Metaobject protocols: Why we want them and what else they can do

Gregor Kiczales, J.Michael Ashley, Luis Rodriguez, Amin Vahdat, and Daniel G. Bobrow

Published in A. Paepcke, editor, *Object-Oriented Programming: The CLOS Perspective*, pages 101 — 118. The MIT Press, Cambridge, MA, 1993.

Metaobject Protocols
Why We Want Them
and
What Else They Can Do

Appears in
Object Oriented Programming: The CLOS Perspective

Gregor Kiczales, J. Michael Ashley, Luis Rodriguez,
Amin Vahdat and Daniel G. Bobrow

Originally conceived as a neat idea that could help solve problems in the design and implementation of CLOS, the metaobject protocol framework now appears to have applicability to a wide range of problems that come up in high-level languages. This chapter sketches this wider potential, by drawing an analogy to ordinary language design, by presenting some early design principles, and by presenting an overview of three new metaobject protcols we have designed that, respectively, control the semantics of Scheme, the compilation of Scheme, and the static parallelization of Scheme programs.

Introduction

The CLOS Metaobject Protocol (MOP) was motivated by the tension between, what at the time, seemed like two conflicting desires. The first was to have a relatively small but powerful language for doing object-oriented programming in Lisp. The second was to satisfy what seemed to be a large number of user demands, including: compatibility with previous languages, performance comparable to (or better than) previous implementations and extensibility to allow further experimentation with object-oriented concepts (see Chapter 2 for examples of directions in which object-oriented techniques might be pushed). The goal in developing the MOP was to allow a simple language to be extensible enough that all these demands could be met.[1]

Traditionally, languages have been designed to be viewed as black box abstractions; end programmers have no control over the semantics or implementatation of these abstractions. The CLOS

[1]Some might argue that the base CLOS language is not so simple. What is more interesting is to consider how much of what is in that language could be dropped given the current or an improved metaobject protocol, now that we have a better handle on that technology. Among those features that would be prime candidates for elision would be: method combination, the `:argument-precedence-order` option, methods on individuals (`eql` specializers) and class variables. More aggressive proposals might drop aspects of the initialization protocol like slot-filling initargs and default initargs. Even more aggressive proposals might drop multiple inheritance and multi-methods.

MOP on the other hand, "opens up" the CLOS abstraction, and its implementation to the programmer. The programmer can, for example, adjust aspects of the implementation strategy such as instance representation, or aspects of the language semantics such as multiple inheritance behavior. The design of the CLOS MOP is such that this opening up does not expose the programmer to arbitrary details of the implementation, nor does it tie the implementor's hand unecessarily — only the *essential structure* of the implementation is exposed.

In more recent work, we have pushed the metaobject protocol idea in new directions, and we now believe that idea is not limited to its specific incarnation in CLOS, or to object-oriented languages, or to languages with a "large runtime" or even to Lisp-like languages. Instead, we believe that providing an open implementation can be advantageous in a wide range of high-level languages and that metaobject protocol technology is a powerful tool for providing that power to the programmer.

The purpose of this chapter is to summarize what has been learned to date about MOP technology, and suggest directions it might be pursued in the near future. In the first two sections, we start with the CLOS MOP, first summarizing the motivation for it, and then giving a brief overview of how one might think about its development. We then present a general framework for thinking about MOP design, together with some early principles for design cleanliness in a MOP. Finally, we present an overview of three new MOPS we have designed for Scheme, showing that a MOP can be used to handle a range of issues other than those in the CLOS case.

Motivating Examples

In this section, we present two simple problems that arise in languages like CLOS without a MOP, as a way of summarizing the motivation for the CLOS MOP. We also generalize from these examples, as a way of suggesting what other kinds of problems an open language implementation might be used to solve. The first example has to do with performance. In it, the programmer has written a program which CLOS expresses quite well, but finds (perhaps to their surprise) that the underlying language implementation doesn't perform adequately. In the next section we return to this example to show how the CLOS MOP addresses this problem by opening up part of the implementation strategy to the programmer. The second example, is a case where the programmer can be well-served by being able to adjust some part of the language semantics to better suit their needs.

A Performance Problem

Consider the two CLOS class definitions shown in Figure 1. The class `position` might be part of a graphics application, where the instances are used to represent the position of the mouse as it moves. The class defines two slots `x` and `y`. In this case, the behavior of the program is such that there will be a very large number of instances, both slots will be used in every instance and access to those slots should be as fast as possible.

The second definition, `person`, might come from a knowledge representation system, where the instances are being used as frames to represent different individuals. The thousand slots defined in the class correspond to a thousand properties of a person that might be known. In this application, the behavior is such that although there will be a very large number of instances, in any given instance only a few slots will actually be used. Furthermore, access to these properties will rarely be in the inner loop of a computation.

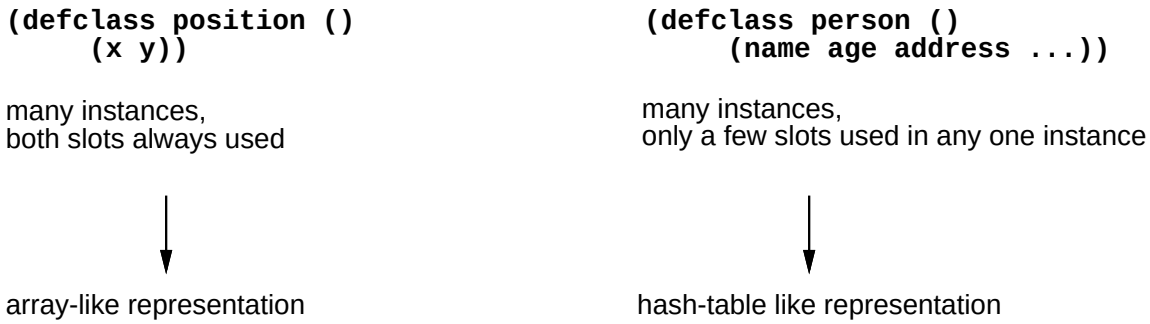

Figure 1: Two sample CLOS class definitions. Ideally, each class would like a different underlying instance implementation strategy. In a traditional (sans MOP) CLOS implementation, one or the other of these classes (probably `person`) will have bad performance.

Clearly, the ideal instance implementation strategy is different for the two classes. For `position`, an array-like strategy would be ideal; it provides compact storage of instances, and rapid access to the `x` and `y` slots. For `person`, a hash-table like strategy would be more appropriate, since it isn't worth allocating space for a slot until it is known that it will be used. This makes access slower, but it is a worthwhile tradeoff given a large number of instances.

The likely default implementation in most object-oriented languages is the array-like strategy. This serves the author of the `position` class quite well, but author of the `person` will not be so happy. Even though the CLOS language abstraction serves to express their program quite clearly, (supposedly) hidden properties of the implementation will impair performance.

This particular problem is not an isolated incident, it is an instance of a common, and much more general problem. As a programming language becomes higher and higher level, its implementation in terms of the underlying machine involves more and more tradeoffs, on the part of the implementor, about what cases to optimize at the expense of what other cases. That is, the fact that a typical object-oriented language implementation will do well for `position` and poorly for `person` is no accident — the designer of the implementation made (what we hope was a concious) decision to tune their implementation this way. What a properly designed open language implementation does is allow the end-programmer to go back, and "re-make" one of those tradeoffs to better suit their needs. A more complete discussion of this kind of performance problem, and the ways in which open language implementations based on MOPs can help address it is given in [1].

The Desire to Customize Behavior

As an example of how a programmer can derive benefit from being able to adjust the language's semantics, consider the case of a programmer who is porting a large body of Flavors [2] or Loops [3] code to CLOS. In most respects, Flavors, Loops and CLOS are sufficiently similar that the task of porting the code is (relatively) straightforward. Each has classes, instances with slots (or instance variables), methods, generic functions (or messages), and all the other basics of object-oriented behavior. It appears that a largely syntactic transformation can be used.

But there is one critical difference between the languages: while they both support multiple

inheritance, they use different rules to order the priority of superclasses. This means that a simple mapping of Flavors programs into CLOS can fail because of asymmetries in the inheritance of methods and slots. Given a traditional CLOS implementation, the programmer would have to either rewrite the code appropriately, or (perish the thought) implement Flavors for themselves from scratch.

This too is an instance of a more general and more common problem. Again, as language become higher and higher level, and their expressive power becomes more and more focused, the ability to cleanly integrate something outside of the language's scope becomes more and more limited. An open language implementation that provides control over the language permits the programmer to shift the language focus somewhat, so that it becomes a better vehicle for expressing what they have in mind.

This example also reflects the ability of MOP-based open language implementations to help keep languges smaller and simpler. That is, by having a metaobject protocol, the base language need not provide direct support for functionality which only some users will want—that can be left to the user to provide for themselves using the metaobject protocol.

Simple Metaobject Protocols

The first example above was brought to our attention by users who complained about the performance of their KR programs. To fix this, we needed to find a way to give them control over the part of the implementation that decides the instance representation strategy. To make that strategy replaceable, we need to put all the parts of the runtime that are based on it under the control of generic functions: the code that allocates instances—which needs to know how much space to allocate; and the code that accesses slots—which needs to know where to go to get or put a slot value.

Three generic functions in the protocol suffice: `allocate-instance`, `get-value` and `set-value`.[2] We require that the runtime, whenever it needs to create an instance or access a slot, do so by calling these generic functions. The simple metaobject protocol now looks like:

Run Time Action	Implemented By Call To
instance allocation	(`allocate-instance` *class*)
slot read or write	(`get-value` *class instance slot-name*)
	(`set-value` *class instance slot-name new-value*)

Note that these generic functions all receive the metaobject—the class in question—as their first argument. This is the case even for `get-value` and `set-value`, which could conceivably determine it from the object. They must receive it as an argument to make it possible to define methods specialized to the class metaobject class.

Given this protocol, the user can write meta-code—an extension to the CLOS implementation that provides a new kind of class—to support instances with a hash-table representation. As shown

[2] In this paper, where we are trying to briefly summarize the MOP approach, we have allowed ourselves some leeway from the details of the real CLOS MOP. To avoid confusion, we have used different names for protocol presented here which differs from the real MOP.

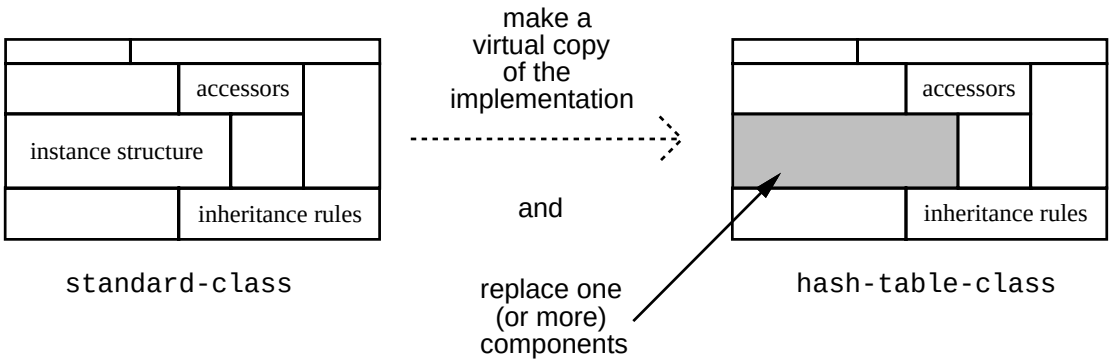

Figure 2: The standard implementation of classes has an internal structure, defined by the metaobject protocol, consisting of a number of components, including the instance representation strategy. Defining a new kind of class (i.e. `hash-table-class`) is a two-step operation: (i) Defining the subclass is like making a virtual copy of the standard implementation. (ii) Defining specialized methods (i.e. on `allocate-instance`, `get-value`, and `set-value`) is like replacing an internal component in that copy.

in Figure 2, the use of object-oriented techniques in the MOP makes this convenient. The code looks something like:

```
(defclass hash-table-class (standard-class) ())

(defmethod allocate-instance ((c hash-table-class))
  ...allocate a small hash table to store the slots...)

(defmethod get-value ((c hash-table-class) instance slot-name)
  ...get the slot value out of the hash table...)

(defmethod set-value ((c hash-table-class) instance slot-name new-value)
  ...store the slot value in the hash table...)
```

Then, in their base program, programmers can request that the metaobject for specific classes they define be instances of `hash-table-class` rather than `standard-class`. This is done by marking the definition of those classes using the `:metaclass` option.

```
(defclass person ()
     (name age address...)
  (:metaclass hash-table-class))
```

Recovering Performance

While the approach outlined above does provide the programmer a great deal of power, the fact that it introduces meta-level generic function calls into basic operations of the language runtime means it incurs a significant performance penalty. In the CLOS case, it is clear that we can't require the implementation of slot access, which should inherently be a two memory read operation, to require a generic function call at the meta-level.[3] Our current approach to this problem is to incrementally redesign the protocol so that we "pull the meta-level generic function dispatches out of the critical runtime paths," although we hope to explore the use of automatic techniques, such as partial evaluation, to help with this kind of problem.

Again turning to the CLOS MOP, the protocol for generic function invocation and method dispatch provides an excellent example of how this can be done. According to the semantics of CLOS, when a generic function is called, the following actions are performed: (i) The class of each of the required arguments is determined. (ii) Using those classes, the sorted set of applicable methods is determined. (iii) The applicable methods are combined into an effective method. (iv) That effective method is run, receiving as arguments all the original arguments to the generic function.

Under a simple protocol in the previous style, we might introduce generic functions for the second and third of these steps. The nature of this protocol can be seen in the following piece of code, which would be the runtime code required to implement generic invocation under this protocol. Note that, for simplicity, we are ignoring optional, keyword and rest arguments, and also ignoring all error checking. (Uppercase is used to mark calls to generic functions in the metaobject protocol.)

```
(defun apply-generic-function (gf args)
  (let* ((classes (mapcar #'class-of args))
         (methods (COMPUTE-METHODS gf classes))
         (effective (COMPUTE-EFFECTIVE-METHOD gf methods))
         (compiled (compile-effective-method effective)))
    (call-effective-method compiled args)))
```

Again, as with slot access, this protocol is powerful, but the requirement that meta-level generic functions be called as part of the invocation of each base level generic function means that performance will be unacceptable.

The optimization to the protocol is based on two observations: First, by far and away most of the work is in the generic functions `compute-methods` and `compute-effective-method`. Second, that by placing only modest restrictions on these generic functions, we can allow the implementation to cache their results. The restriction is simply that any method on these generic functions be *functional* in nature; that is, that given the same arguments it must return the same results. (The real MOP uses a somewhat more elaborate set of rules, but the basic principle is the same.)

The essence of the resulting protocol, together with its implementation, can be seen in the following pseudo-code for the part of the runtime that applies a generic function to arguments. This code shows that in the common path through the runtime—the cache hit—there are no meta-level generic function calls. This means that in steady state, no performance penalty is incurred for

[3] Because of the particular rules for slot inheritance in CLOS, a typical implementation, which supports incremental development, cannot get down to just a single memory read. See Chapters 13 and 14 for a discussion of these implementation issues.

having the MOP [4]. Nevertheless, the protocol does allow the programmer to customize method lookup and combination rules, and thereby achieve a wide variety of alternate language behavior.

```
(defun apply-generic-function (gf args)
  (let* ((classes (mapcar #'class-of args))
         (cached (cache-lookup gf (class-of first))))
    (if cached
        (call-effective-method cached args)
        (let* ((methods (COMPUTE-METHODS gf classes))
               (effective (COMPUTE-EFFECTIVE-METHOD gf methods))
               (compiled (compile-effective-method effective)))
          (fill-cache gf classes compiled)
          (call-effective-method compiled args)))))
```

Optimization of procedural protocols, such as `get-value` and `set-value` is more complex but is based on the same general approach. As with method dispatch, some of the protocol is "pulled back" to earlier times in the run-time image, like cache-filling time. Other aspects of the protocol are actually run as part of compiling method bodies. This is covered in greater detail in [5]. Pulling the protocol back as far as compile-time also shows up in the Scheme compiler MOPs discussed later.

Methodology

One way of thinking about MOP design is by analogy to programming language design. As shown in the left half of Figure 3, a good language designer works by first having in mind — or, better yet, "down on paper" — examples of the kinds of programs they want their users to be able to write. The language then follows from those examples as the designer works to be able to express them cleanly. Of course the process is far more iterative and ad-hoc, but the basic point is that language designers are, in effect, working with two different levels of design process at the same time: the level of designing particular programs in terms of a given language, and the level of designing the language to support the lower-level design processes.

MOP design is similar, with the addition of yet one more level of design process. Again, the fundamental driving force is a sense of the kinds of programs users should be able to write. But, in this case the designer is not thinking about a single language that might be used to express those programs, but rather a whole range of languages. So, in addition to the two previous levels — the programmer designing a program in terms of a single language and the language designer designing a single language to support that lower level — there is a third level, designing a MOP to support a range of language designers. This is depicted in the right half of Figure 3.

In this kind of design process, one important observation is that user complaints about previous languages and implementations take on tremendous value. Read carefully, they can provide important clues as to what flexibility programmers might want from the MOP. It is also important to note that, like language design, MOP design is inherently iterative; it is difficult to guess just what the users will want the first time out. This is certainly true of the CLOS MOP. The use of object-oriented techniques to organize a meta-level architecture first appeared in [6] and [7]. The former was a suggestive implementation for a MOP for Common Lisp. It took a prototype

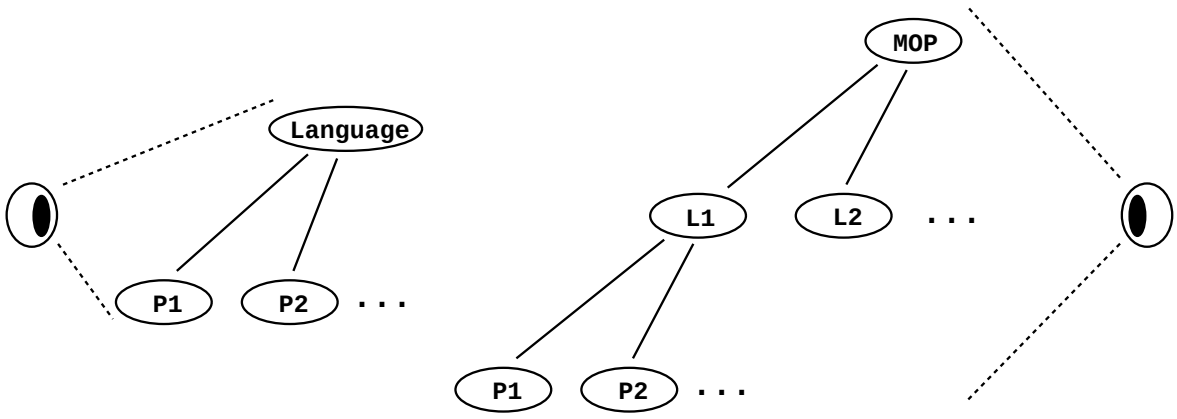

Figure 3: Language design and MOP design. On the left, the language designer envisages a range of programs the language should support elegantly, and designs a language accordingly. On the right, the MOP designer thinks about an even wider range of programs, and a *range* of languages, and designs the MOP accordingly.

implementation [4], and the feedback from a large community over five years to refine it. Moreover, the documentation of this kind of protocol brings up a number of subtle and unsolved issues in object-oriented specification [8].

In this sort of design process, the designer makes recourse to inuitive, aesthetic principles like "elegance" and "concision." That is, when desiging a programming language, one wants to make the resulting programs "simple" and "elegant." When designing a MOP, one wants to make not only the base programs simple and elegant, but also the meta-programs that define language extensions simple and elegant.[4]

0.1 Locality in MOP Design

This notion of elegance is inherently informal, and there is a great deal of work to be done before it can be reduced to standard practice, but there are some early design principles we have found it productive to think in terms of. There are based on a common aesthetic principle in computer science, the notion of localization of concerns and effect. Following are five coarse notions of locality in metaobject protocols. These are neither sharp nor orthogonal, but they do serve to talk about intuitive notions that come up in MOP design.

- *Feature Locality* – The metaobject protocol should provide access to individual features of the base language. So, in CLOS for example, a programmer should be able to customize slot inheritance without having to take over all aspects of inheritance. In a MOP for Scheme, the programmer should be able to get at binding discipline without dealing with order of evaluation.

[4]It is arguably more important for the base programs to be elegant than the meta programs, since there are far more of them, and we anticipate they will be written by less sophisticated programmers.

- *Textual Locality* – The programmer should be able to indicate, using convenient reference to their base program, what behavior they would like to be different. So for example, if the programmer is changing slot inheritance, they should be able to conveniently mark what classes in their program use the new inheritance.

- *Object Locality* – The programmer should be able to affect the implementation on a per-object basis. So, for example, in a MOP for Scheme, the programmer should be able to refer to individual closures, or all the closures resulting from a particular `lambda`.

- *Strategy Locality* – The programmer should be able to affect individual parts of the implementation strategy. So, for example, it should be possible to affect instance representation strategy without affecting method dispatch strategy.

- *Implementation Locality* – Extension of an implemenation ought to take code proportional to the size of the change being contemplated. A simple customization ought to be an *incremental* change. Programmers don't want to have to take total responsibility for the entire implementation; they don't want to write a whole new implementation from scratch. A reasonably good default implementation must be provided, and the programmer should be able to describe their extension as an incremental deviation from that default.

Applicability to Other Languages

While the metaobject protocol mechanism depends on having an object-oriented meta-language, there is no requirement that the base language be object-oriented. In this section we discuss the application of the approach to Scheme [9], using three metaobject protocols we have developed at PARC. There are two important structural differences between these MOPs and the CLOS MOP: First, the base language (Scheme) is not object-oriented, so the MOP is controlling issues of Lisp implementation and semantics, such as function calling convention and variable binding semantics, that are outside the purview of the CLOS MOP. Second, the meta-language used to implement the MOP is CLOS, not Scheme or an object-oriented extension to Scheme. That is, these systems are not meta-circular and they do not have towers.

Ploy

The first of these protocols, called Ploy, is a simple interpreter for Scheme. As an interpreter, Ploy's main use is as a testbed for working with programmer extensions of base language semantics. We are, in essence, using it to drive the iterative process of discovering *what* aspects of Scheme semantics to put under control of the protocol, before we attempt to develop a fully-featured, high-performance protocol.

In the Ploy MOP there are three distinct categories of metaobject: (i) Those that represent the program being interpreted — nodes in the program graph. (ii) Those that represent internal runtime data structures of the interpreter — environments and bindings. (iii) Those that represent Scheme values that are visible in the base level program — pairs, numbers, procedures and the like. The default metaobject class graph provided by Ploy is shown in Figure 4.

The interpreter works in the natural way: a documented set of generic functions evaluates the program, building environment structure as it goes. The most general of these generic functions

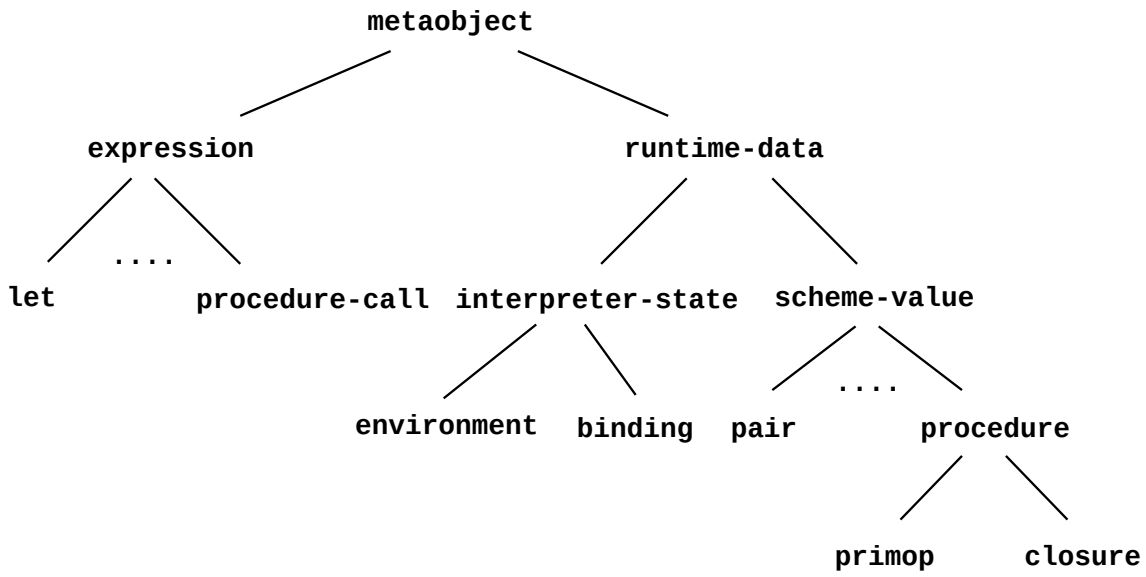

Figure 4: An overview of the default classes in Ploy, showing the three basic categories of metaobject: `program-element`, `interpreter-state`, and `scheme-value`.

is `eval`. In addition, to make user extension simpler, the protocol is also *layered*. That is, there are subsidiary generic functions that have more specific purposes and are thereby easier to define methods for. For example, there are special generic functions that manipulate binding metaobjects: creating them, and reading and writing their values. Figure 5 illustrates the layers in the Ploy protocol.

The current protocol is small, with only 9 generic functions. But, using it, we have been able to implement a number of traditional extensions to Scheme semantics, including: normal order evaluation, partial closures [10], control over order of evaluation, values that trace their path through the program, monitored variables and fully dynamic variables. The implementation of each of these extensions is simple and small, ranging from 8 to no more than 30 lines of code. The protocol localizes effect well enough that all these extensions can be loaded at the same time, and they can all be used in the same program.

In this paper, we demonstrate the use of the Ploy MOP using a toy example, monitored variables. More elaborate examples, including the implementation of partial closures and normal order evaluation, can be found in [11].

A monitored variable binding is one that prints out a message whenever its value is read or set. The Ploy MOP supports this kind of extension by requiring that access to the value of a binding go through the documented generic functions `read-value` and `write-value`. This protocol makes the implementation of monitored bindings quite straightforward. The core of the code is the definition of a new class of binding metaobject and appropriate methods on the reading and writing generic functions. (Note that because the meta-level code is in CLOS, not Scheme, it is easier to distinguish it from base-level code. This is one of the reasons we have chosen this approach.)

```
(defclass monitored-binding (binding) ())
```

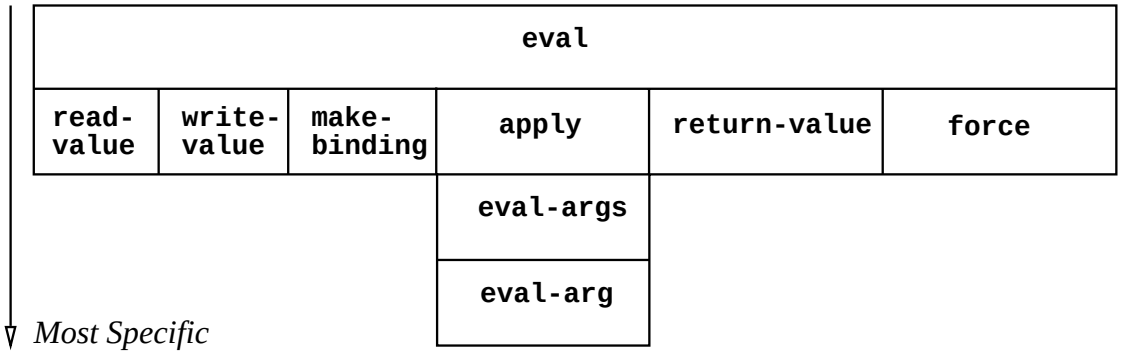

Figure 5: The calling structure of the Ploy protocol, indicating its layering.

```
(defmethod read-value :before ((b monitored-binding))
  (format t "Reading the value of ~S." b))

(defmethod write-value :before ((b monitored-binding) new)
  (format t "Setting the value of ~S to ~S." b new))
```

Now we must arrange for all the binding metaobjects corresponding to a particular variable to be of this new class. The Ploy MOP allows the user to control, using special syntax in the base-level program, the class of program element metaobjects; we must therefore arrange for a special class of variable program element metaobject to produce the special class of binding metaobject. To support this "chaining" from program element metaobject to interpreter state metaobject, the protocol provides the `binding-class` generic function. (This kind of chaining is common in our MOPs for Scheme, and more will be said about it in the next section.) The rest of the code for the extension looks like:

```
(defclass monitored-variable (variable) ())

(defmethod binding-class ((f monitored-variable))
  (find-class 'monitored-binding))
```

Then, using a special syntax with curly braces, the programmer requests that the program element metaobject for `x` be of class `monitored-variable` rather than the default `variable`.

```
(let (({monitored-variable}x 1)
      (y 2))
```

within the body of the let, access
to x prints out a message, but
access to y, or any variables free
in the let, are unaffected

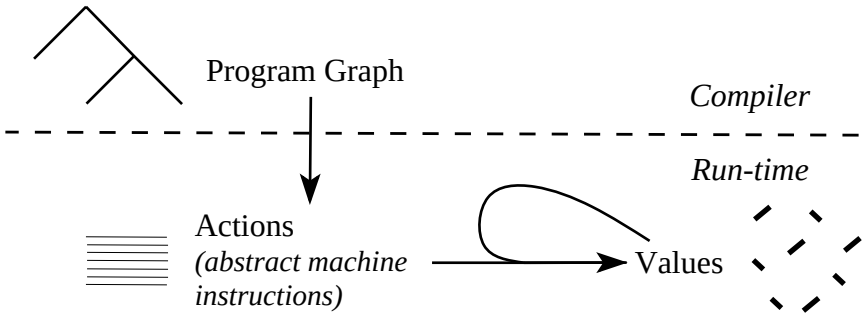

Figure 6: The Sartor architecture. The compiler protocol translates the source program graph into actions, or abstract machine instructions. The runtime protocol executes those actions, producing and operating on runtime data objects.

)

Sartor

Sartor is a compiler that compiles Scheme programs to abstract object code for a virtual machine. The architecture is illustrated in Figure 6. As indicated in the figure, an internal representation of the source program, a *program graph*, is compiled to a program in a high-level target language of *actions*. When this program is run, it creates and manipulates a variety of run-time data including both Scheme values such as pairs and closures, and internal values such as activation records.

Sartor is intended to be a testbed for experimenting with the use of MOP techniques to allow a user to control compilation strategies. As with Ploy, our initial goal has been to study *what* aspects of a compiler's behavior it makes sense to expose, rather than to handle actual code generation. In this sense, Sartor resembles the initial slot access protocol presented above. It introduces meta-level control at the cost of "runtime" method dispacth overhead. Once we have a better sense of what to protocolize, we will focus on restructuring the architecture to generate high-performance code.

Sartor in fact has two metaobject protocols: The compiler MOP operates on program graph metaobjects, producing action metaobjects. The compiler MOP makes decisions about order of evaluation and how the environment structure will be organized. This allows programmer extensions to control such issues as whether a free variable should be stored in a closure, or passed in by all callers to the closure.

The runtime MOP operates on actions, internal runtime data structures, and Scheme values. It controls the representation of runtime structures data, including such issues as how an activation record should be implemented, or how a closure's environment "tuple" should be represented in memory.

As an example of the protocol and how it might be customized, consider the implementation of activation records. The sequence of actions for performing a procedure call invokes the `construct-activation-record` generic function on the closure being called. By default, the record

constructed allocates storage for itself in the heap. But, the programmer can define a new class of activation record, that allocates itself on the stack. Or perhaps one that stores some of the arguments in registers.

The "metaobject chaining" issue mentioned in the previous section can be seen here quite clearly. In this case, the programmer is going to want a special kind of activation record to be created at runtime, but in order to do this, they are going to want to mark a particular `lambda` node (or perhaps a call site) in the source program. From that `lambda` node, there is a chain to the action that creates the closure at runtime, to the closure itself, and finally to the activation record. This kind of chaining structure is so common in our Scheme MOPs that we are now developing a new kind of object-oriented language specifically to support it.

A more detailed discussion of Sartor, together with examples of specializing both its compiler and runtime MOPs are presented in [12].

Anibus

We have also developed a MOP-based parallelizing Scheme compiler, called Anibus. Anibus is our first metaobject protocol to operate entirely at compile-time; there is no runtime object-oriented dispatch.

In Anibus, the programmer's model of the base language is Scheme, together with a set of parallelization directives, or marks, that provide high-level direction about what parallelization strategy the compiler should use. This general architecture is common in such compilers, as it provides the programmer with a clean separation between the algorithm their program implements and its parallelization.[13, 14, 15, 16]

But, a compiler with just this base model suffers from the same kind of performance problems discussed with CLOS above. Some programs, because of particular properties of their behavior, will require a parallelization strategy other than one of those provided by default. To address this, Anibus provides a metaobject protocol that allows the programer to define new kinds of marks.

The Anibus architecture is similar to Ploy and Sartor in that it operates on program graph metaobjects. The protocol consists of a small number of generic functions, each concerned with a separate aspect of parallelization, such as distribution of data, distribution of computation and synchronization. As the compilation proceeds, these generic functions gradually rewrite the program into a language with more primitive distribution and synchronization constructs.[5] As in Ploy and Sartor, a mark on the source program causes the corresponding program element metaobject to be of that class, and thereby affects what methods will be applicable to that metaobject.

Defining a new parallelization strategy is done simply by defining a new mark class, together with appropriate methods on the protocol generic functions. Very often, this can be done by subclassing one of the existing marks, and only one or two method definitions are required.

A complete description of Anibus, including examples of using it to define several alternative parallelization marks can be found in [17]. A discussion of how the functionality provided by Anibus differs from more traditional parallelizing compilers can be found in [18].

[5] The lower level language is also a variant of Scheme. It is then compiled by a compiler for the target architecture.

Conclusion

This chapter has shown that the idea underlying the CLOS Metaobject Protocol — to provide an open language implementation using object-oriented and reflective techniques to organize a meta-level architecture — is far more general than its incarnation in CLOS, or object-oriented programming, or even Lisp-like languages.

The original intuition behind this idea is that the very thing that makes high-level languages great — that they hide implementation details from the programmer — is also their greatest liability, since it is the inability of the programmer to be able to control those details that can result in poor performance.

One traditional approach to resolving this dilemma has been to hide in the programming language only those implementation details which can be automatically optimized — that is, to keep the language lower level than might otherwise be desirable. This philosophy, which is most clearly evident in languages designed for systems programming [19], is reflected in the following quote [20]:

> I found a large number of programs perform poorly because of the language's tendency to hide "what is going on" with the misguided intention of "not bothering the programmer with details." *N. Wirth, "On the Design of Programming Languages," Information Processing 74, pp. 386-393.*

Another approach has been to explicitly provide the user with declarative extra-lingual mechanisms to advise the implementation (i.e. compiler pragmas) about how some part of the program should be implemented.[6] As discussed in the section on Anibus, this approach can suffer from the problem that specific users may want to instruct the compiler in ways not supported by the supplied pragmas.

Metaobject protocols provide an alternative framework that opens the language implementation up to user "intervention." The major difference is that they are imperative in nature, and as a result are much more powerful. The metaobject protocol approach also distinguishes itself from previous approaches in that it allows the programmer to alter the semantics of the language. While this latter may seem controversial, we have found, in our work with the CLOS MOP, that properly used, programmers can derive tremendous benefit and program clarity from being able to customize the language semantics.

Acknowledgements

Work on the CLOS Metaobject Protocol was done jointly with Jim des Rivières. The development of more general ideas about the need for open language implementations and metaobject protocols has benefited tremendously from discussions with Hal Abelson, Mike Dixon, John Lamping and Brian Smith.

This work has also benefited from the contributions and feedback of the entire CLOS Metaobject Protocol user community. Their willingness to experiment with our rapidly changing prototype implementation (PCL) allowed us to develop the CLOS MOP and these ideas much more quickly than would otherwise have been possible.

[6] We call these extra-lingual because they do not in general affect the program semantics.

References

[1] Gregor Kiczales. Towards open implementations – a new model of abstraction in software engineering. In *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures*, 1992. Also to appear in forthcoming PARC Technical Report.

[2] D. Moon. Object-oriented programming with Flavors. In *OOPSLA '86 Conference Proceedings, Sigplan Notices* **21***(11)*. ACM, Nov 1986.

[3] Daniel G. Bobrow and Mark Stefik. The Loops manual. Technical report, Xerox PARC, 1983.

[4] Gregor J. Kiczales and Luis H. Rodriguez Jr. Efficient method dispatch in PCL. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 99–105, 1990.

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

[6] D.G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. Commonloops: Merging Lisp and object-oriented programming. In *OOPSLA '86 Conference Proceedings, Sigplan Notices* **21***(11)*. ACM, Nov 1986.

[7] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 147–155, 1987.

[8] Gregor Kiczales and John Lamping. Issues in the design and documentation of class libraries. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, 1992. To Appear.

[9] IEEE Std 1178-1990. *Ieee Standard for the Scheme Programming Language.* Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[10] Shinn-Der Lee and Daniel P. Friedman. Quasi-static scoping: Sharing variable bindings across multiple lexical scopes. Technical report, Indiana University Computer Science Department, Aug 1992. Forthcoming Technical Report.

[11] Amin Vahdat. The design of a metaobject protocol controlling the behavior of a scheme interpreter. To appear in forthcoming PARC Technical Report., August 1992.

[12] J. Michael Ashley. Open compilers. To appear in forthcoming PARC Technical Report., August 1992.

[13] Rajive Bagrodia and Sharad Mathur. Efficient implementation of high-level parallel programs. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 142–151, 1991.

[14] William Weihl, Eric Brewer, Adrian Colbrook, Chrysanthos Dellarocas, Wilson Hsieh, Anthony Joseph, Carl Waldspurger, and Paul Wang. Prelude: A system for portable parallel software. Technical Report MIT/LCS/TR-519, MIT, October 1991.

[15] J. Allen Yang and Young il Choo. Meta-crystal – a metalanguage for parallel-program optimization. Technical Report YALEU/DCS/TR-786, Yale University, April 1990.

[16] Monica S. Lam and Martin C. Rinard. Coarse-grain parallel programming in Jade. In *Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 94–105, 1991.

[17] Luis H. Rodriguez Jr. Coarse-grained parallelism using metaobject protocols. Master's thesis, Massachusetts Institute of Technology, 1991.

[18] Luis H. Rodriguez Jr. Towards a better understanding of compile-time mops for parallelizing compilers. In *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures*, 1992. Also to appear in forthcoming PARC Technical Report.

[19] Greg Nelson, editor. *Systems Programming with Modula-3.* Prentice Hall, Englewoods Cliff, New Jersey, 1991.

[20] Niklaus Wirth. On the design of programming languages. In *Information Processing 74*, 1974.