
An Object-Oriented Concurrent Reflective Language ABCL/R3

Its Meta-level Design and Efficient Implementation Techniques

Hidehiko Masuhara¹ Akinori Yonezawa²

¹*Department of Graphics and Computer Science,
Graduate School of Arts and Sciences, University of Tokyo
Tokyo, 153-8902 JAPAN
masuhara@acm.org*

²*Department of Information Science, University of Tokyo
yonezawa@is.s.u-tokyo.ac.jp*

ABSTRACT. *This article presents the design principles and efficient implementation techniques for ABCL/R3, an object-oriented concurrent reflective language. One of the most distinguished features of ABCL/R3 is compilation techniques using partial evaluation, which effectively remove interpretation from meta-level programs. The meta-level objects are designed so that they can be partially evaluated in an effective manner. Benchmark programs show that our compilation frameworks make object execution drastically faster than interpreter-based implementations, and achieves performance close to nonreflective compilers.*

RÉSUMÉ. *Cet article présente les principes orientateurs et les techniques utilisées pour l'implantation efficace d'ABCL/R3, un langage de programmation concurrent orienté-objet avec réflexion. L'une des caractéristiques majeures d'ABCL/R3 est sa compilation avec évaluation partielle, qui élimine l'interprétation pour les programmes du méta-niveau. Des programmes de test ont montré que notre approche exécute les objets incomparablement plus vite qu'une implantation naïve, avec des performances proches des compilateurs sans réflexion.*

KEY WORDS: *Reflection, meta-objects, meta-level architecture, object-oriented concurrent languages, partial evaluation, specialization, and Futamura projections.*

MOTS-CLÉS: *Réflexion, méta-objets, achitecture avec méta-niveau, langages concurrents orientés-objet, évaluation partielle, spécialisation, et projections de Futamura.*

1. Introduction

Over the past decade, *reflection* has been recognized as useful for parallel and distributed programming[CHI 93A, ISH 96, LOP 96, OKA 94, ROD 92, WAT 88] because the application programmer can extend a language through an abstract model of the language. One of the most serious problems with reflection is that the model of a language requires interpretive execution. Some naïve implementations, which run full-fledged interpreters, degrade run-time performance by a factor of 10 to 1000 over the nonreflective languages with optimizing compilers.

Previous performance improvement techniques usually works for less flexible reflective models, and still leave considerable amount of overhead. In those techniques, part of a program is interpreted for extensibility, and the rest is executed by using compiled code. It improves the performance, but still has overhead of a factor of 10 over nonreflective languages. Moreover, such a hybrid interpreter weakens the extensibility and clarity of the meta-level architecture.

Partial evaluation[FUT 71, JON 93] can be a means for radically reducing the run-time overhead from reflective programs, and, at the same time, it provides less restrictive reflective models. This is because partial evaluation can compile a program that is executed under a user-defined interpreter, and generate a specialized and more efficient program before its execution.

However, application of partial evaluation to the meta-level of existing object-oriented concurrent reflective languages causes several technical difficulties. The meta-level architecture should be designed so as to suit partial evaluation. Partial evaluation techniques themselves should also be improved so that they can properly handle our target languages—object-oriented concurrent languages. Also, language mechanisms that promotes meta-level programmability (*e.g.*, inheritance, delegation, etc.) could be incorporated without degrading the run-time performance, if those mechanisms can be optimized by partial evaluation.

In the following sections, we present design and efficient implementation techniques of ABCL/R3's meta-level. As this paper aims at introducing an overview of these topics, interested readers should consult [MAS 95, MAS 98, MAS 99] for more detailed discussions.

2. Overview of ABCL/R3

The meta-level architecture of ABCL/R3 is based on that of ABCL/R[WAT 88] and ABCL/R2[MAT 91, MAS 92]. Figure 1 illustrates an overview of ABCL/R3. For each base-level object, there is a *meta-object* at the meta-level, which explicitly contains the names and values of instance variables, list of methods, and other state for mutual exclusion. It also defines how a method invocation request is processed (*i.e.*, policy for mutual exclusion and method lookup). In a broad sense, a meta-object refers to any object at the meta-level. In this paper, however, we call such an object a *meta-level object*. Evaluation of expressions in a base-level method is defined by meta-level objects called *meta-interpreters*.

In our partial evaluation framework, a method of a meta-object that processes a method invocation request is specialized with respect to each base-level method name. Meta-interpreters are also specialized with respect to the body of a selected method in this process, since they are called from a method of the meta-object. In the following sections, however, we separately discuss meta-objects and meta-interpreters because they exhibit different characteristics in the context of partial evaluation. In fact, when we partially evaluate a meta-object that has a default meta-interpreter, we separate the partial evaluation of the meta-object and the meta-interpreter by replacing the meta-interpreter with a *fake evaluator* [MAS 98].

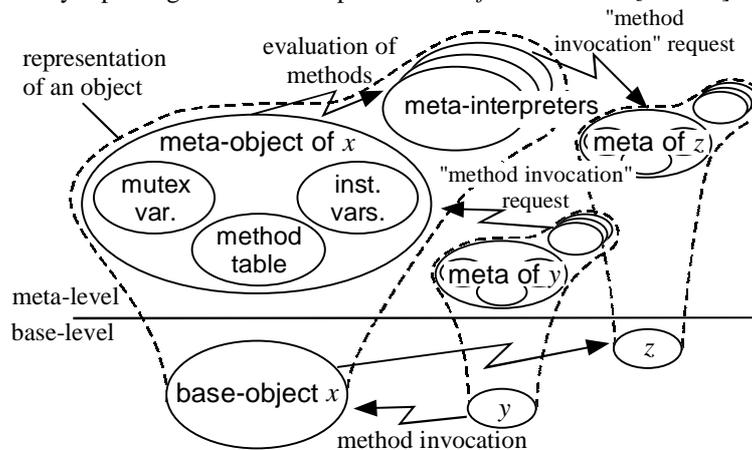


Figure 1. Objects in ABCL/R3.

3. Meta-interpreters

A *meta-interpreter* in ABCL/R3 is an object that evaluates base-level expressions as traditional Lisp meta-circular interpreters do. It is a major source of extensibility, yet it is that of inefficiency of reflective languages. We first discuss the design of the meta-interpreters in ABCL/R3, which employ fine-grained protocols for maximizing flexibility. We then present efficient implementation techniques (using partial evaluation) and performance evaluation.

3.1 Meta-interpreter Design

We first describe our proposed meta-interpreter design of an object-oriented concurrent language ABCL/R3, which has the following characteristics: (1) the full-fledged meta-interpreters offering a clear model for customization, (2) the fine-grained protocols and the delegation mechanism facilitating modular and scope-

controlled meta-level programming, and (3) the *reflective annotations* realizing separation and cooperation of base- and meta-level programs.

On designing meta-interpreters, the following issues should be considered:

- Among various reflective capabilities, meta-interpreters are suitable abstraction to customize existing language constructs and to provide novel ones, as many previous studies show[SMI 84, DR 88, SIM 92]. This is also true, or even more important, for concurrent programming, as it often requires various language constructs to adapt applications for a special hardware, to implement application specific optimizations, etc.[WAT 88, MAT 91, MAS 92, OKA 94]
- Mechanisms that support easier meta-level programming should be provided as long as they do not pose critical performance penalty. Specifically, modularity—the ability to compose meta-level programs, and scope-controllability—the ability to limit the effect of a meta-level program to specific part of base-level programs, are the concerns.

The solutions in this study are: (1) full-fledged interpreters with fine grained protocols by assuming partial evaluation (2) delegation for modularity. The delegation relationship can be extended (pseudo-)dynamically, which is useful to follow the scope of base-level programs[MAS 96]. The major features are as follows:

- **Full-fledged meta-interpreter**, which interprets every expression in base-level programs, provides clear view of the “behavior” of programs. Such an otherwise sluggish interpreter can be efficiently implemented by our compilation framework.
- **Fine-grained methods**. A meta-interpreter defines the semantics of base-level programs in a similar manner to the traditional Lisp meta-circular interpreters. A prominent feature is that it defines by using a number of fine-grained methods, so that the user can reuse many of existing methods for his/her customizations. The methods of the *primary* (i.e., the default) meta-interpreter include **eval-entry** as an entry point, **eval** as a dispatcher, **eval-var** for variable references, etc.
- **Delegation mechanism**. ABCL/R3 has a delegation mechanism to customize meta-interpreters, instead of an inheritance mechanism. This enables the user to define modular customizations, and to compose customizations dynamically.
- **Reflective annotations** serve as flexible programmable directives to the meta-level from the base-level programs. An annotation to an expression is written in curly braces immediately after the expression, and consists of a keyword and arbitrary argument expressions. Our annotations are called *reflective annotations*, as their interpretation can be customized by overriding the method **eval-annotation** at the meta-level.

3.2 Compiling Away the Meta-interpreters

In ABCL/R3, base-level programs, whose behavior is defined by full-fledged meta-interpreters, are compiled by using partial evaluation, or more specifically, the technique known as the first Futamura projection[FUT 71]. Although the first Futamura projection has been known for long years, it is not trivial to apply to our meta-interpreters due to concurrency and dynamic dispatching at the meta-level.

In practice, existing partial evaluators do not allow us to directly deal with the meta-circular interpreters written in object-oriented concurrent languages. Here we explain the underlying problems and our proposed solutions.

- **Concurrent meta-system.** Since the meta-system of ABCL/R3 consists of concurrent objects, we have to partially evaluate concurrent objects if we target to the *entire* meta-system. To the best of our knowledge, no partial evaluation studies that deal with concurrent objects.
Solution: Although the meta-level of ABCL/R3 consists of a number of concurrent objects, meta-interpreters for each meta-object do not interfere with each other. Therefore, when we focus on meta-interpreters, we can use a partial evaluator for sequential languages. For each meta-object, the system converts associated meta-interpreter definitions into functions, and separately applies partial evaluation to them. Interactions to other objects are regarded as side-effects during partial evaluation.
- **Dynamic dispatching.** The delegation mechanism, which dynamically determines an appropriate definition for each method invocation, may cause a partial evaluator to yield uncompiled results. This is because, for each method invocation, if the body of the invoked method were not determined at the partial evaluation time, the partial evaluator could not perform further specialization.
Solution: Our system restricts delegation chains to the ones whose elements are known at partial evaluation time so that method dispatching can be resolved. Otherwise, it reports an error.
- **Side-effect in programs.** The meta-interpreter definitions have side-effecting operations, with which simple partial evaluators may incorrectly translate programs. They include I/O type operations for message passing and assignments for base-level assignments.
Solution: For I/O type operations, we propose a partial evaluation mechanism called *preaction* for preserving the traces of I/O operations. For assignments, we simply avoid the direct use of assignments in meta-interpreters. In our meta-level design, instance variables of an object are kept in a meta-level object `state-values` (see Section 4.1). At the beginning of a base-level method evaluation, the meta-object extracts the values from `state-values`, and passes them upon a meta-interpreter. Assignments to instance variables are represented as a method invocation to `state-values`, which is treated as an I/O operation. Although the method invocation to `state-values` should be performed at run-time, the number

of the invocation is at most one per base-level method, thank to the “become” syntax[TAU 96].

We have developed a scheme to compile a base-level method that is executed by customized meta-interpreters. The compilation scheme consists of: (a) *preprocessing*: the meta-interpreter definitions (including both default and user defined ones) are converted into a set of Scheme functions; (b) *partial evaluation*: the converted functions are specialized with respect to each base-level method, yielding a set of residual expressions, by using a Scheme partial evaluator[ASA 97]; (c) *postprocessing*: the residual expressions are further converted into a Schematic program by adding class/method interface (this process will be explained in Section 4); and (d) *back-end compilation*: the generated program is compiled into an executable code by the Schematic compiler. More detailed discussion on this compilation process can be found in [MAS 95].

3.3 Performance Evaluation of Meta-interpreters

We executed benchmark programs to evaluate the performance of the proposed compilation framework. To illustrate performance improvement from a naive implementation, and ‘residual’ overhead of meta-level interpretation, the following three executions are compared: (**PE**: partially evaluated) The partially evaluated meta-interpreters were compiled by a back-end compiler. This showed the performance of our optimization framework. (**INT**: interpreted) The default meta-interpreter was directly compiled by a nonreflective compiler, and then the compiled code interpreted the benchmark programs. This showed the performance of naively implemented meta-level objects. (**NR**: nonreflective) The benchmark programs were directly compiled by a back-end compiler. This showed the performance of nonreflective languages.

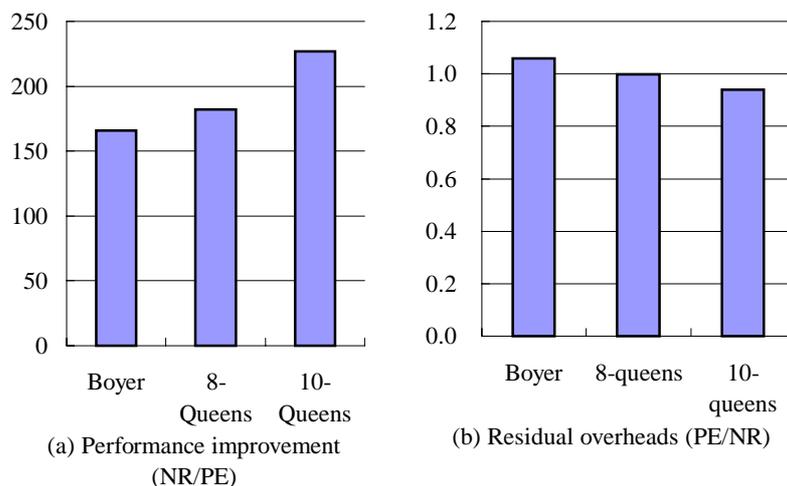


Figure 2. *Meta-interpreter performance*

The evaluation in this section focuses on the performance of meta-interpreters. The rest part of a meta-object is executed without interpretation. As a back-end compiler, we used CMU Common Lisp 17e. The benchmark programs are the Boyer[GAB 85] and n-queens problem. All programs are executed on a SUN Sparcstation 10 workstation (SuperSparc 50MHz, 128MB memory).

Figure 2(a) shows the ratios of the running times of **INT**'s executions to the **PE**'s executions, and Figure 2(b) shows the ratios of the running times of **PE**'s executions to the **NR**'s. We observe that (1) the proposed compilation scheme exhibits equivalent performance¹ to traditional (*i.e.*, **NR**) compilers, and (2) compared to naive interpretation, our compilation scheme improves performance more than 100-fold.

4. Meta-objects

4.1 Partial Evaluation Conscious Design

A *meta-object* in ABCL/R3 is a meta-level object that defines behavior of the respective base-level object *except for interpretation of expressions in base-level methods*, which is defined by a *meta-interpreter*. More specifically, a meta-object contains a reference to the class, a list of methods, and a list of instance variable names of the base-level object. It also defines how to find an appropriate method, mutually exclude multiple invocation requests, and update instance variables.

Similar to meta-interpreters, meta-objects provide a mechanism to customize behavior of base-level objects, which is implemented by sluggish interpretive execution. Therefore, partial evaluation of meta-programs with respect to base-level programs is also a promising technique to eliminate the overheads.

Unfortunately, naive application of partial evaluators to meta-object definitions does not yield effective result. Although a number of studies on partially evaluating interpreters have been made[FUT 71, JON 93], meta-objects, as a target of partial evaluation, exhibit difficulties that do not appear in those studies. The difficulties are that (1) the meta-objects in existing reflective languages, which are designed as *state-transition machines*, are 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.

Our proposed meta-object design can be effectively optimized by partial evaluation[MAS 98]. The key idea is to separate state-related operations from the other operations using the reader and writer methods of Schematic[TAU 96]. Below is the definition of a new meta-object in the Schematic's syntax:

¹ The **PE** version runs the 10-queens program slightly faster than the **NR** version does. We conjecture that the program yielded by the partial evaluator happened to unroll loops in the base-level program, due to the loose termination policy in our partial evaluator.

```

;;; 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 state
    (acquire! lock)                       ; mutual exclusion begins
    (let ((result (accept self messages c)))
      (cell-set! state-values (touch c)) ; update inst. vars.
      (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-entry evaluator (method-body m)
              env update-channel))))

```

This 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, which are mutable, are packed in the mutable vector object `state-values`, and accesses to `state-values` are effected through 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 the variables that represent current state of an object. 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. (1) Thanks to the execution model of Schematic[TAU 96], 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 and code explosion during specialization.

4.2 Specialization of Meta-objects

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-effecting operations. In this section, we describe an optimization framework for our meta-objects by using partial evaluation.

The remaining problem is that there are no partial evaluators appropriate for our purpose because the meta-object is written in an object-oriented *concurrent* language. Although there are studies on partial evaluators for concurrent languages [GEN 97, HOS 96, MAR 97], they focus on concurrency and pay little attention to the support of features crucial to sequential languages, such as function closures and data structures.

However, thanks to our new meta-object design (see Section 4.1), the primary behavior of a meta-object is defined as a single thread of execution. Therefore, our solution is to translate methods of a meta-object into a sequential language and to 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. A meta-object method may contain expressions that use concurrency primitives (*e.g.*, future and touch) because our concurrency model is based on Schematic [TAU 96]. Such an expression is translated into an invocation of a function that is *unknown* to the partial evaluator at pre-processing, and then translated back into an expression that uses a concurrency primitive at post-processing.

Since the methods of meta-objects exhibit almost sequential 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 primitive functions.

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.

The optimization framework consists of three steps:

- **Preprocessing.** Meta-object definitions are translated into a Scheme program so that a Scheme partial evaluator can process them. No translations are needed for the base-level definitions, since they are used as data for the meta-level program. Base-level functions are simply copied to the resulting Schematic program.
- **Partial Evaluation.** We partially evaluate the meta-level program for each *base-level method invocation*. To do so, we generate a *specialization point function* for each base-level method invocation, and let a partial evaluator process it. An online partial evaluator for Scheme[ASA 97] specializes not only the methods of `metaobj`, but also those of `evaluator`, as is shown in the previous section.
- **Postprocessing.** The final step is to translate the results of partial evaluation in Scheme back into concurrent objects in Schematic. This process includes generating appropriate class declaration, constructor function, and signatures for methods.

For example, assume there is a base-level program that represents a two-dimensional point class with a reader method `distance` and a writer method `move!`:

```
;;; 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)))
```

When we specialize our meta-object definition (see Section 4.1) with respect to above `point` class, the following definitions, which include combined class declaration, constructor functions, and methods, will be generated:

```
;;; 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 (cell-ref 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 (cell-ref 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)))
        (cell-set! state-values new-state0)
        (release! lock)
        #t))))

```

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` of `point`). 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 operations performed in a writer method in Schematic, others are amenable to further optimization, such as `reply` and `touch` operations for updating `state-values`.

4.3 Performance Evaluation of Meta-objects

The proposed optimization framework of meta-objects was also evaluated by executing benchmark programs in **INT**, **PE**, and **NR** as we did in the previous section. In this case, we focus on the performance of meta-objects other than meta-interpreters. Therefore, body of base-level methods is executed without interpretation. The execution platform was Sun UltraEnterprise 4000 that had 1.2GB memory, 14 UltraSparc processors, each operating at 167MHz, and was running SunOS 5.5.1, and Schematic[TAU 96] is used as the back-end compiler.

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

- **Null Reader and Null Writer:** Elapsed time for 1,000,000 method invocations was measured by repeatedly calling a null reader method and a null writer method on an object, respectively.
- **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`.
- **Richards:** It is an operating system simulation that is used as a nontrivial program for evaluating performance of several object-oriented languages[CHA 89].

- **RNA** is a parallel search program for predicting RNA secondary structures[NAK 95]. This program uses an object to maintain and to share information the found answers among concurrently running threads.

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.

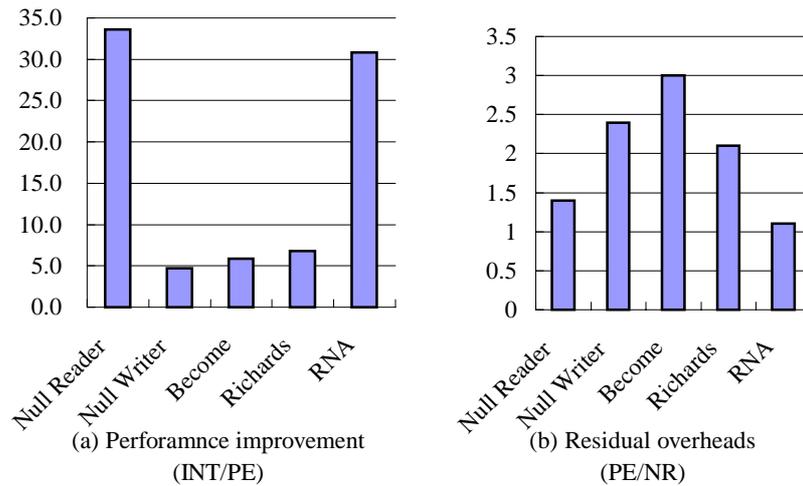


Figure 3. *Meta-object performance*

The results are summarized in Figure 3. As this shows, our optimization framework (**PE**) substantially improves the performance by the factor of more than four, while it is still slower than **NR** by factors of 1.1–3.0. We presume that these overheads are mainly due to the limitations of current partial evaluators.

5. Related Work

In CLOS Meta-Object Protocols (MOP), meta-level methods are split into functional and procedural ones for caching (or *memoization*)[KIC 90]. This splitting approach is in principle similar to our meta-object design, but the memoization technique requires more careful protocol design because the unit of specialization is a function. Thus the “functional” methods cannot include operations that touch dynamic data. Our reader methods, on the other hand, can include such operations, since the partial evaluator automatically residualizes them.

There are several studies that specialize meta-objects with respect to base-level programs. For example, an implementation of the early version of OpenC++[CHI

93B]² specializes meta-objects. Although the technique is effective for some cases, it is limited since it is based on “idiom recognition.” Our proposed techniques, which are based on partial evaluation, are effective to a larger variety of meta-objects.

6. Conclusion

This paper presents an overview of ABCL/R3, an object-oriented concurrent reflective language. One of the primary concerns in ABCL/R3’s language design is meta-level programmability and efficient execution. As for programmability, meta-interpreters are designed with the delegation mechanism and reflective annotations. As for efficient execution, we used partial evaluation techniques to specialize meta-level definitions to base-level programs. The frameworks to apply partial evaluation basically translate meta-level object definitions into functions and structured data in a sequential language, and apply partial evaluation thereupon.

The effectiveness of the optimization frameworks is demonstrated by several benchmarks, which indicate that our optimization eliminates most of the interpretation overhead.

Acknowledgments. We would like to thank Satoshi Matsuoka and Kenichi Asai, who contributed to the early stage of the work. We would also like to thank the anonymous reviewer for useful comments. Finally, we would like to thank Jacques Garrigue for translating the abstract of the paper into French.

References

- [ASA 97] ASAI, K., MASUHARA, H., and YONEZAWA, A. “Partial evaluation of call-by-value lambda-calculus with side-effects”, *Proceedings of Partial Evaluation and Semantics-Based Program Manipulation (PEPM’97) (SIGPLAN Notices Vol.32, No.12)*, p. 12–21, Amsterdam, June 1997.
- [CHA 89] CHAMBERS, C., UNGAR, D., and LEE, E. “An efficient implementation of SELF, a dynamically-type object-oriented language based on prototypes”, *Proceedings of OOPSLA’89 (SIGPLAN Notices Vol.24, No.10)*, p. 49–70, New Orleans, LA, October 1989. ACM.
- [CHI 93A] CHIBA, S. and MASUDA, T. “Designing an extensible distributed language with a meta-level architecture”, *Proceedings of European Conference on Object-Oriented Programming (ECOOP’93)*, 1993. LNCS 707.

² Unlike the later version of OpenC++, which is based on compile-time reflection, the early version has run-time meta-objects. Similar to CLOS MOP, meta-objects in the version interprets narrowed set of events including a method invocation, an object creation and an instance (member) variable access.

- [CHI 93B] CHIBA, S. and MASUDA, T. “Open C++ and its optimization”, *Proceedings of OOPSLA'93 Workshop on Object-Oriented Reflection and Metalevel Architectures*, Washington, D.C., September 1993.
- [DR 88] RIVIÉRES, D. J. “Control-related meta-level facilities in LISP”, *Meta-Level Architectures and Reflection*, Elsevier Science, p. 101–110, 1988.
- [FUT 71] FUTAMURA, Y. Partial evaluation of computation process—an approach to a compiler-compiler. *Systems, Computers, Controls*, Vol. 2, No. 5, p. 45–50, 1971.
- [GAB 85] GABRIEL, R. P. *Performance and Evaluation of Lisp Systems*. MIT Press, 1985.
- [GEN 97] GENGLER, M. and MARTEL, M. “Self-applicable partial evaluation for the pi-calculus”, *Proceedings of Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97) (SIGPLAN Notices Vol.32, No.12)*, Amsterdam, June 1997.
- [HOS 96] HOSOYA, H., KOBAYASHI, N., and YONEZAWA, A. “Partial evaluation scheme for concurrent languages and its correctness”, *Euro-Par'96 Parallel Processing, No. 1123 in Lecture Notes in Computer Science*, p. 625–632, 1996.
- [ISH 96] ISHIKAWA, Y., HORI, A., SATO, M., MATSUDA, M., NOLTE, J., TEZUKA, H., KONAKA, H., MAEDA, M., and KUBOTA, K. “Design and implementation of metalevel architecture in C++: MPC++ approach”, *Reflection Symposium'96*, p. 153–166, San Francisco, CA, April 1996.
- [JON 93] JONES, N. D., GOMARD, C. K., and SESTOFT, P. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
- [KIC 90] KICZALES, G. and RODRIGUEZ, L. “Efficient method dispatch in PCL”, *Proceedings of Conference on LISP and Functional Programming*, p. 99–105, Nice, France, June 1990.
- [LOP 96] LOPES, C. V. “Adaptive parameter passing”, *Object Technologies for Advance Software (ISOTAS'96), No. 1049 in Lecture Notes in Computer Science*, p. 118–136, Springer-Verlag, 1996.
- [MAR 97] MARINESCU, M. and GOLDBERG, B. “Partial evaluation techniques for concurrent programs”, *Proceedings of Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97) (SIGPLAN Notices Vol.32, No.12)*, p. 47–62, Amsterdam, June 1997.
- [MAS 92] MASUHARA, H., MATSUOKA, S., WATANABE, T., and YONEZAWA, A. “Object-oriented concurrent reflective languages can be implemented efficiently”, *Proceedings of OOPSLA'92 (SIGPLAN Notices Vol.27, No.10)*, p. 127–145, Vancouver, B.C., October 1992.
- [MAS 95] MASUHARA, H., MATSUOKA, S., ASAI, K., and YONEZAWA, A. “Compiling away the meta-level in object-oriented concurrent reflective languages using partial evaluation”, *Proceedings of OOPSLA'95 (SIGPLAN Notices Vol.30, No.10)*, p. 300–315, 1995.
- [MAS 96] MASUHARA, H., MATSUOKA, S., and YONEZAWA, A. “Implementing parallel language constructs using a reflective object-oriented language”, *Reflection Symposium'96*, p. 79–91, April 1996.
- [MAS 98] MASUHARA, H. and YONEZAWA, A. “Design and partial evaluation of meta-objects for a concurrent reflective language”, *Proceedings of European Conference on Object-Oriented Programming (ECOOP'98)*, Vol. 1445 of *Lecture Notes in Computer Science*, p. 418–439, Brussels, July 1998.

- [MAS 99] MASUHARA, H. “Architecture Design and Compilation Techniques Using Partial Evaluation in Reflective Concurrent Object-Oriented Languages”, Ph.D. Thesis, Department of Information Science, University of Tokyo, 1999.
- [MAT 91] MATSUOKA, S., WATANABE, T., and YONEZAWA, A. “Hybrid group reflective architecture for object-oriented concurrent reflective programming”, *Proceedings of European Conference on Object-Oriented Programming (ECOOP'91)*, Vol. 512 of *Lecture Notes in Computer Science*, p. 231–250, 1991.
- [NAK 95] NAKAYA, A., YAMAMOTO, K., and YONEZAWA, A. RNA secondary structure prediction using highly parallel computers. *Compt. Appl. Biosci.*, Vol. 11, p. 685–692, 1995.
- [OKA 94] OKAMURA, H. and ISHIKAWA, Y. “Object location control using meta-level programming”, *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, Vol. 821 of *Lecture Notes in Computer Science*, p. 299–319. Springer-Verlag, July 1994.
- [ROD 92] RODRIGUEZ, L., JR. “A study on the viability of a production-quality metaobject protocol-based statically parallelizing compiler”, *Proceedings of International Workshop on New Models for Software Architecture (IMSA92): Reflection and Meta-Level Architecture*, p. 107–112, 1992.
- [SIM 92] SIMMONS II, J. W., JEFFERSON, S., and FRIEDMAN, D. P. Language extension via first-class interpreters. Technical Report No. 362, Computing Science Department, Indiana University, Bloomington, Indiana, September 1992.
- [SMI 84] SMITH, B. C. “Reflection and semantics in Lisp”, *Conference record of Symposium on Principles of Programming Languages*, p. 23–35, 1984.
- [TAU 96] TAURA, K. and YONEZAWA, A. “Schematic: A concurrent object-oriented extension to scheme”, *Proceedings of Workshop on Object-Based Parallel and Distributed Computation*, No. 1107 in *Lecture Notes in Computer Science*, p. 59–82. Springer-Verlag, 1996.
- [WAT 88] WATANABE, T. and YONEZAWA, A. “Reflection in an object-oriented concurrent language”, *Proceedings of OOPSLA'88 (SIGPLAN Notices Vol.23, No.11)*, p. 306–315, San Diego, CA, September 1988.