Method Combination in Julia

António Menezes Leitão

April, 8, 2022

1 Introduction

Method combination is an advanced concept that extends the single- and multiple-dispatch approach that is used in object-oriented languages. Method combination was initially proposed for Flavors, an object-oriented extension of Lisp Machine Lisp that, later, became ZetaLisp and, finally, was standardized in the Common Lisp Object System (CLOS). Few programming languages implement method combination, instead, forcing the programmer to rely on *super* method calls that, in the end, also force the use of rigid class hierarchies.

Unfortunately, it is usually difficult to implement method combination in languages that were not designed to be extended and this has been one of the biggest obstacles for the wider acceptance of method combination.

There are programming languages, however, that promote language extensions and the Julia programming language is in that category.

2 Goals

Implement, in Julia, the ability to have generic functions, multi-dispatch methods, and method combination following the way those ideas were implemented in the Common Lisp Object System. It is important to note that Julia already provides its own version of generic functions and multiple dispatch methods but it does not support method combination. This means that you will need to implement your own version of these concepts, although you might find advantageous to use the ones already available in Julia to support your implementation in a meta-circular style.

You must implement, at the very least, the features described in the following sections:

2.1 Generic Functions

You must implement syntax for the definition of a generic function that includes its name and the list of mandatory parameters. Optionally, the definition might include, at the end, a symbol describing the method combination to use, which might be `standard` (the default) or `tuple`. For example, consider the following syntax for the definition of a generic function that "adds" things using standard method combination:

```
@defgeneric add(x, y)
```

Suggestion: in order to implement generic functions, you will have to reify them, possibily, using Julia's `struct`s. To extend Julia syntax to support generic function definitions, you will have to explore Julia's meta-programming capabilities.

2.2 Methods

You must implement syntax for the definition of a method, including the name of the generic function to which it belongs, the list of parameters, and the body of the method. Each parameter must include the name of the parameter and a type that determines the applicability of the method in what regards that parameter. Optionally, the method definition might also include the symbol `before`, `primary`, or `after`.

As an example, consider the following syntax for the definition of two methods of the generic function `add`:

```
@defmethod add(x::Int, y::Int) =
  x + y

@defmethod add(x::String, y::String) =
  x * y
```

Suggestion: use meta-programming to translate a method definition into a more complex expression that injects a representation of a method into the representation of a generic function.

2.3 Method Combination

So far, the intended generic functions and methods are similar to those already available in Julia. To demonstrate this, consider the following generic function (and corresponding methods) intended to *explain* its argument:

```
@defgeneric explain(entity)

@defmethod explain(entity::Int) =
  print("$entity is a Int")

@defmethod explain(entity::Rational) =
  print("$entity is a Rational")

@defmethod explain(entity::String) =
  print("$entity is a String")
```

The following interaction illustrates the current behavior:

```
> explain(123)
123 is a Int

> explain("Hi")
Hi is a String

> explain(1//3)
1//3 is a Rational
```

There is, however, an important difference between our implementation and Julia's predefined one: we can explore *method combination*. In this case, we will illustrate the *standard* method combination, which supports not only *primary* methods, but also *before* and *after* auxiliary methods.

For example, after adding the following method:

```
@defmethod after explain(entity::Int) =
  print(" (in binary, is $(string(entity, base=2)))")
```

the behavior changes:

```
> explain(123)
123 is a Int (in binary, is 1111011)

> explain("Hi")
Hi is a String

> explain(1//3)
1//3 is a Rational
```

As a further example, suppose we also added the following definition:

```
@defmethod before explain(entity::Real) =
  print("The number ")
```

In this case, the behavior should be the following:

```
> explain(123)
The number 123 is a Int (in binary, is 1111011)

> explain("Hi")
Hi is a String

> explain(1//3)
The number 1//3 is a Rational
```

Besides the *standard* method combination, you must also implement the *tuple* method combination, which combines the results of all applicable methods in a tuple. To exemplify, consider the following generic function that includes the `tuple` keyword:

```
@defgeneric what_are_you(entity) tuple
```

Using this function, we can add a few specializations:

```
@defmethod what_are_you(n::Int64) =
  "I am a Int64"

@defmethod what_are_you(n::Float64) =
  "I am a Float64"

@defmethod what_are_you(n::Number) =
  "I am a Number"
```

Note that it is not necessary to specify the intended method combination in the method definition, as that information is already available in the corresponding generic function.

The following interaction illustrates the intended precedence between methods and also dynamic method definition:

```
> what_are_you(123)
("I am a Int64", "I am a Number")

> what_are_you(1.23)
("I am a Float64", "I am a Number")

> what_are_you(1//3)
("I am a Number",)

> @defmethod what_are_you(n::Rational) = "I am a Rational"

> what_are_you(123)
("I am a Int64", "I am a Number")

> what_are_you(1.23)
("I am a Float64", "I am a Number")

> what_are_you(1//3)
("I am a Rational", "I am a Number")
```

2.4 Generic Function Calls

You must implement the application of a generic function to concrete arguments.

Given a generic function *gf* and a list of arguments *args*, it is necessary to (1) select the set of applicable methods of *gf*, (2) sort them according to their specificity, (3) combine them according to the *gf* method combination and the applicable methods' role in that combination, (4) generate the effective method, (5) save it in a cache so that future applications of that method avoid the previous steps and, finally, (5) apply that effective method to *args*. If there is no applicable method, the generic function `no_applicable_method` is called, using the *gf* and *args* as its two arguments. The default behavior of that function is to raise an error. Note that additional method definitions (or redefinitions) will need to invalidate the effective methods cache.

As an example, considering the previous method definitions for the generic function `add`, the following interaction should be expected:

```
> add(1, 2)
3
> add("Foo", "Bar")
"FooBar"
> add(1, "Bar")
ERROR: No applicable method for arguments (1, "Bar") of types (Int64, String)
```

You might assume that precedence among applicable methods is determined by left-to-right consideration of the parameter types. Method m_1 is more specific than method m_2 if the type of the first parameter of m_1 is more specific than the type of the first parameter of m_2. If they are **identical** types, then specificity is determined by the next parameter and so on.

2.5 Extensions

You can extend your project to further increase your grade. Note that this increase will not exceed **two** points that will be added to the project grade for the implementation of what was required in the other sections of this specification.

Be careful when implementing extensions, so that extra functionality does not compromise the functionality asked in the previous sections. To ensure this behavior, you should implement all your extensions in a different file.

Some of the potentially interesting extensions include:

- Allowing generic functions to accept an argument precedence order option similar to the one used in CLOS.
- Supporting other strategies for method combination.
- Implementation of a class system with multiple inheritance that automatically establishes the subtyping relation.
- Providing a meta-object protocol that allows subclasses of generic functions and methods to have different behavior.

3 Code

Your implementation must work in Julia, version 1.7.2.

The written code should have the best possible style, should allow easy reading and should not require excessive comments. It is always preferable to have clearer code with few comments than obscure code with lots of comments.

The code should be modular, divided in functionalities with specific and reduced responsibilities. Each module should have a short comment describing its purpose.

4 Presentation

For this project, a full report is not required. Instead, a public presentation is required. This presentation should be prepared for a 10 minutes slot, should be centered in the architectural decisions taken and might include all the details that you consider relevant. You should be able to "sell" your solution to your colleagues and teachers.

5 Format

Each project must be handled by electronic means using the Fénix Portal. Each group must handle a single compressed file in ZIP format, named as `project.zip`. Decompressing this ZIP file must generate a folder named `g##`, where `##` is the group's number, containing the source code, within subdirectory `/src/`.

The only accepted format for the presentation slides is PDF. This PDF file must be submitted using the Fénix Portal separately from the ZIP file and should be named `presentation.pdf`.

6 Evaluation

The evaluation criteria include:

- The quality of the developed solutions.
- The clarity of the developed programs.
- The quality of the public presentation.

In case of doubt, the teacher might request explanations about the inner working of the developed project, including demonstrations.

7 Plagiarism

It is considered plagiarism the use of any fragments of programs that were not provided by the teachers. It is not considered plagiarism the use of ideas given by colleagues as long as the proper attribution is provided.

This course has very strict rules regarding what is plagiarism. Any two projects where plagiarism is detected will receive a grade of zero.

These rules should not prevent the normal exchange of ideas between colleagues.

8 Final Notes

Don't forget Murphy's Law.

9 Deadlines

The code must be submitted via Fénix, no later than 23:00 of **April**, **24**. Similarly, the presentation must be submitted via Fénix, no later than 23:00 of **April**, **24**.

The presentations will be done during classes after the deadline. Only one element of the group will present the work. The element will be chosen by the teacher just before the presentation. Note that the grade assigned to the presentation affects the entire group and not only the person that will be presenting. Note also that content is more important than form. Finally, note that the teacher may question any member of the group before, during, and after the presentation.