—to appear in ECOOP'98 proceedings—

Design and Partial Evaluation of Meta-objects for a Concurrent Reflective Language

Hidehiko Masuhara[1] and Akinori Yonezawa[2]

[1] Department of Graphics and Computer Science,
Graduate School of Arts and Sciences, University of Tokyo
masuhara@graco.c.u-tokyo.ac.jp
[2] Department of Information Science, University of Tokyo
yonezawa@is.s.u-tokyo.ac.jp

Abstract. Customizable meta-objects are a powerful abstraction for extending language features and implementation mechanisms, but interpretive execution suffers from severe performance penalty. Some of this penalty can be reduced by applying partial evaluation to meta-*interpreters,* but partial evaluation of meta-*objects* in existing concurrent object-oriented languages is ineffective. This paper proposes a new meta-object design for our reflective language ABCL/R3. It yields meta-objects that can be optimized effectively using partial evaluation. The crux of the design is the separation of state-related operations from other operations, and this separation is accomplished by using reader/writer methods in our concurrent object-oriented language called Schematic. Our benchmark trials show that non-trivial programs with partially evaluated meta-objects run more than six times faster than ones that are interpreted by meta-objects. In addition, a partially evaluated program that uses a *customized* meta-object runs as efficiently as a program that is manually rewritten so as to have the same functionality without using meta-objects.

1 Introduction

1.1 Reflection in Parallel/Distributed Programs

The structure of objects in parallel and distributed applications tends to be complex because they would otherwise not be efficient, reliable, portable and reusable. A number of language mechanisms—inheritance, transaction, object migration, etc.—reducing this complexity have therefore been proposed. Guarded method invocation, for example, which accepts invocation requests conditionally, is useful for describing objects like bounded buffers. In many languages, however, such mechanisms are not always implemented because (1) some are incompatible with each other, (2) supporting a new mechanism can degrade the language efficiency even when the mechanism is not used in a program, and (3) implementation of a new mechanism in the language requires a tremendous amount of effort.

A possibly better approach is to extend languages by using meta-objects. By installing a customized meta-object, the application programmer can use a new language mechanism as if it were built-in. Advanced programmers can even develop their own meta-objects to meet specific requirements. This approach is also beneficial to the language implementors; they can devote themselves to the implementation of *simple core* languages, leaving hard-to-be-implemented mechanisms to be dealt with as extensions.

We are consequently developing a reflective object-oriented concurrent language, ABCL/R3, in which we provide such extensibility by means of *computational reflection*[17, 24]. In ABCL/R3, a *meta-object* provides an abstraction with which the user can extend or modify crucial mechanisms of an object (e.g., method invocation request, method dispatch, state management, and mutual exclusion). In addition, a *meta-interpreter* provides an abstraction that can be used to customize the syntax and semantics of bodies of methods and functions. *Reflective annotations,* which can also be defined by means of meta-level programming, are used as programming directives in base-level programs.

The features of ABCL/R3 have been described in detail elsewhere[19, 21]. The present paper discusses the meta-object design of ABCL/R3 from the viewpoint of efficient implementation.

1.2 Techniques for Implementation of Meta-objects

Despite the extensibility they provide, customizable meta-objects have a problem with regard to efficiency. Assume that a meta-object implements a customized method dispatch algorithm. In a naive implementation, the method dispatch for the corresponding base-level object is achieved by a number of method invocations at the meta-level; i.e., by *interpretive execution.* Moreover, the customization hinders the application of important optimization techniques such as method inlining[4] because such techniques are defined under the assumption that the semantics (rules for method dispatch, in this case) of the language is stable. As a result, the existence of meta-objects easily slow the execution of the language by a factor of more than 10.

There have been several studies on this problem:

- Some parts of the meta-system are not subject to meta-level modification by means of reification. In the reflective language Open C++ version 1[6], for example, only message passing, object creation, and instance variable accesses can be reified. This not only restricts user programmability, but also makes the language model unclear because much of the meta-level functionalities are hidden inside 'black-boxes.' The programmer therefore cannot get a clear view of how his meta-level programming and the black boxes will interact.

  Another example is JDK 1.1, which offers "reflection" API[25]. It supports only introspective operations, which are obviously implemented efficiently. They give, however, few extensibility at the same time.

- The system embodies a set of ad-hoc optimizations transparent to the user. For example, our previous language ABCL/R2[20] assumes that most objects will not be customized, and thus compiles objects without meta-objects. When the user accesses a meta-object, the corresponding object is then switched to general interpreted execution. Its effectiveness, however, is limited to cases where optimization is possible. When optimization is not possible, the interpretation overhead greatly affects the overall performance.
- The compiler inspects behavior of a meta-level program with respect to a base-level program and removes interpretation by using techniques like partial evaluation[12]. This approach is more systematic than the above two approaches, but it requires intensive analysis of the meta-level program that may include user customization.

1.3 Optimization Using Partial Evaluation

A base-level program in a reflective language is executed by an interpreter. The interpreter, which is represented as an object at the meta-level of the language, can be customized by the programmer. Once the base-level program is given, however, most computation that depends only on the base-level program can be performed in advance to the program execution. By removing such computation, a specialized program that contains computation which depends on run-time data for the base-level program, can thus be extracted from the meta-level computation. This process is often referred as *partial evaluation*[12] of the meta-level program with respect to the base-level program, or as the first Futamura projection[7]. Our previous studies successfully apply partial evaluation to the meta-*interpreters* in reflective languages[2, 19].

Meta-*objects*, in theory, could also be optimized by partial evaluation, but they actually cannot because (1) the design of meta-objects in existing reflective languages is not suitable for partial evaluation, and (2) there are few partial evaluators that can deal with concurrent objects. We therefore redesigned meta-objects with consideration to the application of partial evaluation, and here we will show an optimization framework for the resulting meta-objects.

The rest of the paper is organized as follows: In Section 2 we discuss why reasoning about the meta-objects is difficult by reviewing an existing meta-object design. In Section 3 we describe our proposed meta-object design, and in Section 4 we describe its optimization technique by using partial evaluation. In Section 5 we show our performance evaluation of the optimized objects, and in Section 6 we discuss other techniques for the efficient implementation of meta-objects. In Section 7 we conclude by briefly summarizing the paper.

2 Problems of Existing Meta-object Design

Many concurrent object-oriented languages have mutual exclusion mechanisms to assure consistency. A conservative, commonly found, approach is to mutually

exclude all method executions on an object. This approach alleviates the programmers' concern about interference with multiple read/write operations on an instance variable.

The mutual exclusion mechanism in a language drastically affects the meta-object design. This is because (1) the meta-objects explicitly implement the mechanism of base-level objects, and (2) the meta-objects, themselves, are implicitly controlled by a certain mutual exclusion mechanism, which is usually the same one as base-level objects.

In order to meet the above requirements, a meta-object is defined as a *state transition machine* in previous reflective languages. For example, Figure 1 is a simplified definition[1] of the default meta-object in the language ABCL/R[28]. Its state transition diagram can be illustrated as in Figure 2.

A method invocation on a base-level object is represented by an invocation of the method `receive!`[2] on its meta-object. In `receive!`, the message (an object that contains the method name and arguments) is immediately put into the message queue (`queue`), so that it will eventually be processed. If the object is not processing any methods (i.e., `mode` is `'dormant`), the meta-object changes `mode` to `'active` and calls the method `accept!`.

The method `accept!` gets one message from `queue` and lets the evaluator execute the matching method for the message. The evaluator interprets expressions of the method recursively, and when it reaches the end of the base-level method, it invokes the method `finish!` of the meta-object. The method `finish!` examines `queue` for any pending messages received during the evaluation. If `queue` is empty, the meta-object changes `mode` to `'dormant`. Otherwise, it invokes `accept!` again for further execution.

When we apply partial evaluation to this meta-object definition with respect to a certain base-level object, the result is far from satisfactory. The reasons are the following:

- Since the meta-object is defined as a state transition machine, its behavior cannot be determined without static information on some key instance variables such as `mode` and `queue`. For example, if the return value of `(get! queue)` in the method `accept!` were "unknown" (dynamic) at the specialization time, method dispatch (`(find methods m)`) and interpretation of the method body (`(eval evaluator exp env self)`) would be left unspecialized. This means that a large amount of interpretive computation cannot be eliminated by merely applying partial evaluation.
- Information that should be "known" (static) to the partial evaluator is transferred via instance variables between consecutive method invocations. Such information is not available on the receiver's side unless data structures are analyzed extensively. For example, the value of `(get! queue)` in `accept!`, which would be the value of `message` in `receive!`, is crucial for specializa-

[1] The syntax of the definition is that of in Schematic's[27] for the sake of uniformity.
[2] The exclamation mark in the method name conventionally indicates that the method may change the object's state.

```
;;; Class definition
(define-class metaobj ()
  mode queue state methods evaluator) ; instance variables

;;; Method definition for class metaobj
(define-method! metaobj (receive! self message)
  ;; Here, self is bound to the meta-object itself.
  (put! queue message)
  (if (eq? mode 'dormant)              ; If it is dormant, the received
    (begin (set! mode 'active)         ; message is accepted immediately.
           (future (accept! self)))))

;;; method dispatch
(define-method! metaobj (accept! self)
  (let* ((mes (get! queue))                      ; Get a message from the queue.
         (m (find methods mes))                  ; method lookup
         (env (make-env self (formals m) mes))); creation of an evaluation env.
    (future (eval evaluator (exps m) env self)))); evaluation

;;; end of method execution
(define-method! metaobj (finish! self)
  (if (empty? queue)                   ; Check the queue for pending messages.
      (set! mode 'dormant)             ; If none, turn into the dormant mode.
      (future (accept! self))))        ; Otherwise, accept one of them.

;;; meta-interpreter
(define-method! metaobj (eval self exp env owner)
  It evaluates exp under env.  When finished, it invokes finish! of owner.  )
```

The expression (**future** ($m\ \ r\ \ e_1 \ldots e_n$)) *asynchronously* invokes method m of object r with parameters $e_1 \ldots e_n$. It is asynchronous in that the sender continues subsequent computation without waiting for the return value. The expression, ($m\ \ r\ \ e_1 \ldots e_n$), on the other hand, is *synchronous* invocation; the sender waits for the return value.

Fig. 1. Definition of an ABCL/R meta-object.

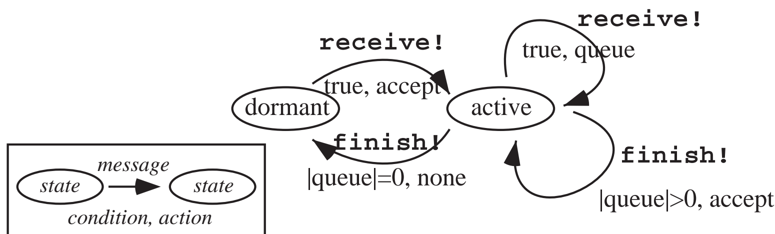

Fig. 2. State transition diagram of an ABCL/R meta-object.

tion, but obtaining it requires analysis of `queue`. This requirement sometimes become overwhelming because `queue` might be a user-defined object.
- The key instance variables are mutable; i.e., their values are changed during execution. The execution model of the meta-objects—ABCM[29] in this case—however, specifies that method invocations will be processed in FIFO order in each object. We thus have to anticipate that the execution of two consecutive methods may be interleaved; i.e., it is safe to assume that mutable instance variables may be changed between method invocations. For example, assume that the method `receive!` invokes the method `accept!`. The variable `queue` at the beginning of `accept!` may have a value different from the one in `receive!` because other methods can be executed before the execution of `accept!`. Though there are partial evaluators that can deal with mutable variables, they regard a mutable variable as unknown (dynamic) unless they can statically determine all update operations to the variable[1, 3].

For the above reasons, a partial evaluator conservatively regards most variables as "dynamic." Without much of "static" information, the partial evaluator yields a program that still performs almost all the computation as the program for the original meta-object does.

3 A New Meta-object Design

We propose, for a reflective concurrent object-oriented language ABCL/R3[19, 21], a meta-object design that can be effectively optimized by partial evaluation. The key idea is to separate, using the reader and writer methods of Schematic, state-related operations from the other operations.

3.1 Reader/Writer Methods

Schematic[23, 27] is a concurrent object-oriented language based on Scheme. It has concurrency primitives such as *future* and *touch* and has class-based objects, but we describe only the *reader/writer methods* because they play key roles in our meta-object design.

The construct `define-method!` defines a *writer method*, which can modify values of instance variables in an object. At the end of a writer method, there should be a form "`(become` *rexp* `:`v_1 e_1 `:`v_2 e_2 `...)`". When this form is evaluated, expressions e_i are first evaluated in sequence. Then the results are set to the variables v_i all at once. Finally, the expression *rexp* is evaluated. The value of *rexp* is returned as the result of the `become` form. Multiple invocations of writer methods on an object are mutually excluded. (Precisely, the evaluation of *rexp* is not excluded; i.e., the *critical section* finishes immediately after updating instance variables.)

The construct `define-method` defines a *reader method*, which cannot modify instance variables. The reader methods are not governed by the mutual exclusion

mechanism; a reader method on an object can even be executed concurrently with a writer method on the same object. During execution of a reader method, the instance variables are bound to the values extracted from the object's state at the beginning of the method. Even when a writer method executes `become` to modify some of the instance variables, reader methods that have started their execution before the `become` operation do not observe the effect of modification.

3.2 Proposed Meta-object Design

The outline of a new meta-object design solving the problems discussed in Section 2 is shown in Figure 3, in which we exploit the reader/writer methods of Schematic. Our design has the following characteristics:

- The behavior of the meta-object is principally defined in the reader methods. Operations that deal with mutable data are defined separately as writer methods or as method invocations on external objects. For example, values of instance variables that are mutable are packed in the mutable vector object `state-values`, and accesses to `state-values` are effected by using the writer methods `cell-set!` and `cell-ref`.
- The meta-object straightforwardly processes each method invocation request and provides mutual exclusion by using blocking operations (e.g., `acquire!` and `release!`). As a result, the meta-object is no longer a state-transition machine. The reader methods, which can be invoked without mutual exclusion, make it possible to define such a meta-object. If the meta-objects were defined with only writer methods, use of the blocking operations would easily lead to deadlock.
- For mutual exclusion, a meta-object has the instance variable `lock` in place of `mode` and `queue`. By default, `lock` is a simple semaphore that has the operations `acquire!` and `release!`. The user can replace `lock` with an arbitrary object, such as a FIFO queue and a priority queue, by means of the meta-level programming.

These characteristics solve the application problems of partial evaluation that were discussed in Section 2. (1) Under the execution model of Schematic[27], it is safe to assume that consecutive invocations of reader methods are not interrupted by other activities; we therefore can use most partial evaluation techniques for sequential languages by regarding the reader methods as functions. (2) Since the "known" (static) information is propagated through the arguments of the method invocations, the partial evaluators easily use such information for specialization. (3) The mutual exclusion mechanism, which is implemented by the blocking operations, gets rid of the dynamic branches (conditionals with dynamic predicates) that would cause a termination-detection problem during specialization.

How the methods in Figure 3 handle messages sent to the base-level object is explained as follows:

```
;;; Class definition
(define-class metaobj ()
  lock state-variables state-values methods evaluator)

;;; Reception of a message
(define-method metaobj (receive self message)
  (if (writer? (selector message))          ; check message type
      (accept-W self message)               ; for a writer method
      (accept self message 'dummy)))        ; for a reader method

;;; Processing for a writer method
(define-method metaobj (accept-W self message)
  (let ((c (make-channel)))                 ; channel for receiving updated state
    (acquire! lock)                         ; mutual exclusion begins
    (let ((result (accept self messages c)))
      (cell-set! state-values (touch c)) ; update instance variables
      (release! lock)                       ; end of mutual exclusion
      result)))

;;; Method lookup and invocation
(define-method metaobj (accept self message update-channel)
  (let* ((m (find methods message))     ; method lookup
         (env (make-env self (formals m) message)))
    (future (eval evaluator (method-body m) env update-channel))))

;;; Meta-interpreter
(define-method evaluator (eval self exp env update-channel)
  It evaluates exp under env. When a become form is evaluated, it creates a
  vector of new updated instance variables, and sends it to update-channel. )
```

The primitive `make-channel` creates an empty channel, which is a communication medium among concurrently running threads. A thread sends a value to a channel c by executing `(reply value` c`)`, and receives from c by `(touch` c`)`.

Fig. 3. Our new meta-object design.

`receive`: The method `receive` simply proceeds to invoke methods `accept-W` or `accept`, depending on the type of the base-level method that is to be invoked.

`accept-W`: The method `accept-W` wraps the method `accept` in the code for mutual exclusion and update of base-level instance variables. It first locks the object and then calls the method `accept` of the same object with a channel `c`. It then waits for a vector of updated instance variables—which is sent by the `become` form in the base-level method—on `c` by executing the `touch` form and updates `state-values` with the received value. Finally, it unlocks the object and returns the result of the method. Note that `accept-W` itself does not modify instance variables.

`accept`: The method `accept` merely looks up a method for a given message and lets the evaluator execute it. The method `make-env`, whose definition is omitted, creates an evaluation environment. It first extracts a vector of instance variable values by executing `(cell-ref state-values)` and then creates an association list that maps each of the instance variable names to the extracted value and maps each formal parameter name of the method to the parameter value in the `message`.

`eval`: The method `eval` of class `evaluator` and its auxiliary methods embody a meta-circular interpreter, which is similar to the traditional Lisp meta-circular interpreter. When it encounters a `become` form, it creates a vector of the updated instance variables and then sends the vector to `update-channel`.

Since the reader/writer methods are not supported in the previous meta-objects, the proposed meta-object design does not have the same semantics as the previous one.

4 Optimization Using Partial Evaluation

In our proposed meta-objects, most operations are defined in the reader methods, and a few invocations on external objects are used for mutual exclusion and state modification. As we stated earlier, the meta-objects can, from the viewpoint of partial evaluation, be regarded as functional programs with I/O-type side-effects. In this section we describe an optimization framework for our meta-objects by using partial evaluation.

The biggest problem we face in using partial evaluation is that there are no partial evaluators appropriate for our purpose because the meta-object is written in a *concurrent* object-oriented language. Although there are studies on partial evaluators for concurrent languages[8, 9, 18], they focus on concurrency and pay little attention to the support of features crucial to sequential languages, such as function closures and data structures.

Our solution is to translate meta-objects into a sequential program and use a partial evaluator for a sequential language. Partial evaluation is applied for each base-level method invocation; i.e., the specialization point is a base-level method invocation. Since the methods of meta-objects exhibit almost sequential

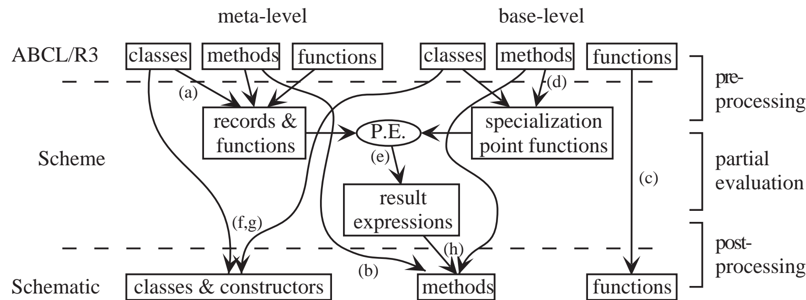

Fig. 4. Overview of our optimization framework.

```
;;; 2d-point
(define-class point () x y)

;;; returns the distance from the origin---a reader method
(define-method point (distance self)
  (sqrt (+ (square x) (square y))))

;;; moves a point---a writer method
(define-method! point (move! self dx dy)
  (become #t :x (+ x dx) :y (+ y dy)))
```

Fig. 5. Example base-level program.

behavior, the partial evaluator for a sequential language can effectively optimize the meta-objects. Concurrency in the meta-objects will be residualized as applications to primitives.

Another problem is compatibility with other objects. The optimized object should support meta-level operations that are defined in the original meta-object. At the same time, the object should behave like a base-level object so that it can be used with other base-level objects. To satisfy these two requirements, our framework generates an object that combines the base- and meta-level objects in a single level. The object has the same methods that are in the original base-level object, and the body of each method is a specialized code of the meta-object.

Figure 4 shows the overview of our optimization framework, in which there are three steps: (1) translation from ABCL/R3 to Scheme, (2) partial evaluation, and (3) translation from Scheme to Schematic. In the following subsections we explain each step in detail by using an example base-level program (Figure 5) and the default meta-object `metaobj` (Figure 3).

4.1 Preprocessing

Meta-object definitions are translated into a Scheme program so that they can be processed by a Scheme partial evaluator (Figure 4(a)). A meta-level object is converted into a record[3] whose fields are its class name and values of instance variables. A reader method is converted into a dispatching function and a class-specific function. The former examines the class-name field in the receiver and calls a matching class-specific function.

Invocations of writer methods that are defined at the meta-level should not be performed during the partial evaluation because they will modify the state of objects. Therefore, the writer methods are not passed to the partial evaluators but are instead simply copied into the resulting Schematic program (Figure 4(b)).

No translations are needed for the base-level definitions, since they are used as data for the meta-level program. Functions, however, are simply copied to the resulting Schematic program (Figure 4(c)).

4.2 Partial Evaluation

We partially evaluate the meta-level program for each *base-level method invocation.* For example, given the base-level program like that in Figure 5, the meta-level computation that will be processed is the one corresponding to the following base-level method invocation:

$$\texttt{(move!}\ p\ dx\ dy\texttt{)}$$
$$\text{where } p = \texttt{point}\{\texttt{x} = x,\ \texttt{y} = y\}.$$

The variables written in italic font (e.g., dx, dy, x, and y) are dynamic data. The data denoted by the variable p is a partially static; it is known as an object of class `point`, but values of instance variables `x` and `y` are dynamic (unknown).

The corresponding meta-level computation is the following expression:

$$\texttt{(receive}\ mobj\ message\texttt{)}$$

where

$$\begin{aligned} mobj &= \texttt{metaobj}\{\texttt{class} = \texttt{'point},\\ &\qquad\qquad \texttt{methods} = \texttt{'((distance (self) ...) ...)},\\ &\qquad\qquad \texttt{state-vars} = \texttt{'(x y)},\ \texttt{state-values} = s,\ \texttt{lock} = l,\\ &\qquad\qquad \texttt{evaluator} = \texttt{(make-evaluator)}\},\\ message &= \texttt{message}\{\texttt{selector} = \texttt{'move!},\ \texttt{arguments} = \texttt{(list}\ dx\ dy\texttt{)}\}. \end{aligned}$$

To partially evaluate a meta-level computation like the above one, we generate a specialization point function for each base-level method (Figure 4(d)). The function takes as its arguments a vector of instance variables, lock, and parameters for the method. When called, it creates *mobj* and *message*, and it invokes the method `receive` on *mobj* (Figure 6). The function is specialized under the assumption that all the arguments are dynamic.

[3] Since our partial evaluator does not natively support records, we further translate the record into `cons`-cells.

```
(define (specalization-point-move!-point state-values lock dx dy)
  (let ((mobj (metaobject 'point '((distance (self) ...) ...)
                          '(x y) state-values lock
                          (make-evaluator)))
        (message (message 'move! (list dx dy))))
    (receive mobj message)))
```

Fig. 6. Specialization point function for method `move!` of class `point`.

An online partial evaluator for Scheme[3] (Figure 4(e)) specializes not only the methods of `metaobj`, but also those of `evaluator`[4]. The compilation techniques of the meta-interpreter are described elsewhere[19].

4.3 Postprocessing

The final step is to translate the results of partial evaluation (in Scheme) back into concurrent objects (in Schematic). This is done by generating class declarations, constructor functions, and methods as shown in Figure 7.

- For each combination of base- and meta-level classes, a specialized class is defined (Figure 4(f)). Since the class is a specialized version of the meta-level class, it has the same instance variables as the original meta-object. (E.g., the class `metaobject**point` in Figure 7.)
- A function that mimics the base-level constructor is defined for each specialized class (Figure 4(g)). For example, the function `point` in Figure 7 is a base-level constructor that creates an object belonging to class `metaobject**point` with proper initial values.
- Methods of the specialized classes are defined (Figure 4(h)). The name of each method is the same as that of the original base-level method. (The method `distance` and `move!` of class `metaobject**point` in Figure 7 are examples.) The specialized object therefore has the same interface as the original base-level program. The body of the method is the result of partial evaluation. Note that because the generated methods are specialized versions of `receive` of the meta-object, they should be defined as reader methods regardless of the type of the corresponding base-level method.

When a meta-object is specialized with respect to a reader method, the optimized method has the essentially same definition as the original base-level method, except for the indirect accesses to the instance variables (cf. the method `distance` in Figure 7). When it is specialized with respect to a writer method, on the other hand, the optimized method evidently contains extra operations. Although most of the operations in the optimized method are the same as the

[4] For convenience in executing the benchmark programs, instead of using a real meta-interpter we used a *fake evaluator* that directly executes the body of methods. This will be discussed in Section 5.

```
;;; a combined class of metaobject w.r.t. point
(define-class metaobject**point ()
  class methods state-vars state-values lock evaluator)

;;; constructor
(define (point x y)
  (metaobject**point
   (quote *metaobject*) (quote *methods*) (quote (x y))
   (make-cell (vector x y)) (make-lock) (quote *evaluator*)))

;;; reader method
(define-method metaobject**point (distance self)
  (begin (let* ((values0 (read-cell state-values))
                (x0 (vector-ref values0 0))
                (y0 (vector-ref values0 1))
                (g0 (square x0))
                (g1 (square y0)))
           (sqrt (+ g0 g1)))))

;;; writer method
(define-method metaobject**point (move! self dx dy)
  (begin (acquire! lock)
         (let* ((state-update-channel0 (make-channel))
                (values0 (read-cell state-values))
                (x0 (vector-ref values0 0))
                (y0 (vector-ref values0 1))
                (g0 (vector (+ x0 dx) (+ y0 dy))))
           (reply g0 state-update-channel0)
           (let ((new-state0 (touch state-update-channel0)))
             (update-cell! state-values new-state0)
             (release! lock)
             #t))))
```

Fig. 7. Result of optimization (the underlined expressions come from the base-level method).

operations performed in a writer method in Schematic, others are amenable to further optimization. For example, the newly created vector of instance variables `g0` is handed over by means of `reply` and `touch` operations in the same thread because our current partial evaluator regards those operations as mere "unknown" functions. An optimized method less extra operations could be produced by using partial evaluators for concurrent languages or by applying static analysis for concurrent programs[10, 15, 16] to the resulting code.

5 Performance Evaluation

To evaluate the efficiency of our partially evaluated meta-objects, we executed benchmark programs in the following three ways:

PE(partially evaluated): The default meta-object was partially evaluated with respect to each benchmark program, and the generated code was further compiled by Schematic. This showed the performance of our optimization framework.

INT(interpreted): The default meta-object was directly compiled by Schematic, and then the compiled code interpreted the benchmark programs. This showed the performance of naively implemented meta-objects.

NR(nonreflective): The benchmark programs were directly compiled by Schematic[5]. This showed the performance of nonreflective languages.

All programs were executed on Sun UltraEnterprise 4000 that had 1.2GB memory, 14 UltraSparc processors,[6] each operating at 167MHz, and was running SunOS 5.5.1.

The differences between the PE and INT performances show the amount of speedup gained by partial evaluation, while the differences between the PE and NR performance show the *residual overheads*—the overheads that the partial evaluator fails to eliminate.

The overheads solely caused by the meta-objects, were evaluated by executing the body expressions in PE and INT without meta-interpreters. For example, when a base-level program has an expression "`(distance p)`," then a meta-object looks up `distance` in its method table and extracts instance variables from `p`. However, the method body "`(sqrt (+ (square x) (square y)))`" should be executed directly. To do this, we generate a *fake evaluator* for each base-level class (Figure 8). Without fake evaluators, interpretive execution of method bodies would make an overwhelmingly large contribution to the execution time in INT. The fake evaluators are also useful for skipping over the partial evaluation of meta-interpreters whenever a base-level object uses only the default meta-interpreter.

[5] Our Schematic compiler has some overheads for concurrent execution; a sequential program (Richards) compiled by a sequential Scheme compiler (DEC Scheme-to-C) was faster than the one compiled by Schematic by a factor of 5.4.

[6] Though we used a multi-processor machine, the programs are executed on a single processor execution.

```
;;; Class definition
(define-class evaluator**point ())

;;; The method called by the meta-object.
(define-method evaluator**point (eval-begin self method-name exp env)
  (cond ((eq? method-name 'distance) ; for method distance
         (let ((x (lookup 'x env)) (y (lookup 'y env)))
           (sqrt (+ (square x) (square y)))))
        ((eq? method-name 'move!)    ; for method move!
         (let ((x (lookup 'x env)) (y (lookup 'y env))
               (dx (lookup 'dx env)) (dy (lookup 'dy env)))
           (let ((new-values (vector (+ x dx) (+ y dy))))
             (update self new-values))))))
```

Each clause of the `cond` form in `eval-begin` corresponds to the method of the base-level class `point`. A clause is selected by the argument `method-name`. The body part of a clause has the code for extracting the base-level arguments and instance variables and for the method body. A `become` form in the original program is converted into an invocation of the `update` method of the meta-object, which takes a vector of the updated instance variables as an argument.

Fig. 8. "Fake" evaluator for `point`.

5.1 Base-level Applications

The following three kinds of programs were executed as the base-level applications:

Null Readers **and** Null Writers**:** Elapsed time for 1,000,000 method invocations was measured by repeatedly calling a null method on an object. We tested objects with different numbers of instance variables (i) and tested methods with different numbers of arguments (j). The average time over some parameter combinations ($i \in \{0, 5, 10\}, j \in \{1, 5, 10\}$) are shown as a representative result.

Become**:** Elapsed time for 1,000,000 invocations of writer methods which update instance variables was measured by repeatedly calling a method that immediately performs `become`. We tested objects with different numbers of updated variables (k), and the average time over the parameter combinations $i = 10, j = 1, k \in \{1, 5, 10\}$ is shown as the representative result[7].

Richards**:** The Richards benchmark is an operating system simulation that is used as a nontrivial program in evaluating several object-oriented languages[4].

RNA**:** RNA is a parallel search program for predicting RNA secondary structures[22, 26]. This program uses an object to maintain and to share information the found answers among concurrently running threads.

[7] The combination of the values of i and j yields the worst result in Null Writers.

Table 1. Performance improvement and residual overheads.

benchmark applications	elapsed time (sec.) PE	INT	NR	improvement INT/PE	residual overheads PE/NR
Null Readers	3.2	107.7	2.3	33.6	1.4
Null Writers	40.7	190.8	16.9	4.7	2.4
Become	46.6	272.8	15.7	5.9	3.0
(w/manual opt.)	(21.3)			(12.8)	(1.4)
Richards	20.7	140.7	9.4	6.8	2.1
RNA	1.7	53.3	1.6	30.8	1.1

Since Richards and RNA use both functions and methods, their executions show how the efficiency of the meta-objects affects overall execution speed in realistic applications.

The results are summarized in Table 1. As the "improvement" column shows, the programs in PE are more than four times faster than the ones in INT. This improvement is significant even in realistic applications such as Richards and RNA, whose speeds are increased by factors of 6.8 and 30.8, respectively.

As the "residual overheads" column shows, the programs in PE are slower than the ones in NR by factors of 1.1–3.0. These overheads are mainly due to the limitations of current partial evaluators, as we have pointed out in Section 4.3. In fact, when we further optimized the partially evaluated meta-objects for Become by hand—eliminating obvious channel communications, etc.—the average factor by which programs are slowed because of residual overheads was reduced to 1.4.

5.2 Performance of Customized Meta-objects

The above benchmark programs were executed under the default meta-objects, but of more practical interest is the efficiency of *customized* meta-objects. The next benchmark program was a bounded-buffer that uses the guarded method invocation mechanism, which is implemented by a customized meta-object. Since the guarded methods are not directly supported in Schematic, we simulated them by user-level programming, in which objects are programmed to check the guard conditions and to suspend/continue their invocation requests. The programs are described in Appendix A.

Table 2 shows the elapsed time for 1,000 read/write operations from/to a bounded buffer whose size is 10. The PE buffer shows almost the same efficiency as does the NR one. This result could be understood as that the the overheads caused by frequent method invocations in NR cancel out the residual overheads in the PE buffer. The NR buffer uses three methods in order to represent a guarded method. On the other hand, the PE buffer uses only one because the partial evaluator successfully inlines the methods of the meta-object that deal with the guarded methods.

Table 2. Performance of bounded buffer with guarded methods.

	elapsed time (sec.)			improvement	residual overheads
	PE	INT	NR	INT/PE	PE/NR
Bounded Buffer	3.94	4.46	3.96	1.13	0.99

The partially evaluated meta-objects are approximately 10 percent faster than the interpreted ones (INT). This improvement is less significant than that observed with the previous benchmarks. We conjecture that this is because each of these benchmark programs requires a large number of context switches, and context-switching is expensive in the current Schematic implementation. The time spent for context-switching is thus so great that the efficiency differences between the three programs are relatively small.

6 Related Work

In CLOS Meta-Object Protocols (MOP), meta-level methods are split into functional and procedural ones for caching (or *memoization*)[13, 14]. This splitting approach in principle similar to our meta-object design, but the memoization technique requires more careful protocol design because the unit of specialization is function. Thus the "functional" methods cannot include operations that touch dynamic data. On the other hand, such operations can be written in our reader methods, since the partial evaluator automatically residualizes them.

Another approach to efficient reflective systems is use compile-time MOP[5, 11], in which efficiency is guaranteed by allowing the meta-level computation to be performed only at the compile-time. This means that the changes in the run-time behavior of the base-level program should be made by writing translation rules that convert the program into one containing the expected behavior. This task could be burdensome if the modification involved run-time representation of an object, because no run-time meta-objects are available in compile-time MOPs.

7 Conclusion

We have described a method for designing meta-objects in the reflective language ABCL/R3 and presented a framework for their optimization using partial evaluation. In the meta-object's description, operations that are state-related are separated from operations that are not, and it is this separation that makes partial evaluation effective. The meta-objects and their reader methods are translated into records and functions in Scheme, and they are then optimized by using a Scheme partial evaluator. The optimized code is a combination of the base-level and meta-level programs, a combination from wihch most interpretive operations at the meta-level (such as the method dispatch and the manipulation of

the environment) have been removed. Effectiveness of this optimization framework is shown by benchmark programs in which the partially evaluated objects run significantly faster than the interpretive meta-objects. Moreover, the partial evaluation lets a program with customized meta-objects run as efficiently as an equivalent nonreflective program.

Acknowledgments

The earlier version of ABCL/R3 was designed in collaboration with Satoshi Matsuoka, and we would like to express our thanks to him. We would also like to thank Kenjiro Taura, Kenichi Asai, and Ken Wakita for their valuable comments and for their technical help to run the Schematic compiler and the Scheme partial evaluator.

References

1. Andersen, L. O.: Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen (1994). (DIKU Report 94/19)
2. Asai, K., Masuhara, H., Matsuoka, S., Yonezawa, A.: Partial Evaluation as a Compiler for Reflective Languages. Technical Report 95–10, Department of Information Science, University of Tokyo (1995)
3. Asai, K., Masuhara, H., Yonezawa, A.: Partial Evaluation of Call-by-value lambda-calculus with Side-effects. In Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97), SIGPLAN Notices, Vol. 32, No. 12. ACM (1997) 12–21
4. Chambers, C., Ungar, D., Lee, E.: An Efficient Implementation of SELF, a Dynamically-Type Object-Oriented Language Based on Prototypes. In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'89), SIGPLAN Notices, Vol. 24, No. 10. ACM (1989) 49–70
5. Chiba, S.: A Metaobject Protocol for C++. In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95), SIGPLAN Notices, Vol. 30, No. 10. ACM (1995) 285–299
6. Chiba, S. Masuda, T.: Designing an Extensible Distributed Language with a Meta-Level Architecture. In European Conference on Object-Oriented Programming (ECOOP'93), Lecture Notes in Computer Science, Vol. 707. Springer-Verlag (1993) 482-501.
7. Futamura, Y.: Partial Evaluation of Computation Process—an Approach to a Compiler-compiler. Systems, Computers, Controls, Vol. 2, No. 5 (1971) 45–50
8. Gengler, M. Martel, M.: Self-applicable partial evaluation for the pi-calculus. In Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97), SIGPLAN Notices, Vol. 32, No. 12. ACM (1997)
9. Hosoya, H., Kobayashi, N., Yonezawa, A.: Partial Evaluation Scheme for Concurrent Languages and Its Correctness. Euro-Par'96 Parallel Processing, Lecture Notes in Computer Science, Vol. 1123. Springer-Verlag (1996) 625–632
10. Igarashi, A. Kobayashi, N.: Type-Based Analysis of Usage of Communication Channels for Concurrent Programming Languages. In International Static Analysis Symposium (SAS'97), Lecture Notes in Computer Science, Vol. 1302. Springer-Verlag, (1997) 187–201

11. Ishikawa, Y., Hori, A., Sato, M., Matsuda, M., Nolte, J., Tezuka, H., Konaka, H., Maeda, M., Kubota, K.: Design and Implementation of Metalevel Architecture in C++: MPC++ Approach. In Reflection Symposium'96 (1996) 153–166
12. Jones, N. D., Gomard, C. K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall (1993)
13. Kiczales, G., Rivières, J.des , Bobrow, D. G.: The Art of the Metaobject Protocol. MIT Press, Cambridge, MA (1991)
14. Kiczales, G. Rodriguez, L.: Efficient Method Dispatch in PCL. In LISP and Functional Programming (LFP'90), ACM (1990) 99–105
15. Kobayashi, N., Nakade, M., Yonezawa, A.: Static Analysis of Communication for Asynchronous Concurrent Programming Languages. In International Static Analysis Symposium (SAS'95), Lecture Notes in Computer Science, Vol. 983. Springer-Verlag (1995) 225–242
16. Kobayashi, N., Pierce, B. C., Turner, D. N.: Linearity and the Pi-Calculus. In Principles of Programming Languages (POPL'96) (1996) 358–371
17. Maes, P.: Concepts and Experiments in Computational Reflection. In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87), SIGPLAN Notices Vol. 22, No. 12. ACM (1987) 147–155
18. Marinescu, M. Goldberg, B.: Partial Evaluation Techniques for Concurrent Programs. In Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97), SIGPLAN Notices, Vol. 32, No. 12. ACM (1997) 47–62
19. Masuhara, H., Matsuoka, S., Asai, K., Yonezawa, A.: Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Languages Using Partial Evaluation. In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95), SIGPLAN Notices, Vol. 30, No. 10. ACM (1995) 300–315
20. Masuhara, H., Matsuoka, S., Watanabe, T., Yonezawa, A.: Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently. In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'92), SIGPLAN Notices, Vol. 27, No. 10. ACM (1992) 127–145
21. Masuhara, H., Matsuoka, S., Yonezawa, A.: Implementing Parallel Language Constructs Using a Reflective Object-Oriented Language. In Reflection Symposium'96, San Francisco, CA. (1996) 79–91
22. Nakaya, A., Yamamoto, K., Yonezawa, A.: RNA Secondary Structure Prediction Using Highly Parallel Computers. Compt. Appl. Biosci. **11** (1995) 685–692
23. Oyama, Y., Taura, K., Yonezawa, A.: An Efficient Compilation Framework for Languages Based on a Concurrent Process Calculus. In Euro-Par '97 Object-Oriented Programming, Lecture Notes in Computer Science, Vol. 1300. Springer-Verlag, (1997)
24. Smith, B. C.: Reflection and Semantics in Lisp. In Principles of Programming Languages (POPL'84), ACM (1984) 23–35
25. Sun Microsystems, : Java(TM) Core Reflection: API and Specification, (1997)
26. Taura, K.: Efficient and Reusable Implementation of Fine-Grain Multithreading and Garbage Collection on Distributed-Memory Parallel Computers. PhD thesis, Department of Information Science, University of Tokyo (1997).
27. Taura, K. Yonezawa, A.: Schematic: A Concurrent Object-Oriented Extension to Scheme. In Object-Based Parallel and Distributed Computation, Lecture Notes in Computer Science, Vol. 1107. Springer-Verlag, (1996) 59–82
28. Watanabe, T. Yonezawa, A.: Reflection in an Object-Oriented Concurrent Language. In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'88), SIGPLAN Notices, Vol. 23, No. 11. ACM (1988) 306–315.

29. Yonezawa, A. (ed.): ABCL: An Object-Oriented Concurrent System. MIT Press, Cambridge, MA (1990)

A Programs Using Guarded Methods

A.1 Base-level Program

A base-level object that uses the guarded method mechanism has an optional form "`(:metaclass ...)`" in the class declaration, and has an expression "`(:guard ...)`" in each guarded method. The following program is the definition of the bounded buffer used in Section 5.2:

```
(define-class bb () size elements
  (:metaclass guard-meta))

(define-method! bb (put! self item)
  (:guard (< (length elements) size)) ; guard expression
  (become self :elements (append elements (list item))))
```

A.2 Meta-level Program

We define the class `guard-meta`, as a subclass of metaobject, at the meta-level.

```
(define-class guard-meta (metaobject) ; a subclass of metaobject
  (guard (make-guard)))               ; scheduler
```

In the additional instance variable `guard`, each instance of `guard-meta` has a scheduler, which is a user-defined meta-level object. We also override the following two methods of `guard-meta`:

```
(define-method guard-meta (receive self mes &reply-to mresult)
  (let* ((selector (message-selector mes))
         (method (find-method methods selector))
         (guard-exp (cdr (method-find-option method ':guard))))
    (register guard
              (lambda ()
                (let* ((env (make-env self (formals method) mes))
                       (result (eval evaluator guard-exp env)))
                  (if result
                      (reply (accept-W self mes) mresult))
                  result))))) ; result of guard expression

(define-method guard-meta (accept-W self mes)
  (let ((r (make-channel)))
    (let ((result (accept self mes r)))
      (update self (touch r))
      (notify guard)
      result))) ; result of method body
```

The method `receive` registers a closure to `guard`. The closure, when activated by the scheduler, evaluates a guard expression and then invokes `accept-W` if the guard expression returns be true. The method `accept-W`, evaluates the method body, as `accept-W` of the class `metaobject` does, and also notifies `guard` at the end of the evaluation.

A.3 Optimized Program

From the base-level and the meta-level programs, our optimization framework generates the following combined program. The meta-level operations for guarded methods, which are defined in the methods `receive` and `accept-W` of `guard-meta`, are embedded in the method `put!` of the optimized class.

```
(define-class guard-meta**bb ()
  class methods state-vars state-values lock evaluator
  (guard (make-guard)))

(define-method guard-meta**bb (put! self item &reply-to mresult0)
  (let ((c0 (lambda ()
              ;; evaluation of guard expression
              (let* ((values0   (read-cell state-values))
                     (size0     (vector-ref values0 0))
                     (elements0 (vector-ref values0 1))
                     (result0   (< (length elements0) size0)))
               (if result0
                 ;; execution of method body
                 (let* ((state-update-ch0 (make-channel))
                        (values1   (read-cell state-values))
                        (size1     (vector-ref values1 0))
                        (elements1 (vector-ref values1 1)))
                   (reply (vector size1
                                  (append elements1 (list item)))
                          state-update-ch0)
                   (let ((new-state0 (touch state-update-ch0)))
                     (update-cell! state-values new-state0)
                     (notify guard)
                     (reply self mresult0))) ; result of method body
                 #f)
               result0)))) ; result of guard expression
    (register guard c0)))
```

A.4 Nonreflective Program

Instead of using customized meta-objects, we can manually rewrite programs that have the same functionality to the ones using guarded methods. One of the simplest approach is to split each guarded into three actual methods: an entry

method, a guard method, and a body method. The following definitions are a manually rewritten bounded buffer:

```
(define-class bb ()     ; nonreflective version
  size elements (guard (make-guard))

(define-method bb (put! self item &reply-to r)
  (let ((c (lambda ()
             (let ((guard-result (put!-guard self item)))
               (if guard-result
                   (reply (put!-body self item) r))
               guard-result))))
    (register guard c)))

(define-method bb (put!-guard self item)
  (< (length elements) size))            ; guard expression

(define-method! bb (put!-body self item)
  (become (begin (notify guard)          ; notification
                 self)
          :elements (append elements (list item))))
```

The class definition has an additional instance variable **guard** for the scheduler. The method `put!` is an entry method that creates and registers a closure to the scheduler. The method `put!-guard` is the guard method, and `put!-body` is the body method. They are invoked from the closure created in `put!`.