



Metaobject Protocols for Julia

Marcelo Santos  

Instituto Superior Técnico, University of Lisbon, Portugal

António Menezes Leitão  

INESC-ID/Instituto Superior Técnico, University of Lisbon, Portugal

Abstract

Metaobject Protocols enable programmers to extend programming languages without the need to understand the lower level details of their implementation. However, designing these protocols comes with two challenges: allow programmers to limit their concerns to higher level concepts and minimize performance penalties in programs. In this work, we propose metaobject protocol for the programming language Julia. Julia’s object system is very limited, when compared to languages following the Object-Oriented paradigm. However, Julia’s compilation approach allows for a considerable degree of code optimization through the exploration of runtime type information. Through the usage of Julia’s run-time optimizations, we propose a metaobject protocol that combines user-extensibility with limited performance penalties. This paper focuses on the development of a multiple inheritance method dispatch and method combination mechanisms with zero runtime overhead.

2012 ACM Subject Classification Software and its engineering → Language features

Keywords and phrases Julia, Metaobject Protocols, Object-Oriented Programming, Performance

Digital Object Identifier 10.4230/OASICS.SLATE.2022.13

Supplementary Material *Software (Source Code)*: <https://github.com/tosmarcel/julia-mop>
archived at `swh:1:dir:b1f9259d39045b00963846ba7d4b8a3dd3829971`

Funding This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with references UIDB/50021/2020 and PTDC/ART-DAQ/31061/2017.

1 Introduction

Traditionally, there has been a separation between programmers and language designers. Programmers treat languages as black boxes with their own rules, as established by the language designers. In this model, the semantics of a language is seen as unchangeable, thus imposing limits to its expressiveness.

Metaobject Protocols (MOPs) come to blur the distinction between programmers and language designers, by providing programmers with an interface to modify the language. By treating the language itself as a mutable object-oriented program, one can alter the semantics and introduce new behaviours to the language through the abstractions made available by the OOP paradigm. We provide two examples as motivation for the use of MOPs:

- As shown in [8], a programmer might be discontent with the performance of the implementation of slot accessing for objects of class A, but an implementation satisfying performance requirements for class A could damage performance for slot accessing for objects of class B. This, in turn, demonstrates the impossibility of the creation of an implementation satisfying every programmer’s needs. With the use of MOPs, one can define a specific implementation for only one class and leave the default implementation for the remaining classes.
- With MOPs, it becomes fairly straightforward to add new behaviour to existing code without directly modifying it. Suppose that it becomes a requirement to serialize objects, i.e., save their data to disk, each time they’re modified. One can leverage the interfaces provided by MOPs to intercept each time object’s slots are changed and thus serialize the object’s state as desired. One could accomplish this task with plain OOP strategies but it



© Marcelo Santos and António Menezes Leitão;
licensed under Creative Commons License CC-BY 4.0

11th Symposium on Languages, Applications and Technologies (SLATE 2022).

Editors: João Cordeiro, Maria João Pereira, Nuno F. Rodrigues, and Sebastião Pais; Article No. 13; pp. 13:1–13:15

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

would require the creation of frontends for slot modification to be able to track changes. This would, in turn, result in the mixture of code relevant to the object's nature and code responsible for serialization, two very distinct concerns. With MOPs, the handling of these concerns would trivially be separated.

A prime example on the implementation of MOPs is the one present in the Common Lisp Object System (CLOS).

1.1 The Julia Programming Language

One of the main problems faced in the implementation of the CLOS metaobject protocol was guaranteeing good performance alongside the flexibility of the MOP [9]. This is a recurring problem in the implementation of higher-level languages, known as the Paradox of High-Level Languages [9]. The main premise of high-level languages is that they allow programmers to better formulate what their programs do through more expressive methodologies. Consequently, compilers for these languages should be able to exploit this knowledge and output faster programs than their low-level counterparts. But reality lies on the opposite side, meaning that there are concepts which are, in fact, not being expressed more clearly, instead requiring programmers to provide sufficient detail to enable efficient execution. We can further divide languages in two categories: static and dynamic languages. Dynamic languages are seen as higher level than static languages, as they allow us to express ideas harder to write in static languages. In the most recent years we've seen a fast growth in popularity of languages featuring object-oriented and dynamic properties [11, 5]. This is due to the increased productivity felt by developers. Python, one of these languages, is used extensively in the fields of numerical and scientific computing, which often require large-scale computations to be executed. One of the major complaints regarding Python is that it is slow [12], so, to resolve that problem, a few strategies are applied:

- Create libraries relying on faster languages to process more intensive operations (e.g. Numpy, a numerical computing library for python which relies on calling C code);
- Optimize the underlying compiler/interpreter (e.g. Pypy, an alternative python implementation featuring Just-in-Time (JIT) compilation);
- Prototyping in this high-level language and then translating the code to a more performant and often lower level language like C, C++, or Fortran. This last route is the one yielding the best results performance-wise, but it falls short for relying on a dichotomy of languages, requiring more knowledge and work from the programmer.

The Julia programming language [2], a dynamic language with a focus on performance, tries to solve this problem. By employing strategies like JIT compilation and code specialization on run-time types, it achieves outstanding results compared to other dynamic languages like Python, while being close to very performant languages like C++ [1].

Multiple dispatch is used extensively in Julia. It is the dynamic dispatch of methods based on the run-time information of all the arguments. Multiple dispatch is applied when calling generic functions to choose which method to dispatch. "A generic function is a function whose behavior depends on the classes or identities of the arguments supplied to it. The methods define the class-specific behavior and operations of the generic function"[3]. This allows for extension of the language by creating more methods for a generic function.

Julia features a very basic object system. Subtyping can only be accomplished from abstract types, the equivalent of abstract classes in Java. So, inheritance must be planned ahead, if one were intending to inheriting behaviour from a struct (concrete class).

1.2 Objectives

Throughout computer language history we have seen a demand for the development of Object Systems atop existing languages. The Lisp community saw the development of CommonLoops [4], which joined Lisp’s procedure-oriented paradigm with object-oriented programming. The same evolutionary strategy was applied to Objective-C, which stemmed from the C language and became a prime example of an OOP language by just adding a small number of syntactic features taken from the Smalltalk language while keeping compatibility with the remaining aspects of C [7].

In this work, we focus on the Julia Programming Language. Currently, the Julia community uses composition as a mechanism to express the sharing of functionality between concrete types. We will design and implement, in Julia, a performant MOP alongside a more expressive Object System.

2 Related Work

The Common Lisp Object System is an object-oriented extension to the Common Lisp[13] language.

CLOS decouples methods from objects through the usage of generic functions. Generic functions are functions whose behaviour depends on the types of the arguments supplied to them. A method defines the behaviour of a generic function for a set of arguments. This object system allows the extension of methods through the definition of method combinations. A method combination results in the application of auxiliary methods triggered by the calling of primary methods, the methods to which auxiliary methods connect to.

CLOS also allows the implementation of mixins, which, like inheritance, allow the sharing of functionality, but without the relationship of subclass. This pattern avoids ambiguity issues related to multiple-inheritance.

CLOS provides a way of changing objects structure through the redefinition of classes at run-time. When a class is redefined, changes are propagated to all instances and the instances of its subclasses[3].

2.1 Metaobject Protocol Implementations

Although many languages leverage MOPs to extend their functionality, we will only focus on the most expressive one, the CLOS MOP, and another one which stands out for its different implementation.

2.1.1 CLOS

The motivation for the development of the CLOS MOP came from the need to give developers the ability to modify the language from a high-level perspective, i.e., without low-level knowledge of the inner workings of the language. The CLOS language provides ways to incrementally change the behaviour of fundamental language constructs. This is possible through the reification of elements such as **class**, **generic-function**, and **method**. These elements are said to be metaobjects and their exposure and consequent modification is what allows for the manifestation of changes to the language semantics. As is often the case, applying global changes to an already existing system without proper testing can lead to unforeseen results. Furthermore, a programmer’s intent might be to just change a portion of the language for a specific section of the program and not its entirety. This problem is

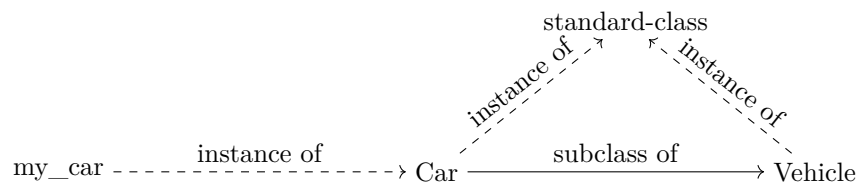
13:4 Metaobject Protocols for Julia

solved by CLOS by exposing metaobjects as part of an object-oriented hierarchy capable of being extended. As such, one can look at the semantics of the language as an object-oriented program as well.

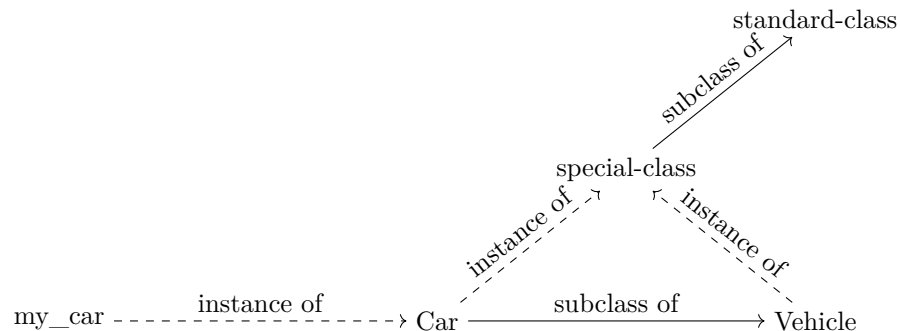
In a traditional class-based object-oriented language, objects, e.g. a car, and classes, e.g., Car and Vehicle, are related through the instance-of and subclass-of relations, as follows:



The implementation of a MOP integrates metaobjects in the system through the reification of classes. Classes are now themselves objects, instances of another class:



The class **standard-class** is said to be a **class metaobject class**, or a **metaclass**. It defines the behaviour of its instances, class metaobjects (**Car** and **Vehicle** in this example), which in turn define the behaviour of regular objects (**my_car** in this example). We can define a subclass of **standard-class** to specify new semantics for the classes **Car** and **Vehicle**, rendering the relationship between the objects as such:



Because **standard-class** still exists and is left unmodified, all other class metaobjects will exhibit the same behaviour as before the creation of **special-class**. Furthermore, only classes explicitly told to have **special-class** as a class metaobject class will follow its defined rules. This MOP allows one to only change a subset of class behaviour, as the remaining functionality will be provided by parent metaclasses, i.e., **standard-class** will be responsible for handling functionality not overridden by **special-class**, as is usual in the inheritance relation of object systems. The selective nature of metaclass specialization is what allows for the incremental extension of programming languages.

Listing 1 presents a concrete CLOS example of metaclass specialization from [9]: We explain how the MOP was used to achieve a modification on the semantics of object instantiation:

- We begin by creating a new metaclass **counted-class** whose main goal is to give classes the ability to track how many times they have been instantiated. It is a subclass of **standard-class**, the default metaclass. We give it a slot, **counter** initialized to zero to track this number, which will be present in all classes that are instances of counted-class.

■ **Listing 1** CLOS Metaclass Example.

```
* (defclass counted-class (standard-class)
  ((counter :initform 0)))

* (defclass counted-rectangle () ()
  (:metaclass counted-class))

* (defmethod make-instance :after
  ((class counted-class) &key)
  (incf (slot-value class 'counter)))

* (slot-value (find-class 'counted-rectangle) 'counter)
0
* (make-instance 'counted-rectangle)
#<COUNTED-RECTANGLE {10042DAC03}>
* (slot-value (find-class 'counted-rectangle) 'counter)
1
```

- Then, we define a class, **counted-rectangle**. This macro also defines it as a metaobject and sets **counted-class** as its metaobject class. The relationship between the classes is as follows:

counted-rectangle $\overset{\text{instance of}}{\dashrightarrow}$ counted-class $\overset{\text{subclass of}}{\longrightarrow}$ standard-class

- We define a method combination for the **make-instance** generic function. Method combinations are a mechanism through which one appends functionality to existing generic functions. In this case, we are adding instructions to be executed after the default implementation runs. This version of **make-instance** is specialized for arguments which are instances of **counted-class**. This method specifies what happens when a new instance is created. In this example, we are incrementing the **counter** slot. This step illustrates the essence of incremental modification - create methods that specialize for the desired metaclasses.
- The remaining instructions demonstrate how the **counter** from the **counted-rectangle** class has been updated after creating an instance of **counted-rectangle**.

The CLOS MOP exposes a reflection interface that enables the program to understand itself while it is running. In the example above, we used **find-class** to retrieve a class metaobject from its name. This mechanism is essential to achieve the reification of language constructs, which is needed for the modification of the language. Besides instantiation, the protocol allows for the inspection and control of the following concepts:

- **Class precedence lists** - the modification of these structures permits the reordering of the priority of superclasses. Given that some languages possessing multiple-inheritance exhibit different ordering schemes, this allows, for example, to port libraries from languages similar to CLOS while maintaining inheritance priority compatibility.
- **Slot Access** - this defines what actions are taken when attempting to access an object's slots.
- **Instance Allocation** - how instances are allocated. The motivation example given in Section 1 would have to resort to this interface in order to accomplish its goal.

13:6 Metaobject Protocols for Julia

■ **Listing 2** Julia Generic Functions and Methods.

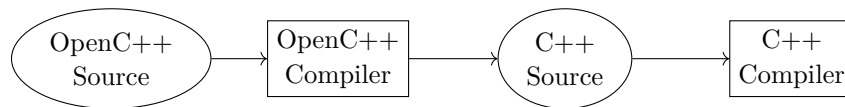
```
f(x::Integer) = x + 1
f(x::Real) = x + 2
```

The reference [9] provides an extensive list of capabilities and methods of the CLOS MOP.

2.1.2 OpenC++

One of the main problems of the CLOS MOP is the performance hit taken. The cause for this problem is the existence of metaobjects at run-time, which are needed to change the semantics of the language. Although this allows for greater flexibility, it increases the level of additional run-time logic.

OpenC++[6] is a metaobject protocol extension to C++[14] similar to CLOS but with a very different approach. It moves all the logic of the MOP to compile time. The purpose of this move is to incur zero run-time speed or space overhead, while still allowing the compiler to perform optimizations. The basic system architecture is as follows: The OpenC++



compiler generates the metaobjects responsible for executing the protocol when compiling OpenC++ to C++. These metaobjects intercept the parsing of regular C++ entities (e.g., class definitions, member access, object creation, virtual function invocation) and take control of their compilation. In essence, these metaobjects modify the program's semantics through the manipulation of parse trees. The resulting modifications are then passed as a regular C++ source to a C++ compiler which doesn't have knowledge of the MOP. With this strategy, no metaobjects exist at run-time, thus saving time and memory. One big disadvantage of this approach is the fact that since the protocol only acts at compile time, it becomes impossible to apply the same changes at run-time. Another minor problem is the increase of time and complexity of compilation.

This solution, which tries to solve the performance issues from metaobject protocols, seems to hint the existence of a tradeoff between execution speed and flexibility, similar to the aforementioned Paradox of High-Level Languages.

2.2 Julia

In this section, we present features of Julia that we consider relevant for our proposal. These include patterns which allow us to build an object system and performance optimizations capable of minimizing the overhead of our implementation.

Julia provides the concepts of generic functions and methods just like in CLOS[3]. Generic functions are implicitly declared through the definition of methods.

We now concern ourselves with method specialization. It is an aggressive mechanism employed by the Julia run-time which, combined with JIT compilation, generates compiled code and caches functions based on their arguments. In Listing 2, we create a generic function **f** with two method implementations: one for **Real** and its subtypes and another for **Integer** and its subtypes. In Listing 3 we perform the respective method calls.

■ **Listing 3** REPL evaluation of generic function `f`.

```
julia> f(1)
2
julia> f(1.0)
3.0
```

■ **Listing 4** Inspection of methods' IR.

```
julia> @code_llvm f(1)
define i64 @julia_f_163(i64 signext %0) {
    %1 = add i64 %0, 1
    ret i64 %1
}

julia> @code_llvm f(1.0)
define double @julia_f_182(double %0) {
    %1 = fadd double %0, 2.000000e+00
    ret double %1
}
```

The Julia programming environment allows us to inspect the Intermediate Representation (IR) of language constructs. IR is a higher level representation of assembly used by LLVM[10], the section of Julia's implementation responsible for generating architecture-specific machine code and applying lower level optimizations. Listing 4 shows that although methods have the same name, they exist in different sections of memory and possess distinct behaviour, as one can see through the IR output by the `@code_llvm` macro.

As a dynamically typed language, Julia allows the programmer to omit types of parameters in function definitions. We can take the example in Listing 4 and generalize it in Listing 5. Intuitively, this method would result in the generation of generic code capable of receiving arguments of any type. By handling a broader set of types, we lose the opportunity to exploit type-specific information that would allow the compiler to optimize this method. Luckily, the approach taken by the developers of the language is far different from this.

We can observe in Listing 6 that each method call executes different code. What this shows is that Julia implicitly employs a JIT strategy to compile at run-time type specific versions of a method according to its arguments. Consequently, exploiting type information becomes possible.

2.2.1 Method redefinition

Immutability, of which Julia takes advantage of, comes as a decisive factor when it comes to compiler optimization. The more structures that are guaranteed to stay constant, the more

■ **Listing 5** Dynamic Typing.

```
g(x) = x + 1

g(1)      # 2
g(1.0)    # 2.0
```

13:8 Metaobject Protocols for Julia

■ **Listing 6** Specialization of generic function.

```
julia> @code_llvm g(1)
define i64 @julia_g_167(i64 signext %0) {
    %1 = add i64 %0, 1
    ret i64 %1
}

julia> @code_llvm g(1.0)
define double @julia_g_186(double %0) {
    %1 = fadd double %0, 1.000000e+00
    ret double %1
}
```

optimizations the compiler can perform. A function accessing variables from the outside will have completely different assemblies depending on the mutability of those variables. If we can assert the immutability of our system, the faster it will run. However, given the nature of run-time metaobject systems, the opposite happens. The ability of changing program structure at run-time is one of the main benefits of using MOPs. Luckily, we can exploit another feature of Julia: function redefinition.

The redefinition of a method changes its behaviour and triggers the recompilation of methods which depend on it. Redefinition allows for behaviour changes without relying on data structures to hold mutable data. This also provides fast access to data without compromising flexibility.

Julia is thus inserted in the group of languages allowing the modification of a program with zero downtime, which opposes the traditional model of shutting down, recompiling, and rerunning programs.

2.3 Problem

This section has shown two Metaobject Protocol systems and exposed a dilemma between them: either opt for a flexible system or a performant one. The nature of these two implementations focuses on the time-frame in which metaobjects exist: run-time or compile-time. Our goal is to bridge the gap between these two architectures and create a performant run-time metaobject protocol on top of the Julia language and its optimizations.

3 Solution

In this section we will describe our implementation proposal for multiple-inheritance method dispatch and method combination mechanisms with zero run-time overhead for the Julia language, without changing its implementation or semantics.

3.1 Class Definition

We begin by describing our class definition mechanism. One can define a new class by calling the `@defclass` macro. It takes the name of the class to be defined and a list of superclasses from which it inherits behaviour. In Listing 7, we define four classes: **A**, which has no superclasses, **B** and **C** which both inherit from **A**, and **D** which inherits from **B** and **C**. This macro is responsible for ensuring the following:

■ **Listing 7** Class definition.

```
@defclass A ()
@defclass B (A,)
@defclass C (A,)
@defclass D (B, C)
```

■ **Listing 8** Defining and calling methods (following Listing 7).

```
> @defmethod bar(a::A) = 0
> bar(D())
0
> @defmethod bar(b::B) = 1
> @defmethod bar(c::C) = 2
> bar(D())
1
```

- Create a Julia structure with the name of the class. This structure has no supertypes, even if it is defined with superclasses.
- Define the method **superclasses** which takes the Julia type for the class and returns the types corresponding to the superclasses. If the class is defined without superclasses, its only superclass is **Any**, the type in Julia of which every type is a subtype of.
- Define the method **prelist** which takes the Julia type for the class and returns the precedence list for the class. A precedence list is an ordered list of classes which determines which are more specific than the others. This is necessary to determine from what classes objects inherit behaviour. By default this order is determined by applying a topological sort to the class hierarchy, similar to CLOS.
- Define the method **classof**, which when receiving an instance of the class, returns the class itself. This method is analogous to Julia's **typeof**.

These mechanisms for retrieving class information rely on methods and not mutable data structures or constants because:

- Although constants allow for the optimization of generated code, they forbid future changes, which goes against the purpose of metaobject protocols.
- Even though mutable data structures would allow for changes in the class system, they prevent the same optimizations done to constants.

Methods whose only purpose is to return data give us the best of both worlds: mutability and optimizations. The only cost for mutability is recompilation time.

3.2 Method Dispatch

We now describe how we use Julia's language feature to build a multiple-inheritance method dispatch mechanism. We provide the **@defmethod** macro, that, just like regular Julia method definitions, takes a method name, the list of arguments and optionally their types, and the method body, as we can see in Listing 8.

This macro is responsible for:

- Store an anonymous function taking generic arguments with the same method body as the one passed to **@defmethod** macro. This is stored in a key-value manner, in which the key is the list of the types of the parameters and the value is the anonymous function.

13:10 Metaobject Protocols for Julia

■ **Listing 9** Method combination.

```
> @defmethod foo(d::D) = println("Primary method")
> foo(D())
Primary method
> @defmethod foo::before(d::D) = println("Calling before")
> @defmethod foo::after(d::D) = println("Calling after")
> foo(D())
Calling before
Primary method
Calling after
```

- Create a julia method, the method computer, taking generic arguments with the same number of arguments as the one specified with **@defmethod** whose purpose is to select the appropriate method to apply given the types of the arguments. If the method computer already exists, it is redefined to include the update of the available anonymous functions.

3.2.1 The Method Computer

The method computer is where most of the dispatch work takes place. It executes the following steps:

- Gather a list of the types of the arguments supplied to the method.
- Get the list of methods whose parameters are compatible with the arguments.
- Sort this list of methods in order to obtain the most specific method.
- Call the most specific method by passing it the arguments received and return its return value.

We say that a list of parameters P is compatible with a list of arguments A if:

- The length of P is equal to the length of A .
- For each pair (P_k, A_k) where P_k is the k^{th} parameter and A_k is the k^{th} argument, P_k is in the precedence list of A_k .

A method M is more specific than some method N , with respect to a list of arguments A if M has the smallest k for which M_{P_k} comes before N_{P_k} in the precedence list of A_k , where M_{P_k} is the k^{th} element of the parameters of M , N_{P_k} is the k^{th} element of the parameters of N , and A_k is the k^{th} argument.

3.3 Method Combination

Currently we support the same method combination implemented by the CLOS Standard Method Combination: before, after and around methods. See Listing 9 for an example.

Besides selecting the most specific method, the Method Computer must also take into account method combination. To do so, it must:

- Separate the method lists into four groups: before, after, around and primary. Primary methods are those defined without any method combination specifier.
- Apply the same filtering and ordering from the arguments for each group.
- Generate and call an effective method, which comes from applying all valid methods from the combination.

■ **Listing 10** Joining anonymous functions with `join_lambdas`.

```
function join_lambdas(around::Tuple, b::Tuple, p::Tuple, a::Tuple)
  (x...) -> begin
    next = join_lambdas(around[2:end], b, p, a)
    callnextmethod() = next(x...)
    callnextmethod(y...) = next(y...)
    hasnextmethod() = true
    around[1](x..., callnextmethod, hasnextmethod)
  end
end
```

3.3.1 Computing the Effective Method

The effective method is an anonymous function returned by the recursive method `join_lambdas`. `join_lambdas` receives four arguments, one for each method group. Each call to `join_lambdas` attaches one method from one group, while processing one group at a time. The order for processing groups is the same as the one specified by CLOS' method combination semantics: `around`, `before`, `primary`, and `after`.

The method `join_lambdas` has a different way of joining methods depending on which group it is currently processing:

- While processing a method from the `around` group, return an anonymous function which calls the method being processed. Besides receiving the arguments passed to the effective method, the method being processed also receives two functions as arguments: `nextmethod` and `hasnextmethod`. `nextmethod` is the next recursive call to `join_lambdas`. `hasnextmethod` returns true if there are more methods following in the combination and false otherwise. The `join_lambdas` method for processing the `around` group is shown in Listing 10.
- Processing the `before` group is much simpler. The only concern of the returned anonymous function is calling the current method and recurring into the `join_lambdas` call.
- Handling the `primary` group is very similar to the `around` group, given that it must allow calling `hasnextmethod` and `nextmethod`. The biggest difference from the `around` group processing is storing the return value from the call and returning it only after the `after` group.
- Processing the `after` group is identical to what is done to the `before` group.

We follow these steps instead of an iterative method calling approach because the JIT can better optimize method call chains. The resulting effective method of `foo` in Listing 9 can be seen in Listing 11.

3.4 Integrating Julia Types

Since the classes created by our system are Julia types, the preexisting Julia types are automatically partially supported. To fully integrate them, we need to define for the regular types the same methods we define for classes and their instances:

- `superclasses` is equivalent to calling the built-in function `supertype` and returning a single element list;
- `preclist`, the precedence list of a type, is equivalent to returning a list of every supertype until the `Any` type is reached;
- `classof` is the same as calling the built-in `typeof`.

With all of these equivalences in place, we can create methods through our `@defmethod` macro with regular Julia types.

13:12 Metaobject Protocols for Julia

■ **Listing 11** Resulting effective method of `foo` from Listing 9 (simplified).

```
(x1...) -> begin
    ((d) -> println("Calling before"))(x1...)

    ((x2...) -> begin
        res = ((d) -> println("Primary method"))(x2...)

        ((x3...) -> begin
            ((d) -> println("Calling after"))(x3...)
        end)(x2...)

        return res
    end)(x1...)
end
```

■ **Listing 12** IR code from calling `bar(D())`.

```
define i64 @julia_foo_1266() #0 {
top:
    ret i64 1
}
```

4 Evaluation

We now proceed to analyse the performance of our solution. We will take two approaches. First, to look at the LLVM Intermediate Representation (IR) language, a higher-level assembly, generated by the JIT. Second, to compare execution times with CLOS. The following tests were executed on an Intel Core i5-8250U 3.4GHz PC with 16GiB of RAM running the Linux operating system.

4.1 Generated IR

We can analyse the performance of our method dispatch mechanism by taking the code from Listing 8 and reading its IR in Listing 12. As we can see, no computation from method dispatch is present in the end result. Only the relevant method body remains. We can thus conclude that our multiple-inheritance method dispatch is just as performant as Julia's single-inheritance method dispatch.

The overhead resulting from computing the effective method in Listing 9 can be seen in Listing 13. Just like in the previous example, no overhead is present. Not only that, but there are also no intermediate method calls being made from the chain produced by `join_lambdas`. The bodies from each method were joined into a single body. Only the code relevant for printing is displayed. This makes the use of our method combination mechanism equivalent to creating a regular Julia method with a body as the concatenation of the bodies of the methods defined for the combination. This regular Julia method, defined in Listing 14 yields the same IR as the method combination in Listing 9, shown in Listing 13, thus having the same execution times.

■ **Listing 13** IR code from calling the method combination of `foo(D())` (simplified).

```
define void @julia_foo_1172() #0 {
top:
    %0 = alloca {}, align 8

    store {}* inttoptr (i64 139884251139280 to {}*), {}** %0, align 8
    %1 = call nonnull {}* @j1_println_1178(...)

    store {}* inttoptr (i64 139884243053904 to {}*), {}** %0, align 8
    %2 = call nonnull {}* @j1_println_1179(...)

    store {}* inttoptr (i64 139884260369712 to {}*), {}** %0, align 8
    %3 = call nonnull {}* @j1_println_1180(...)

    ret void
}
```

■ **Listing 14** Regular Julia method equivalent to method combination in Listing 9.

```
function foo_effective(d::D)
    println("Calling before")
    println("Primary method")
    println("Calling after")
end
```

4.2 Execution time against CLOS

The Steel Bank Common Lisp (SBCL) is a high-performance Common Lisp implementation. In this section we will compare an ad-doc example of method dispatch between the SBCL CLOS and our Julia solution. For this example, we iterate a list of objects to force method dispatch. We have the Julia version in Listing 15 and the SBCL CLOS in Listing 16. Both versions assume a class hierarchy like the one specified in Listing 7. Each iteration example was ran once. The results for calling `foo` with `arr` as an argument yields times of **0.409013** seconds for Julia and **0.229447** seconds for CLOS, which means SBCL is more optimized. However, Julia only exhibits a worse time because it cannot infer the types of the elements of `arr`. If we specify `arr` as being an array of `D`, then we get a much better result: **0.000012** seconds.

■ **Listing 15** Julia example.

```
@defmethod baz(b::B, n) = n+1
@defmethod baz(c::C, n) = n*2
arr = []
for i in 1:10000000 push!(arr, D()) end
foo(a) =
    let y = 0
        for e in a
            y += baz(e, 10)
        end
    y
end
```

■ **Listing 16** CLOS example.

```
(defmethod baz ((b B) n) (+ n 1))
(defmethod baz ((c C) n) (* n 2))
(defparameter arr (make-list 10000000
                             :initial-element (make-instance 'D)))
(defun foo (a)
  (let ((y 0))
    (loop for e across a do
      (incf y (baz e 10)))
    y))
```

5 Future Work

The work presented in this paper is an element of a metaobject protocol being implemented in Julia. Our future work is to continue adding features to our implementation in order to get closer to levels of expressiveness of CLOS.

6 Conclusion

In this paper, we discussed metaobject protocols. We analysed two different implementations choosing different trade-offs in terms of performance and expressiveness. Through the described optimizations of the Julia language, we proposed a zero run-time overhead solution to two main pieces of metaobjects protocols: multiple-inheritance method dispatch and method combinations.

References

- 1 S Borağan Aruoba and Jesús Fernández-Villaverde. A comparison of programming languages in macroeconomics. *Journal of Economic Dynamics and Control*, 58:265–273, 2015.
- 2 Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.
- 3 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.
- 4 Daniel G Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. Commonloops: Merging lisp and object-oriented programming. *ACM Sigplan Notices*, 21(11):17–29, 1986.
- 5 Stephen Cass. The 2015 top ten programming languages.
- 6 Shigeru Chiba. A metaobject protocol for C++. In *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 285–299, 1995.
- 7 Brad J Cox. *Object oriented programming: an evolutionary approach*. Addison-Wesley Longman Publishing Co., Inc., 1986.
- 8 Gregor Kiczales, J Michael Ashley, Luis Rodriguez, Amin Vahdat, and Daniel G Bobrow. Metaobject protocols: Why we want them and what else they can do. *Object-Oriented Programming: The CLOS Perspective*, pages 101–118, 1993.
- 9 Gregor Kiczales, Jim Des Rivieres, and Daniel G Bobrow. *The art of the metaobject protocol*. MIT press, 1991.
- 10 Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

- 11 Linda Dailey Paulson. Developers shift to dynamic programming languages. *Computer*, 40(2):12–15, 2007.
- 12 Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: How do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, pages 256–267, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3136014.3136031.
- 13 Guy Steele. *Common LISP: the language*. Elsevier, 1990.
- 14 Bjarne Stroustrup. *The C++ programming language*. Pearson Education India, 2000.