us hope that it would trigger some general interest across the existing Lisp implementations. It remains close to Pcl's original design, notably by maintaining a hierarchy distinguishing short and long method combinations (it merely adds to it). Finally, and this is probably the strongest point, it makes the method combination infrastructure extensible, which really is the philosophy behind the Mop and puts this area of Clos back in line with the rest of it. The proposed architecture as been implemented in an Sbcl fork and is publicly available.

In Section 4, we have presented several simple examples demonstrating how we can benefit from an extensible design for further experimentation. There are still a number of things that we plan to work on. One of them is turning the `:description` option of the long form's method group specifiers into something useful for documentation purposes (Declt would like very much to have that). The Pcl implementation seems to ignore that option completely. Arguably, this would not be done as an extension but rather in the core of the architecture.

Another potential area of research is to extend method combinations to forms that would be neither short, nor long. Provided with an alternative to the `define-method-combination` macro which can't be used anymore, the proposed architecture makes it easy to do so: one simply has to sub-class both `standard-method-combination` and `standard-method-combination-type`, and provide additional methods on `compute-effective-method`.

Granted, every possible method combination type can be created with the long form of `define-method-combination`, since it is always possible to provide a single method group matching *all* applicable methods (using `*` as the pattern), and do everything in the method combination's *form*. So the question is not whether it is possible to do it, but rather how easy it is. We can already think of several investigation routes to ease the creation of complex method combinations: alternative semantics for the long form's method groups, such as the ability to re-use the same method in multiple groups or to specify qualifier patterns with regular expressions.

More generally, every time a method combination type is neither as simple as a short one nor as general as a long one (the "medium" type from Section 4.2 falls into that category), there might be a gain in defining it as an *intermediate* form, with a tailored method combination function, leading eventually to improving the performance of `compute-effective-method`.

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, 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.

[7] 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.

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

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

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

[11] 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.

[12] 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.

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

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

[15] Didier Verna. Method combinators. In *11th European Lisp Symposium*, pages 32–41, Marbella, Spain, April 2018. ISBN 9782955747421. doi: 10.5281/zenodo.3247610.