Implementation Techniques for Efficient Reflective Languages*

Shigeru Chiba
Department of Information Science, The University of Tokyo
chiba@is.s.u-tokyo.ac.jp

Abstract

This paper describes three techniques for efficiently implementing reflective languages, which have been proposed by several researchers including the author. The paper presents that, despite their superficial differences, the three techniques are similar techniques based on the same idea, which is to perform meta computation as early as possible. The differences are the way to transform the protocol and what kind of languages the technique is suitable for. This paper also shows that one of the three techniques, a compile-time MOP, is suitable for statically typed languages, and experimentally illustrates that the technique actually improves the execution performance while keeping the ability for extension.

1 Introduction

Reflective languages allow programmers to extend the languages through meta interface often called a metaobject protocol (MOP). Their practical significance have been presented by a number of researchers [20, 4, 19, 11, 23], while the difficulty of efficient implementation of reflective languages has been also recognized. A typical technique for avoiding this problem has been to restrict language mechanisms that programmers can extend, and minimize overheads by extensibility.

On the other hand, a few implementation technique for solving this problem has been proposed so far. The compile-time MOP developed for OpenC++ [5] is one of those technique. Although it might look like a totally

*Technical Report 97-06, Department of Information Science, The University of Tokyo, 1997.

different meta architecture because of its superficial features, it is an implementation technique generally applicable to ordinary reflective languages such as ABCL/R [24] and CodA [18]. Thus, various useful features of these languages can be kept even if this technique is used for the implementation. For example, if a reflective language includes a meta meta level, the implementation using the compile-time MOP can also include a meta meta level. This feature is one of advantages of the compile-time MOP for OpenC++ against other compile-time MOPs, such as Intrigue [15], Anibus [21], and MPC++ [13]. These MOPs are not implementation techniques but original meta architectures. They do not support a meta meta level.

This paper compares a compile-time MOP with other major implementation techniques for efficient reflective languages: currying and partial evaluation. The currying technique was originally developed for the CLOS MOP [14]. The technique using partial evaluation has been studied by several researchers [22, 8, 1, 17], and a notable study of this technique is the implementation of ABCL/R3 [17]. This paper describes that all the three techniques are similar and based on the same idea, which is to move meta computation from runtime to earlier time such as compile time, and that the three techniques are applicable to ordinary reflective languages but they are suitable for different kinds of languages.

In the rest of this paper, we first show a very simple but typical reflective language and its naive implementation. Then, we present how the three implementation techniques are applied to that reflective language. The technique using partial evaluation does not require the protocol designer to transform the protocol, but the techniques using currying and a compile-time MOP require to transform the protocol in order to allow the protocol implementor to move meta computation to earlier time. We discuss that this difference affects what kind of reflective languages those techniques are suitable for. For example, a compile-time MOP is suitable for statically typed languages. To illustrate this, we present experimental results of performance improvement in a static language by a compile-time MOP. Finally, we summarize features of the three techniques.

2 Naive Implementation

First of all, we show a very simple reflective language and a naive implementation of that language as a reference system. We choose a Scheme-like syntax for the reflective language but also introduce a simple object system, which is extensible via a meta-level program. In this language, programmers may write the following code:

```
(let* ((x ...)
       (y ...)
       (p (new point x y)))
  (<- p move 11 y))
```

The operator `new` makes a new instance of a class. In the above example, a class `point` is instantiated and a variable `p` is bound to the instantiated object. `x` and `y` are arguments passed to the initializer function. Another operator `<-` is used to access an instance variable (or a slot) of an object and to call a method for an object. The expression `(<- p move 11 y)` calls a method `move` for `p` with arguments `11` and `y`.

Meta-level Program

Meta interfaces for extending a language are classified into two major design categories: one is a class-based MOP (metaobject protocol[1]) [2, 9, 3] and the other is an object-based MOP [16]. Both the two kinds of protocols allow programmers to extend a language by using metaobjects, which are meta representation of the base-level program. However, a class-based MOP uses classes for primary metaobjects, but an object-based MOP provides special objects representing the behavior of a base-level object and uses these objects for primary metaobjects. We select a class-based MOP for the reflective language used in this paper. This is just because the CLOS MOP and the OpenC++ MOP are class-based MOPs, and this selection does not affect the substantial argument in this paper. The argument is also applicable to an object-based architecture.

With a class-based MOP, language extension is defined by a new metaclass. Suppose that we want to extend the language so that the number of instantiation of the class `point` is counted. To do this extension, the protocol user should write a new metaclass `count-class` for the class `point`:

```
(class count-class (class) (counter))

(defmethod count-class make-instance (env expr)
  (<- self counter (+ (<- self counter) 1))
  (<- super make-instance env expr))

      :
```

[1] A protocol means an interface to access an object and originally comes from Smalltalk.

This meta-level program defines the metaclass `count-class`, which is a subclass of the default metaclass `class` and has an instance variable `counter`. The metaclass `count-class` also has a method `make-instance`, which is invoked when the class `point` is instantiated. This method takes two arguments `env` (current environment) and `expr` (the `new` expression), and increments the value of the instance variable `count` and calls `make-instance` of the super class.

To make this metaclass effective, the protocol user also needs to specify that the class `point` is an instance of `count-class`. Thus the base-level program includes this expression:

```
(metaclass point count-class)
```

Implementation

A naive implementation of the protocol shown above is an interpreter constructed in an object-oriented style. The program of the interpreter would be something like this:

```
(define (eval-expr expr env)
  (cond ((new? expr)
         (let ((class (lookup-class (second expr))))
           (<- class make-instance expr env)))
        ((method-call? expr)
         (let* ((object (eval-expr (second expr) env))
                (class  (object-type object)))
           (<- class call-method expr env)))
        ((let? expr)
         (eval-let expr env))
        ... ))
```

The interpreter `eval-expr` reads a base-level expression and dispatches it to a metaobject that is responsible for the interpretation. For example, if the base-level expression `expr` is `(new point x y)`, then `eval-expr` looks up the `point` class metaobject and calls the method `make-instance` for the metaobject. The expression `expr` is interpreted by `make-instance`.

The meta-level program written by the user is regarded as part of the program of the interpreter. In fact, if `eval-expr` calls the method `make-instance` for the `point` class metaobject, then `make-instance` written by the user is first invoked:

```
(defmethod count-class make-instance (expr env)
  (<- self counter (+ (<- self counter) 1))
  (<- super make-instance expr env))
```

Since this method relies on the super class for the interpretation of the given expression, the implementor of the protocol should provide the default metaclass to perform the default interpretation. The program shown below is the method `make-instance` of the default metaclass `class`:

```
(defmethod class make-instance (expr env)
  (let* ((size   (<- self compute-size))
         (init   (<- self lookup-method 'initialize))
         (object (allocate-object size))
         (args   (eval-arguments (cddr expr) env)))
    (apply init (cons object args))
    object))
```

Execution Model

To make the essential idea clear and make it easy to compare with other implementation techniques, we describe this naive implementation with formal notations. Let L (language) be a meta-level program, P be a base-level program, and D be base-level data given to the base-level program. L also includes the program of the interpreter `eval-expr` shown above. Then execution of a reflective program under the naive implementation is described as follows:

$$\mathcal{E}[\![L]\!](P, D) \qquad \text{where} \quad \mathcal{E} : Prog \rightarrow Prog \times Env \rightarrow Env$$

Here, $\mathcal{E}$ is a semantic function. It intuitively means a compiler or an interpreter that executes a program L. The result of $\mathcal{E}[\![L]\!]$ is a function that takes P and L as inputs. In other words, the meta-level program L is executed to interpret the base-level program P with the base-level data D.

3 Currying

As widely known, the naive implementation involves serious performance penalties, compared with non-reflective languages. To improve the execution performance, a few implementation techniques have been developed so far. We first show the currying technique, which is originally developed for the CLOS MOP [14].

The problem of the naive implementation is that too much meta computation is executed at runtime even though part of the computation can be in principle executed earlier. To move part of the computation to earlier time as much as possible, the currying technique transforms a protocol so that the resulting function of $\mathcal{E}[\![L]\!]$ is *curried*.

Curried Protocol

Recall that the method `make-instance` takes an interpreted expression and an environment at the same time and performs the interpretation. The currying technique requires the protocol designer to transform the protocol, so that `make-instance` written by the protocol user takes only an expression and returns a function that takes an environment and performs the interpretation:

```
(defmethod count-class make-instance (expr)
  (let ((f (<- super make-instance expr)))
    (lambda (env)
      (<- self counter (+ (<- self counter) 1))
      (f env))))
```

Note that `make-instance` is now a higher-order function that returns a `lambda` function. Also, note that `(<- super make-instance expr)` is executed only once when the method above is executed. The returned `lambda` function holds the result of `(<- super ...)` but does not execute it again.

Since we transforms the protocol, the protocol implementor also needs to rewrite the default metaclass `class`. Now `make-instance` of the default metaclass returns a function that performs the default interpretation if called with an environment:

```
(defmethod class make-instance (expr)
  (let ((size (<- self compute-size))
        (init (<- self lookup-method 'initialize))
        (args (eval-arguments (cddr expr))))
    (lambda (env)
      (let ((object (allocate-object size)))
        (apply init (cons object (args env)))
        object))))
```

The values `size` and `init` are computed when `make-instance` is called, since they can be computed without an environment. Note that `eval-arguments` also returns a function that takes an environment and returns actual arguments.

Improved Implementation

The curried protocol itself does not improve execution performance, but it gives opportunities for the protocol implementor to write an efficient implementation. The implementor may memoize functions returned from metaobjects and reuse the memoized functions if the same expression is executed again. To perform memoization, `eval-expr` should be rewritten as follows:

```
(define (eval-expr expr env)
  (let ((f (lookup-hash-table expr)))
    (if (found? f)
        (f env)
        (let ((g (curried-eval expr)))
          (record-hash-table g expr)
          (g env)))))

(define (curried-eval expr)
  (cond ((new? expr)
         (let ((class (lookup-class (second expr))))
           (<- class make-instance expr)))
        ((method-call? expr)
         (eval-method-call expr))
        ((let? expr)
         (curried-eval-let expr))
        ...))
```

`curried-eval` is called only when the expression `expr` is first executed, and the function returned by `curried-eval` is recorded in a hash table to avoid calling `curried-eval` again. This memoization considerably improves execution performance because it avoids redundantly performing computation that does not need an environment.

However, this simple memoization does not improve performance of method calls because dispatching a method call requires the type of the receiver object but this type is not determined without the environment. Thus, most of computation for executing a method call remains in a function returned by `curried-eval`. See the definition of `eval-method-call`:

```
(define (eval-method-call expr)
  (let ((receiver (eval-expr (second expr))))
    (lambda (env)
      (let* ((object (receiver env))
             (class  (object-type object))
             (f      (<- class call-method expr)))
        (f env)))))
```

This problem can be solved without changing the protocol. The protocol implementor should also memoize the function returned from a metaobject. The memoized function can be reused if the same method is called for the same class of object. To do this memoization, `eval-method-call` should be rewritten as follows:

```
(define (eval-method-call expr)
  (let ((receiver (eval-expr (second expr))))
    (lambda (env)
```

```
(let* ((object (receiver env))
       (class  (object-type object)))
       (f      (lookup-hash-table class expr)))
  (if (found? f)
      (f env)
      (let ((g ((<- class call-method expr))))
        (record-hash-table g class expr)
        (g env))))))
```

Since most of computation for executing a method call is moved out of the memoized function indicated by `f`, the execution performance is significantly improved once the method is called for the class of the object.

Execution Model

The implementation using the currying technique is described as follows:

$$\mathcal{E}[\![L_1]\!](P)(D) \qquad \text{where} \quad \mathcal{E} : Prog \to Prog \to Env \to Env$$

Here, L_1 is a meta-level program for the curried version of the protocol. The result of $\mathcal{E}[\![L_1]\!]$ is a curried function that takes only a base-level program P as an input and returns a function that takes base-level data D.

The idea of this technique is that the curried protocol allows the protocol implementor to memoize an intermediate result $\mathcal{E}[\![L_1]\!](P)$ and reuse it at next time. The implementor may also compute $\mathcal{E}[\![L_1]\!](P)$ in advance at load time or at compile time.

Moreover, if needed, the protocol implementor may modify the implementation and use part of base-level data D_1 when computing a memoized function:

$$\mathcal{E}[\![L_2]\!](P, D_1)(D_2) \qquad \text{where} \quad \mathcal{E} : Prog \to Prog \times Env \to Env \to Env$$

Here, L_2 is a meta-level program for this implementation. Since $\mathcal{E}[\![L_2]\!]$ takes not only a base-level program but also part of base-level data, it may execute more computation and reduce runtime overheads.

4 Partial Evaluation

The currying technique needs to transform a protocol and makes it slightly difficult to use. It also requires that a very efficient `lambda` function is available in the language. Hence a few researchers have proposed another technique, which is the use of partial evaluation [10, 12] for implementing a reflective language. With this technique, the protocol designer does not need to transform the protocol, or a `lambda` function is not necessary.

A partial evaluator takes a program and part of its inputs and produces another program that takes the rest of the inputs. The result of the latter program is equivalent to that of the former one. Let P_E be a program of the partial evaluator, P be a processed program, D_1 and D_2 be inputs of the processed program. Then this equation is given:

$$\mathcal{E}[\![P']\!](D_2) = \mathcal{E}[\![P]\!](D_1, D_2) \qquad \text{where} \quad P' \text{ is } \mathcal{E}[\![P_E]\!](P, D_1).$$

Implementation with Partial Evaluation

The idea of this technique is to apply partial evaluation to a meta-level program with a base-level program. Recall that the meta-level program takes the base-level program and base-level data as inputs. The meta-level program is *partially* evaluated only with the base-level program but without the base-level data. This idea enables to move meta computation to compile time; programmers can apply the partial evaluation in advance, perform meta computation, and execute the produced program at runtime.

If partial evaluation is applied to `eval-expr` for the naive implementation shown in Section 2:

```
(define (eval-expr expr env)
  (cond ((new? expr)
         (let ((class (lookup-class (second expr))))
           (<- class make-instance expr env)))
        ((method-call? expr)
         (let* ((object (eval-expr (second expr) env))
                (class  (object-type object)))
           (<- class call-method expr env)))
        ((let? expr)
         (eval-let expr env))
        ... ))
```

and this base-level program:

```
(new point x y)
```

is given for the argument `expr`, then the partial evaluator executes `eval-expr` as far as it can do without the value of another argument `env`. First, the partial evaluator executes `(new? expr)`, and then it also executes `(lookup-class (second expr))`. These two expressions can be executed only if the value of `expr` is given. Furthermore, the partial evaluator executes the next expression `(<- class make-instance expr env)` as much as possible. Eventually, the partial evaluator produces a program like this:

```
(define (eval-expr* env)
  (apply point-initialize
         (cons (allocate-object 2)
               (list (lookup-env env 'x) (lookup-env env 'y)))))
```

The result of this partially evaluated program `eval-expr*` is equal to that of the original program `eval-expr` if the argument `expr` is `(new point x y)`. Note that `eval-expr*` runs faster than `eval-expr` since most of meta computation has been already done for `eval-expr*`.

The program produced by the partial evaluator is similar to the resulting value of `curried-eval` shown in Section 3, except that the former is a program but the latter is a `lambda` function. Thus, the technique using partial evaluation can achieve performance improvement similar to the currying technique. Programmers can run the partial evaluator at compile time and run the produced program at runtime instead of memoized functions. Benefits of using partial evaluation are: (1) it does not need a `lambda` function, and (2) the protocol is easier to use because the protocol designer does not need to transform the original protocol but can keep it original and intuitive.

A drawback of this technique is that developing a powerful partial evaluator is extremely difficult. To get the same effects that the currying technique provides, a partial evaluator has to perform not only simple constant propagation shown above, but also more elaborate processing such as code duplication. In fact, only a partial evaluator for a relatively simple reflective language has been reported so far [17]. Developing a partial evaluator for a practical language such as C++ and Common Lisp would be very difficult. Further research efforts are awaited.

Moreover, partial evaluation is not suitable for handling dynamic method dispatch. Recall that, in the previous section, we showed that most of meta computation for a method call cannot be executed without the type of the receiver object, which is not determined until the environment `env` is given. The protocol implementor, therefore, memoized a pair of the type of the receiver object and the function returned from the metaobject. This kind of optimization is difficult for a partial evaluator to automatically perform. An *ad hoc* technique might be needed for this.

Execution Model

Let P_E be the program of a partial evaluator, L be a meta-level program, P be a base-level program, and D be base-level data. The implementation using partial evaluator is described as follows:

$$\mathcal{E}[\![\mathcal{E}[\![P_E]\!](L, P)]\!](D)$$

Note that $\mathcal{E}[\![P_E]\!](L, P)$ corresponds to $\mathcal{E}[\![L_1]\!](P)$ for the currying technique. Both of them can be computed in advance and reduce meta computation needed at runtime. Their difference is that $\mathcal{E}[\![P_E]\!](L, P)$ is a program but $\mathcal{E}[\![L_1]\!](P)$ is a function. So the former must be explicitly executed by $\mathcal{E}$.

5 Compile-time MOP

The last technique shown in this paper is a compile-time MOP developed by the author for OpenC++ [5]. This is a kind of currying technique that is effective even without a `lambda` function, or it can be regarded as partial evaluation by hand.

The technique using a compile-time MOP requires the protocol designer to transform the protocol as we did for the currying technique. For example, `make-instance` written by the protocol user should be changed to take only an interpreted expression. However, `make-instance` does not return a function but a program if this technique is used:

```
(defmethod count-class make-instance (expr)
  (let ((f (<- super make-instance expr)))
    `(begin (<- self counter (+ (<- self counter) 1))
            ,f)))
```

This returns a quoted expression rather than a `lambda` function, although the meaning of the quoted expression is quite similar to the `lambda` function for the currying technique. In fact, the two versions of `make-instance` are symmetric and translation between them is straightforward. Like the version for the currying technique, this version for the compile-time MOP computes the value of `f` when `make-instance` is called, and substitute it for all occurrences of `,f` in the quoted expression.

The default metaobjects also needs to be reimplemented. The new implementation can be derived by simple translation from the implementation for the currying technique. The major change is only that a `lambda` function is translated into a quoted expression:

```
(defmethod class make-instance (expr)
  (let ((size (<- self compute-size))
        (init (<- self lookup-method 'initialize))
        (args (eval-arguments (cddr expr))))
    `(let ((object (allocate-object ,size)))
        (apply ,init (cons object ,args))
        object)))
```

Implementation of the Compile-time MOP

The compile-time MOP allows the protocol implementor to call metaobjects at compile time and directly execute the program returned from the metaobject at runtime. This is similar to the implementation with the currying technique although the compile-time MOP does not memoize the program returned from the metaobject but substitutes it for the original program. This feature enables better improvement than the currying technique since metaobjects are never executed at runtime and the cost of looking up a memoized `lambda` function is not needed at runtime. Also, overheads for applying a `lambda` function are eliminated.

To call metaobjects at compile time, however, the types of objects need to be determined at compile time especially for processing method calls. There are two solutions of this problem:

- Use static type information, or
- Restrict the ability of metaobjects to customize the behavior of a base-level program

The first solution is suitable for statically typed languages such as C++. The metaobjects are selected according to static types. If this solution is used, the base-level program is processed by the program shown below and the produced program is executed at runtime:

```
(define (compile-expr expr static-env)
  (cond ((new? expr)
         (let ((class (lookup-class (second expr))))
           (<- class make-instance expr static-env)))
        ((method-call? expr)
         (let* ((object (second expr))
                (class (static-type object static-env)))
           (<- class call-method expr static-env)))
        ((let? expr)
         (compile-let expr static-env))
        ... ))
```

The second solution is effective even for dynamic languages but it restricts the capability of the metaobjects to control the behavior of a base-level program. Since the metaobject for handling a method call cannot be determined at the caller side, this solution executes the method dispatch in the default way and passes the control to the metaobject when executing the selected method. After the method dispatch is done, the type of object is determined. Thus, the expression that the metaobject receives is not the method call expression but the method body. The following program is another version of `compile-expr` for this solution:

```
(define (compile-expr expr)
  (cond ((new? expr)
         (let ((class (lookup-class (second expr))))
           (<- class make-instance expr)))
        ((method-call? expr)
         (default-compile-method-call expr))
        ((defmethod? expr)
         (let ((class (lookup-class (second expr))))
           (<- class call-method expr)))
        ((let? expr)
         (compile-let expr))
        ... ))
```

With this solution, the protocol user cannot customize the execution of a method call expression or the mechanism of method dispatch. The method call expression is compiled by `default-compile-method-call` in the default way. Instead, the user can customize only the execution of a method body defined by `defmethod`. That is, a metaobject controls only callee-side execution. If the given `expr` is a `defmethod` expression, then `compile-expr` looks up a class metaobject for the method and calls `call-method` for the metaobject to compile the `defmethod` expression.

Execution Model

Let L_3 be a meta-level program including `compile-expr`, P be a base-level program, and D be base-level data. Then the implementation based on a compile-time MOP is described as follows:

$$\mathcal{E}[\![\mathcal{E}[\![L_3]\!](P)]\!](D)$$

Since the result of $\mathcal{E}[\![L_3]\!](P)$ is a program, it needs to be explicitly processed by $\mathcal{E}$. But, except that point, this implementation is equivalent to the implementation with the currying technique.

Also, the meta-level program L_3 is equivalent to the result of partial evaluation applied to P_E and L, which is $\mathcal{E}[\![P_E]\!](P_E, L)$, where L is the meta-level program for the naive implementation. For this reason, L_3 can be regarded as a program of the partial evaluator specialized for L. In other words, a compile-time MOP is a technique to have programmers to write a partial evaluator specialized for L by hand. Although a general partial evaluator is extremely difficult to write, a specialized one such as L_3 can be written by hand since L_3 is derived from L by simple transformation.

6 Experiment

A compile-time MOP is suitable for statically typed languages. If it is used for such a language, all meta computation can be performed at compile time. Even the cost of switching the control to a specialized program produced by a metaobject is not needed at runtime. Recall that the currying technique requires this cost for looking up a memoized function.

To illustrate how this technique improves runtime performance, this section shows execution speed of distributed objects implemented with a meta-level program on top of C++, which is a statically typed language. We compared two versions of C++ MOPs: one is OpenC++ version 1 [7], which is a naively implemented protocol, and the other is OpenC++ version 2 [6], which is a compile-time MOP.

To implement the distributed objects, the meta-level program alters the behavior of member-function calls for the distributed objects, so that, if a member function is called for a remote object, a network message is sent to the machine where the remote object exists, and when the network message arrives there, the called member function is invoked. The runtime routines for executing this are appropriately invoked by the meta-level program. The overheads due to the meta-level program are runtime costs to invoke those runtime routines. The execution of the runtime routines itself is not overhead since it is mandatory for implementing the distributed objects.

We measured execution speed of calling a null member function for a remote object with different numbers of `int` arguments. The member function returns an `int` value. For comparison, we also measured execution speed of a null member function call optimally implemented by hand.

Table 1: Elapsed time of a null member function call (μsec)

# of arguments	0	1	2	3	4
OpenC++ v1	3.3	3.7	4.1	4.6	5.1
OpenC++ v2	1.4 (60)	1.5 (65)	1.5 (69)	1.6 (72)	1.6 (74)
Hand-coded	1.0 (35)	1.0 (38)	1.0 (41)	1.0 (44)	1.0 (47)

SPARC Station 10, SuperSPARC 50MHz, GNU g++ 2.6.3
Transfer time on network is not included. The numbers in parentheses indicate the number of instructions.

Table 1 shows the results of the measurement. The elapsed time shown in the table does not include transfer time on network since the transfer time is several milliseconds and makes overheads due to the meta-level program negligible. The experiment shows that the compile-time MOP is about three times faster than the naively-implemented MOP. But, comparing with the hand-coded program, the compile-time MOP is still 1.5 times slower in terms of elapsed time. In fact, it consumes about 1.6 to 1.7 times more instructions than the hand-coded program.

These extra instructions are needed because the meta-level program produces a message dispatch table for each class. An incoming network message is first sent to the object's class and then dispatched to the called member function. But the hand-coded program optimizes this dispatch mechanism and produces a single dispatch table for the whole program. An incoming network message is directly dispatched to the called member function. This drawback of the meta-level program is for supporting separate compilation. Since the meta-level program cannot know all the classes used in the whole program, it cannot produce a single dispatch table to cover all the classes. If the capability of the OpenC++ MOP is augmented so that the meta-level program can access other source files, the difference in execution speed would be reduced.

7 Summary

This paper showed that a compile-time MOP is a general implementation technique that is applicable to ordinary reflective languages, and it also presented similarity and difference among a compile-time MOP and other major implementation techniques, currying and partial evaluation. Moreover, this paper experimentally showed that a compile-time MOP improves execution performance if it is applied to a statically-typed reflective language.

From the viewpoint of the protocol users, the technique using partial evaluation is preferable because it does not change the protocol. But this technique is difficult for the protocol implementor to use for practical languages. In fact, a non-trivial partial evaluator is not known for C++ or Common Lisp. Also, code explosion caused by code duplication is a problem that should be solved to use this technique for practice.

The other techniques, currying and a compile-time MOP, are more feasible for the protocol implementor than partial evaluation, but they require the protocol designer to transform the protocol. Although this transformation is straightforward for the protocol designer, it forces the protocol users to write a less intuitive meta-level program. The ability of these techniques for performance improvement is comparable to partial evaluation; they can be regarded as partial evaluation by hand.

An advantage of the currying technique is that it is suitable for dynamically typed languages. It allows to move meta computation to earlier time even if the meta computation requires runtime data. However, the currying technique needs efficient `lambda` function to implement it and implies runtime penalties because dispatching the control to memoized functions is performed at runtime.

A compile-time MOP can be regarded as a version of the currying technique that are effective without a `lambda` function. A primary difference from the currying technique is that a compile-time MOP is suitable for statically typed languages. If this technique is applied to such a language, all meta computation can be moved to compile time, so that the base-level program runs with almost zero overheads.

References

[1] Asai, K., S. Matsuoka, and A. Yonezawa, "Duplication and Partial Evaluation," *Lisp and Symbolic Computation*, vol. 9, pp. 203–241, 1996.

[2] Bobrow, D. G., K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel, "CommonLoops, Merging Lisp and Object-Oriented Programming," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 17–29, ACM, September 1986.

[3] Briot, J. and P. Cointe, "Programming with Explicit Metaclasses in Smalltalk-80," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 419–431, ACM, October 1989.

[4] Buschmann, F., K. Kiefer, F. Paulisch, and M. Stal, "The Meta-Information-Protocol: Run-Time Type Information for C++," in *Proc. of the Int'l Workshop on Reflection and Meta-Level Architecture* (A. Yonezawa and B. C. Smith, eds.), pp. 82–87, 1992.

[5] Chiba, S., "A Metaobject Protocol for C++," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, vol. 30 of *Sigplan Notices*, pp. 285–299, ACM, 1995.

[6] Chiba, S., "OpenC++ Programmer's Guide for Version 2," Technical Report SPL-96-024, Xerox PARC, 1996.

[7] Chiba, S. and T. Masuda, "Designing an Extensible Distributed Language with a Meta-Level Architecture," in *Proc. of the 7th European Conference on Object-Oriented Programming*, LNCS 707, pp. 482–501, Springer-Verlag, 1993.

[8] Chiba, S. and T. Masuda, "Open C++ and Its Optimization (Extended Abstract)," in *Proc. of OOPSLA '93 Workshop on Reflection and Metalevel Architectures*, Oct. 1993.

[9] Cointe, P., "Metaclasses are first class: The ObjVlisp model," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 156–167, 1987.

[10] Ershov, A., "On the Essence of Compilation," in *Formal Description of Programming Concepts* (E. Neuhold, ed.), pp. 391–420, North-Holland, 1978.

[11] Fabre, J., V. Nicomette, T. Perennou, R. J. Stroud, and Z. Wu, "Implementing Fault Tolerant Applications using Reflective Object-Oriented Programming," in *Proc. of the 25th IEEE Symp. on Fault-Tolerant Computing Systems*, pp. 489–498, 1995.

[12] Futamura, Y., "Partial Computation of Programs," in *Proc. of RIMS Symposia on Software Science and Engineering*, no. 147 in LNCS, pp. 1–35, 1982.

[13] Ishikawa, Y., A. Hori, M. Sato, M. Matsuda, J. Nolte, H. Tezuka, H. Konaka, M. Maeda, and K. Kubota, "Design and Implementation of Metalevel Architecture in C++ — MPC++ Approach —," in *Proc. of Reflection 96*, pp. 153–166, Apr. 1996.

[14] Kiczales, G., J. des Rivières, and D. G. Bobrow, *The Art of the Metaobject Protocol.* The MIT Press, 1991.

[15] Lamping, J., G. Kiczales, L. Rodriguez, and E. Ruf, "An Architecture for an Open Compiler," in *Proc. of the Int'l Workshop on Reflection and Meta-Level Architecture* (A. Yonezawa and B. C. Smith, eds.), pp. 95–106, 1992.

[16] Maes, P., "Concepts and Experiments in Computational Reflection," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 147–155, 1987.

[17] Masuhara, H., S. Matsuoka, K. Asai, and A. Yonezawa, "Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Languages Using Partial Evaluation," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 300–315, 1995.

[18] McAffer, J., "Meta-level Programming with CodA," in *Proc. of the 9th European Conference on Object-Oriented Programming*, LNCS 952, pp. 190–214, Springer-Verlag, 1995.

[19] Okamura, H. and Y. Ishikawa, "Object Location Control Using Meta-level Programming," in *Proc. of the 8th European Conference on Object-Oriented Programming*, LNCS 821, pp. 299–319, Springer-Verlag, 1994.

[20] Paepcke, A., "PCLOS: Stress testing CLOS Experiencing the metaobject protocol," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 194–211, 1990.

[21] Rodriguez Jr., L. H., "Coarse-Grained Parallelism Using Metaobject Protocols," Techincal Report SSL-91-61, XEROX PARC, Palo Alto, CA, 1991.

[22] Ruf, E., "Partial Evaluation in Reflective System Implementation," in *Proc. of OOPSLA'93 Workshop on Reflection and Metalevel Architectures*, 1993.

[23] Stroud, R. J. and Z. Wu, "Using Metaobject Protocols to Implement Atomic Data Types," in *Proc. of the 9th European Conference on Object-Oriented Programming*, LNCS 952, pp. 168–189, Springer-Verlag, 1995.

[24] Watanabe, T. and A. Yonezawa, "Reflection in an Object-Oriented Concurrent Language," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 306–315, 1988.