Representing method combinations

Robert Strandh

LaBRI, University of Bordeaux

April, 2020

European Lisp Symposium, Zürich, Switzerland
Context: The SICL project

https://github.com/robert-strandh/SICL
Several objectives:

▸ Create high-quality *modules* for implementors of Common Lisp systems.
▸ Improve existing techniques with respect to algorithms and data structures where possible.
▸ Improve readability and maintainability of code.
▸ Improve documentation.
▸ Ultimately, create a new implementation based on these modules.
A feature of object-oriented programming languages.

Can be thought of as a collection of *methods* sharing the same name.

In traditional languages, a generic function is not a first-class object.

In Common Lisp a generic function is an instance of a subclass of function, so it is a first-class object.
Generic-function invocation

In traditional languages: `<arg1>.gf(<arg2>, ...)`

In Common Lisp (gf `<arg1> <arg2> ...`)

Because of subclassing, an invocation may result in several methods being applicable.
Traditional generic-function invocation

In traditional languages, only the most specific applicable method is invoked.

Invoking less specific methods requires an explicit call, such as `super(...)`. Explicit code is required to combine the results of the different applicable methods being called.
Generic-function invocation in Common Lisp

The applicable methods are *combined* in some way.

The combination is expressed as a lambda expression called the *effective method*.

The effective method contains calls to the individual methods.

The compiler is called in order to turn the effective method into an *effective method function*.

When applied to the arguments of the invocation of the generic function, the effective method function calls the methods in the right order.
compute-effective-method

This generic function has three parameters:

1. A generic-function metaobject
2. A method-combination metaobject
3. A list of method metaobjects

A call to this generic function returns an an effective method.
In `compute-effective-method`, the method-combination metaobject designates a *method-combination procedure*.

This procedure can be implemented in different ways:

- As a method on `compute-effective-method`, specialized to some method-combination class.
- As a function stored in the method-combination object.
- Any other way that will accomplish the task.
Method combinations in the Common Lisp standard

A system class method-combination is defined.

Method-combination metaobjects must be *indirect instances* of this class.

A method-combination metaobject contains information about its type and the *arguments* used with that type.
Example use of method combinations

(defgeneric foo (x)
  (:method-combination and :most-specific-last))

Here, and is the name of a method-combination type and :most-specific-last is an argument that this particular type accepts.

The applicable methods are combined with the and standard operator, and the arguments to the operator are the return values of the invocations of the applicable methods in the order of the least specific to most specific.
define-method-combination

This operator defines a method-combination type

It has two versions: the short version and the long version.
define-method-combination (short version)

(define-method-combination name options)

Options are unimportant for this presentation.

The short version defines a method-combination type that accepts an optional argument that can be :most-specific-first or :most-specific-last.

If no argument is given, it defaults to :most-specific-first.
define-method-combination (long version)

(define-method-combination name lambda-list ... body)

The ... represent information that is not important to this presentation.

The lambda-list is an ordinary lambda list that specifies what arguments can be given after :method-combination.
1. The user defines a method-combination type using `define-method-combination`, then uses that type in a `defgeneric` form, but makes a mistake in the arguments.

2. The user defines a method-combination type using the long version of `define-method-combination`, but makes a mistake in the lambda list. Then uses `defgeneric` with the intended arguments.

3. The user defines a method-combination type using the long version of `define-method-combination`, then uses `defgeneric` with acceptable arguments. Later, the user modifies the lambda list and redefines the method-combination type.
Previous work

We investigated several FLOSS implementations:

- Portable Common Loops (PCL). A mostly portable library used for adding CLOS to a pre-ANSI Common Lisp implementation.
- Steel Bank Common Lisp (SBCL)
- Clozure Common Lisp (CCL)
- Embedded Common Lisp (ECL)
- Clasp

Many of them fail several scenarios.

All of them sometimes fail to verify the validity of method-combination arguments early. Instead, the arguments are sometimes verified when compute-effective-method is called, resulting in cryptic error messages.

Details in the paper.
Our technique
Our technique

In this presentation, we cover only our technique for reporting incorrect method-combination arguments early.

Refer to the paper for the way we handle the other scenarios.
Our technique

Recall the long version of define-method-combination: 
(define-method-combination name lambda-list ... body )

The lambda-list is an ordinary lambda list that specifies what arguments can be given after :method-combination.

We translate the short version to the long version.

We analyze the lambda-list of the long version and extract all the parameters (say v1, ..., vn) that can be used in the body.
Our technique

We construct a lambda expression as follows:

\[(\text{lambda} (...) \ (\text{list} \ v_1 \ ... \ v_n))\]

where (...) is the original lambda list.

We compile the lambda expression to obtain a function.

When this function is applied to the method-combination arguments, it returns a list of objects that can be used to identify a particular method-combination metaobject of a given type.

This list is used for recycling existing method-combination metaobjects.
Our technique

In our define-method-combination forms, we include &aux parameters with expressions that verify the arguments.

Example of long version for method-combination type and:

(define-method-combination and
  (&optional (order :most-specific-first)
  &aux (ignore (unless (member order
    '(:most-specific-first
      :most-specific-last))
    (error ...))))
  ...)
Our technique

When our constructed function is applied to some incorrect method-combination arguments:

- It may fail because the arguments are incompatible with the parameters of the lambda list.
- It may fail because one or more `&aux` initialization forms call `error`.

In both cases, we handle the error and report incorrect arguments.
The technique described here has been only partially incorporated into SICL. We are working on finishing this incorporation.

Our technique could benefit from some weak data structure to avoid memory leaks due to generic-function metaobjects being stored in our data structure. SICL does not yet have any such data structure. We plan to add it.
Acknowledgments

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