—in Proceedings of Reflection'96, pp.79–91—

Implementing Parallel Language Constructs Using a Reflective Object-Oriented Language

Hidehiko Masuhara* Satoshi Matsuoka† Akinori Yonezawa‡
*Department of Graphics and Computer Science, College of Arts and Sciences
†Department of Information Engineering
‡Department of Information Science
University of Tokyo

Abstract

To provide various parallel language constructs, extensible languages based on reflection are attractive for both implementors and users. This paper describes our proposed meta-level architecture of a concurrent object-oriented language ABCL/R3, which has the following characteristics: (1) language customization through meta-interpreters and meta-objects, (2) delegation and inheritance mechanisms facilitating modular and re-usable meta-level programming, and (3) the *reflective annotations* and *meta-level arguments* realizing separation of base- and meta-level programs. We also show that several meta-level programs that provide common parallel programming strategies, such as object replication and latency hiding, are easily implemented.

1 Parallel Languages and Reflection

Practical parallel and distributed programs often have complicated computation and communication structures for achieving efficiency. For example, latency hiding—which is an optimization technique for remote messages—makes programs complicated because a thread of control must be deliberately 'broken up' into multiple process. To alleviate this complexity, language systems often provide high-level parallel constructs that support such specific optimization techniques. However, it is difficult to provide 'generic' constructs to cover all circumstances, since optimizations tend to be specific to applications or their executing platforms. In addition, implementation of new constructs in the language requires tremendous effort for compiler-writers.

A different (and possibly better) approach is to create a simple, yet extensible language, and provide user-defined constructs as if built-in. This approach is beneficial not only to language implementors, but also to users as they can design and build their own parallel constructs afterwards. Based on this observation, we are developing a reflective object-oriented concurrent language ABCL/R3[6], based on ABCL/f[16], that we provide such extensibility via reflection. The major features of our meta-level architecture are as follows: (1) Meta-interpreters provide programming abstractions with which the user can customize language's syntax and semantics. (2) Meta-objects serve as the meta-level representations of objects that can be used for defining customized object behavior. (3) *Reflective annotations* can be defined as programming directives to the meta-level, and their interpretations can be modified by customizing the meta-interpreters.

The technical details of the ABCL/R3 compiler that uses partial evaluation for compile-time optimization have been presented elsewhere[6]. In this paper, we describe how parallel language constructs

*Contact: 3–8–1 Komaba, Meguro-ku, Tokyo 153, Japan.
E-mail: masuhara@graco.c.u-tokyo.ac.jp
Phone: +81–3–5454–6679, Fax: +81–3–3465–2896.

can be constructed using the meta-level architecture of ABCL/R3; they include object replication, latency hiding, termination detection, and user-level scheduling.

The rest of the paper is organized as follows: The basic (non-reflective) features of ABCL/*f* are described in Section 2. Then example parallel programs are examined and discussed in Section 3. Section 4 presents the meta-level architecture, and the use of the architecture for parallel programs is described in Section 5. Section 6 discusses efficient implementation and related language systems. Finally, Section 7 concludes the paper.

2 Base Language ABCL/*f*

We first introduce the basic (i.e., non-reflective) language ABCL/*f*, a concurrent object-oriented language[16]. In ABCL/*f*, multiple-threading and synchronization are controlled via *future* and *touch* mechanisms[1]. An *object* is an instance of a class, embodying shared mutable data. Messages (or methods) sent to an object are asynchronous, and dispatched according to its class, and mutually excluded in order to preserve consistency.

```
;;; class definition
(defclass account ()
  ((balance :initform 0)))

;;; method definition
(defmethod account deposit (amount)
  (setf balance (+ balance amount))
  balance)

;;; object creation and method invocation
(let ((acc (make-account :balance 100)))
  (print "Current balance: " (deposit acc 50)))

;;; use of the future and touch
(let ((acc (make-account :balance 100)))
  (let ((rbox (future (deposit acc 30))))
    ⟨some computation⟩
    (print "Current balance: " (touch rbox))))
```

Figure 1: Sample code fragment of ABCL/*f*

[1]The future mechanism in ABCL/*f* is similar to the one in Multilisp[2]. The difference is that the former allows the programmers to describe more explicit controls[16]. For example, it can be described when a function should wait for the return value of a future call in ABCL/*f*.

Figure 1 shows an example program in ABCL/*f*. The first expression defines a class `account` with an instance variable `balance`. The second expression defines a method `deposit` for the class `account`. This method adds the given amount to the instance variable `balance`, and returns the updated value. The third expression shows how an object is created and manipulated; `(make-account ...)` creates an object of the class `account`; finally, `(deposit acc 50)` invokes the method `deposit` of object `acc` with argument `50`.

The fourth expression illustrates typical use of `future`. The `future` form invokes a method of object `acc`, but the caller does not block and wait for the end of the method execution. Instead, it receives a "reply box," in which the result of the method will be stored. After the end of ⟨*some computation*⟩, the caller extracts the result from the reply box using the `touch` form. If the result is not ready (i.e., the method `deposit` has not finished yet), the caller is blocked until it becomes available.

The future form can also be used for function invocations. In this case, the function invocation is concurrently executed by a newly created thread. In addition, the future form can take an optional argument "`:on` *p*", which specifies the processor ID where the function will be executed.

3 Examples of Parallel Language Constructs

This section examines several parallel language constructs which are typically used for optimization, etc. in parallel applications.

3.1 Object Replication

Object replication greatly improves performance of programs in a distributed memory environment. When a program frequently accesses a remote object, creating a replica of the object within the local processor will substantially reduces the number of remote messages.

For concreteness, assume there are two vector objects `v1` and `v2` on different processors, and a function `product` (Figure 2) is called on the processor on which `v1` is placed. Since `v2` is a remote object, each

invocation of `nth-element` on `v2` sends and waits on a remote message; this results in $(2 \times |\texttt{v2}|)$ fine-grained remote messages, where $|\texttt{v2}|$ is the length of vector `v2`.

```
(defun product (v1 v2)
  (let ((sum 0.0) (size (size-of v1))
    (dotimes (i size)
      (setf sum
        (+ sum (* (nth-element v1 i)
                  (nth-element v2 i)))))
    sum)))
```

Figure 2: A naively written dot-product function

If replication mechanism were available as a language feature, this program could be optimized by creating a replica of `v2` at the local processor during the computation of `product`[2]. However, this is not a simple task: for effective execution, we must consider the following aspects: (1) how replicas are created and managed (mechanism), (2) how programmers specify creation of replicas (syntax), and (3) how to decide whether an object should be replicated (policy).

Mechanism: There could be a variety of replication algorithms one could provide as a built-in feature of a language. This is not simple as it may seem, because of interaction with other parts of the language. For example, an original object should be locked if instance variables of its replica could be updated and written back afterwards.

Instead, meta-objects in reflective languages could be used to implement different replication algorithms transparent to the user program depending on his base-level algorithmic requirements.

Syntax: Since replicas may be used to optimize existing programs, syntactic support to create replicas without modifying the structure of the original programs is beneficial. In other words, if the replication mechanism is provided as library functions, we may have to modify the structure of the original programs, causing loss of clarity and portability.

[2]This optimization not only reduces the number of messages transferred, but also the time for waiting for answers to each request—so called *latency*.

High Performance Fortran (HPF)'s distribution directives are declarative annotations (comments) which allow the programmers to control the distribution and replications in a non-intrusive manner to base-level programs[3]. The key idea is to provide directives as comments, so that they are non-intrusive. However, the syntax and semantics of HPF directives are fixed and not extensible. Instead, we propose the *annotations* as non-intrusive syntactic extensions at the base-level, and ways of meta-programming to define the associated interpretation of annotations at the meta-level.

Policy: Since creation of a replica has larger overhead than normal method invocation, it does not always improve performance. Unfortunately, there are no general rules to tell when replication is beneficial, but rather, rules are heuristic and situation-dependent. For example, in `product`, `v2` should be replicated only when it is larger than a certain size.

To incorporate such rules, replication mechanism should be flexible so that users can specify their own heuristics. For example, we could have an extended annotation syntax, which accepts an optional expression to decide whether specified objects are to be replicated using run-time values. Our ABCL/R3 allows meta-programming of interpretation of the annotations to cope with such cases.

3.2 Latency Hiding

Latency hiding is an optimization technique to eliminate time to wait for remote messages, where the basic idea is to overlap local computation and remote communication. This is usually realized by modifying programs manually, i.e., by breaking up a single thread of control into multiple threads. The problem is that the modification is not small.

Figure 3 shows two versions of the function `product`, which are manually modified for latency hiding from Figure 2. These two versions are different in the number of method invocation requests that are sent in advance to the actual use of the data (effectively, *prefetching*). In (a), only a request for the element that is used in the next iteration is sent in advance to its use, while requests for all the elements are sent before the computation in (b).

```
(a) (defun product (v1 v2)
      (let ((sum 0.0)
            (size (size-of v1))
            (elm-a (future (nth-element v2 0)))          ; request for the first
            (elm-b nil)
        (dotimes (i size)
          (setf elm-b
                (future (nth-element v2 (+ i 1))))       ; request for the next
          (setf sum (+ sum (* (nth-element v1 i)
                              (touch elm-a))))           ; use of the value
          (setf elm-a elm-b))
        sum))
(b) (defun product (v1 v2)
      (let* ((sum 0.0)
             (size (size-of v1))
             (elms (make-array size)))
        (dotimes (i size)                                ; request for all
          (setf (aref elms i)                            ;     the elements
                (future (nth-element v2 i))))
        (dotimes (i size)
          (setf sum (+ sum (* (nth-element v1 i)
                              (touch (aref elms i))))))) ; use of the values
        sum))
```

Figure 3: "Manual" latency hiding versions of dot-product

We provide a mechanism for latency hiding, in which the programmer specifies when and what method invocation should be requested for prefetching in advance to the actual use of the result of the prefetch in the expressions by means of annotations embedded in the original programs.

The annotation for latency hiding has the following form.

$$e\texttt{\{prefetch } e_a\texttt{\}}$$

This annotated expression is interpreted as follows. Before evaluating the expression e, the expression e_a is evaluated. The expression e_a is expected to result in a sequence of synchronous method invocations (i.e., *present* type messages in ABCL[19]), which are executed as asynchronous (i.e., *future* type) invocations. The return values of these invocations are not used in e_a itself; and resulting reply boxes are stored for later use. Then, the expression e and subsequent expressions are executed. In those expressions, a method invocation form is executed as `touch` operation to the stored reply box, if the form results in the same invocation to one of the invocations performed during the execution of e_a.

Using this mechanism, the annotated latency hiding versions of `product` become as shown is Figure 4. Note that without annotations, these two programs are identical to the original one in Figure 2.

3.3 Termination Detection

Some parallel applications, such as search problems, invokes a large number of threads, where *termination detection* of all the threads is a difficult problem because there is no global control. Several algorithms (*cf.* [9, 14]) have been proposed to solve this problem.

However, when we incorporate a termination detection algorithm into a naively written parallel program, we often have to modify the structure of the original program, such as adding parameters to each function definition and invocation, sending control messages to the other objects, etc. In addition, using a different termination detection algorithm requires different modification, which results in loss of portability.

To cope with this problem, we provide new language constructs `fork` and `fork/wait` for termination detection, and termination detection algorithms that are implemented at the meta-level.

```
(a) (defun product (v1 v2)
      (let ((sum 0.0) (size (size-of v1))
        (dotimes (i size)
            (setf sum
              (+ sum (* (nth-element v1 i)
                        (nth-element v2 i))))     ; performs touch here
            {prefetch (nth-element v2 (+ i 1))}) ; sends a request
        {prefetch (nth-element v2 0)}
        sum)))
(b) (defun product (v1 v2)
      (let ((sum 0.0) (size (size-of v1))
        (dotimes (i size)
            (setf sum
              (+ sum (* (nth-element v1 i)
                        (nth-element v2 i))))) ; performs touch here
        {prefetch (dotimes (i size)            ; even iterations can be
                    (nth-element v2 i))}       ; written in annotations
        sum)))
```

Figure 4: Latency hiding versions of the dot-product function using annotations

Special forms `fork` and `fork/wait` are similar to the asynchronous invocation form `future`, except that they detect global termination. The form `fork/wait` invokes a specified method or function, and waits for the termination of all subsequent sibling computations invoked with `fork` (Figure 5).

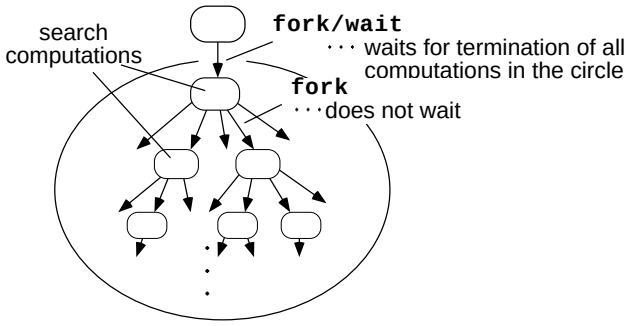

Figure 5: `fork/wait` and `fork` for termination detection

For example, Figure 6 is a `n-queens` problem using this termination detection support. The annotation at the first line declares that a termination detection algorithm called `weight` will be used. The top-level caller invokes function `n-queens` using the `fork/wait` form. Subsequent recursive `n-queens` invocations are achieved with the `fork` form. The top-level caller waits for the termination of all sibling `n-queens` computations. Note that the definition of `n-queens` is independent of the underlying termination detection algorithms.

```
;;; specify the termination detection algorithm
{termination-detection weight}

;;; the top-level caller
(let ((counter (make-counter)))
 ;; invokes and waits
 (fork/wait (n-queens size 0 '() counter))
 (print "number of answers: " (get counter)))

;;; search function
(defun n-queens (size col rows counter)
  (if (= size col)
    (count-up counter)
    (dotimes (row size)
      (if (not-attacked? size col rows row)
        ;; invokes and does not wait
        (fork (n-queens size (1+ col)
                (cons row rows) counter))))))
```

Figure 6: Description of n-queens problem using `fork` and `fork/wait`

3.4 User-Level Scheduling

Application level information is often useful for controlling scheduling to improve performance. For example, the A*-search is an algorithm to find the

best answer in terms of some evaluation function. It uses the estimated value of the answer, which is computed from the intermediate status, as a scheduling priority of a thread. As the branch-and-bound algorithms do, it prunes—terminating subcomputations that have no possibility to reach the best answer—is effective for reducing the search space.

Since such scheduling facilities are not provided in most language systems, programmers are forced to write a program that *explicitly* controls the order of execution. Usually, this is realized with an user-level scheduler object as a server embedded in the base-level application code, and searcher objects as clients, as is shown in Figure 7. (1) The scheduler activates a searcher object. (2) The object sends requests for object creation and activation, instead of creating its sub-objects, for the next search step. These requests are stored in the queue belonging to the scheduler object. (3) When the activated object finishes its execution, it yields its execution by sending a message to the scheduler. (4) The scheduler then selects a request having the highest priority from the queue, and creates and activates a search object that corresponds to the selected request. The queue of the scheduler is sorted by the priority value of each request, and the scheduler can prune requests from the queue.

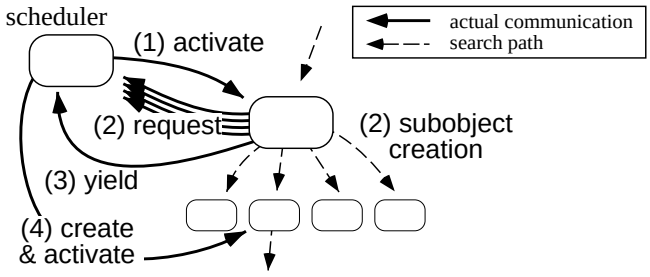

Figure 7: Explicit user-level scheduling system

One of the problem of this programming style is that the control flow in the original algorithm, which is represented as dashed arrows in the figure, is replaced with more complicated communications, which is represented as solid arrows. As a result, the program becomes unclear and difficult to maintain. Our goal is to provide syntactic support which hides such explicit communications with the scheduler, allowing a programmer to write their search algorithms in a 'natural' style in Figure 6.

4 The Meta-Level Architecture

We designed the meta-level architecture of ABCL/R3 so that various language constructs such as the ones shown in Section 3, are straightforwardly implemented at the meta-level as language extensions. Its major features are as follows:

- *Meta-interpreters* of base-level methods and functions are represented as objects, providing an object-oriented programming model of base-level execution. They can be customized to introduce syntax and semantics extensions into the language.
- A *meta-object* gives the meta-level representation of an object, including its class name, instance variables, method definitions, and message queue, all of which can be manipulated as first class data.

In addition to above two major features, there are mechanisms to facilitate simple and modular meta-level programming:

- *Meta-level arguments* can be passed between meta-level programs, which avoids modifying the structure of base-level programs.
- A *reflective annotation* serves as a programmable directive to the meta-level from the base-level programs. In addition, the interpretation of annotations is also defined in the meta-level interpreters.

4.1 Extensions via Meta-Interpreters

Meta-interpreters of ABCL/R3 are meta-level objects that executes *body part* of base-level methods and functions. Although these are defined as objects, the definition looks like the traditional meta-circular interpreters of Lisp, and those of sequential reflective languages (e.g., 3-Lisp[15] and Brown[17]). The definition is divided into smaller methods, each of which corresponds to a specific type of an expression.

Followings are part of the method definitions in the meta-interpreter:

(eval-entry exp env)[3]**:** This method is called at the beginning of each base-level method/function invocation. The argument **exp** is an unevaluated expression of the method/function, and the argument **env** is an environment, which binds instance variables of an object , and formal parameters of a method/function. It simply calls the method **eval** by default.

(eval exp env): This method serves as a dispatcher—it calls an appropriate sub-method (e.g., **eval-var** and **eval-method-call**) according to the expression type.

(eval-var var env): This method handles variable references—it returns value of the variable in the environment.

(eval-method-call exp env): This method handles method-invocation forms in base-level programs. By default, (1) it first determines the invocation type (e.g., future type or present type); and (2) it also determines the target object (i.e., the receiver of the message), and the arguments by using the method **eval**. (3) The "meta-arguments" of the invocation, which is explained later, are determined by calling the method **meta-args**. (4) Finally, it sends a invocation request to the target object using the method **do-method-call**.

(do-method-call type target selector args meta-args env):
This method actually sends a message to the target object. Firstly, a data structure which has all the information for the invocation (e.g., method name and arguments) is composed by the method **make-message**; and then the created data is passed by calling the method **message** of the meta-object of the target.

The extensions are written using *delegation,* so that meta-level description are re-usable.[4]

[3]This denotes that a method called **eval-entry** takes two arguments **exp** and **env**.

[4]This is similar to that CLOS metaobject protocol[5] allows users to write their extensions using *inheritance.*

Meta-level Arguments On a method/function invocation, parameters can be transparently passed from the caller to the callee at the meta-level using the *meta-level argument* interface. When a method or function is invoked, a method **meta-args** of the evaluator is called. The return value of this method, which is a list of keywords and values, is passed onto the target's meta-object along with ordinary parameters of the base-level method or function. Eventually, these values are stored in the evaluation environment on the callee's side, and accessed by the function **lookup-meta**.

An advantage of this mechanism is that the meta-level arguments are invisible from the base-level programs. As a result, we need not modify base-level programs in order to pass parameters that are used only at the meta-level. The use of meta-level arguments is presented in Section 5.3.

Reflective Annotation In ABCL/R3, annotations can be used as directives to the meta-level from base-level programs. An annotation to an expression consists of a keyword and argument expressions; and it is written as follows:

body{*keyword args...*}

Our annotations, called *reflective annotations,* can be customized how they are interpreted. In fact, an annotated expression is evaluated by the following method at the meta-level:

(eval-annotation keyword args body env):
When an annotated expression is to be evaluated, this method is called beforehand. By default, a new evaluator object whose class is specified by the **keyword** argument is created, and then the **body** expression is evaluated by the created evaluator.

Since the method definition can be overridden by user-defined methods, the above interpretation can be changed, as will be shown in Sections 5.1 and 5.2.

4.2 Extensions via Meta-Objects

A *meta-object* is a meta-level representation of a base-level object. Its definition is similar to the

ones in ABCL/R[18] and ABCL/R2[7, 8]. Meta-objects are instances of class `metaobject`. Customized meta-objects can be defined by inheriting this class. Followings are the instance variables in the class `metaobject`, which have implementation-level information of an object:

name	meaning
class	name of the class
state-vars	names and values of instance variables
message-queue	list of messages waiting for acceptance
evaluator	reference to its evaluator object

The methods of the class `metaobject` define how a method invocation on a base-level object is processed. The method invocation mechanism including mutual exclusion, method dispatches, etc., can be modified by defining customized methods. The followings are their protocols:

(message m): When a method of an object is invoked, this method of the corresponding meta-object is invoked. Since multiple method invocations on an object are mutually excluded, this method first checks whether the invocation should be suspended. If so, it then puts the message into the message-queue; otherwise, it calls method `accept`.

(accept m): When a meta-object decides to process a base-level method invocation, this method is called. In the method, an appropriate method definition to the message **m** is selected, and evaluated by calling a method **eval-entry** of its evaluator object.

(become env): When the execution of a base-level method finishes, this method is called at the meta-level. The parameter **env** has values of base-level instance variables, which may have been updated during the execution. The meta-object copies these updated values from **env** in its instance variable **state-vars**. Subsequently, the method **process-next** will be called.

(process-next): This method checks the **message-queue**. If there is any pending message waiting for acceptance, it will be removed from the queue, and subsequently processed by the method **accept**.

5 Implementation of Customized Language Constructs

We have seen several language constructs which can be beneficial for parallel programming. This section shows how these constructs are implemented using meta-level architecture of ABCL/R3.

5.1 Object Replication

Mechanism Since the meta-object of an object contains enough information to create a replica, the replication mechanism is implemented as a method of the class `metaobject`.

First, we define two subclasses of `metaobject`: a class `replicatable` for objects that can create their replicas, and a class `replica-meta` for replicated objects. A method `copy-object` creates a replica on a specified processor (`p`), which is defined as follows:

```
;;; metaobjects of objects that can create replicas
(defclass replicatable (metaobject))

;;; metaobjects of replicated objects
(defclass replica-meta (metaobject)
  (original)) ; additional instance variable

;;; creation of a replica of an object
(defmethod replicatable
                copy-object (p &reply-to r)
  (future
    (make-replica-meta
      :class class :state-vars state-vars
      :evaluator evaluator :original self)
    :on p :reply-to r))
```

In addition, policies for maintaining consistency between an original object and replicas can be controlled. For example, one might want to allow method invocations to an original object while it has replicas. Such a control can be programmed by overriding the methods `message` and `accept` of the class `replicatable`.

Syntax Here, we show an example syntax that creates replicas. The syntax uses the *reflective annotation* so that base-level programmers can exploit

the replication mechanism without modifying the structure of the original programs. The annotation to create replicas is written as follows:

exp{replicate (v_1 v_2 ...) :when *pred*}

When this annotated expression is to be evaluated, *pred* in the annotation is evaluated first. If the result is true, replicas of objects that are bound to the variables v_1 v_2 ... are created; and the *exp* is evaluated in an environment such that the variables v_1 v_2 ... are bound to the replicas.

We can add annotations to the function `product` shown in Figure 2, so as to employ replicas. The program with annotations is shown in Figure 8. Note that the only difference between this program and the original one is the addition of the annotation.

```
;;; specify the default evaluator
{default-eval replica-eval}

(defun product (v1 v2)
  (let ((sum 0.0) (size (size-of v1))
    (dotimes (i size)
        (setf sum
              (+ sum (* (nth-element v1 i)
                        (nth-element v2 i)))))
    {replicate (v2) :when (< 20 size)}
    sum)))
```

Figure 8: Dot-production function using a replica

As stated in the previous section, the interpretation of annotations can be modified by overriding method `eval-annotation`. Here, we define a class `replica-eval` and a method `eval-annotation`, as is shown in Figure 9.

5.2 Latency Hiding

Here, we show how the latency hiding mechanism described in Section 3.2 is implemented at the meta-level. The implementation consists of two parts: invoking methods to prefetch the arguments before their usage in an expression, and substitution of results of the prefetch where needed.

Let us review the annotation syntax that requests prefetch method invocations, which is proposed in Section 3.2:

e{prefetch e_a}

To perform method invocations in e_a, we define two evaluator classes `prefetch-eval` and `prefetch-anno-eval`, and a method for each—definitions are shown in Figure 10.

```
;;; for expressions that may have prefetch annotations
(defclass prefetch-eval ())

;;; for expressions only in annotations
(defclass prefetch-anno-eval ())

;;; interpretation of the annotation
(defmethod prefetch-eval eval-annotation
                        (keyword args body env)
  (if (eq keyword 'prefetch)
    ;; create an evaluator for the annotation
    (let ((eval-a (make-prefetch-anno-eval
                                :delegate self)))
      ;; evaluate expressions in the annotation
      ;; under a special evaluator
      (eval-progn eval-a args env)
      ;; evaluate the body expression
      (eval self body env))
    ;; interpretation of other annotations are delegated
    (eval-annotation
                  super keyword args body env)))

;;; method invocation in the annotation
(defmethod prefetch-anno-eval do-method-call
        (type target method args meta-args env)
  ;; invoke the method in the future type
  (let ((rbox
         (do-method-call super 'future target
                    method args meta-args env)))
    ;; and remember the reply box in the environment
    (remember-prefetched-method env
      type target method args meta-args rbox)))
```

Figure 10: Meta-level program for latency hiding (1)

Expressions except for the prefetch method invocations in the annotation are evaluated as usual. This is implicitly achieved by the delegation mechanism.

The latter part is for using prefetched values; i.e., do `touch` operations instead of method invocation. This is achieved by defining a method `do-method-call` of the class `prefetch-eval` (Figure 11). For each base-level method invocation, this user-defined `do-method-call` checks if the invocation is already performed in an annotation. If so, it touches a corresponding reply box, instead of actual

```
1: (defmethod replica-eval eval-annotation (body args env)
2:   (if (eval self (replica-predicate args) env)
3:     (let* ((target (replica-target args))
4:            (obj (lookup target env))
5:            (rep (denotation (copy-object (meta obj) (this-pe))))
6:            (ex-env (extend-env env (list target) (list rep))))
7:       (let ((answer (eval self body ex-env)))
8:         (copy-back (meta rep))
9:         answer))
10:    (eval self body env)))
```

(l.2) It evaluates *pred* in an annotation. If it is true, (l.5) a replica of a specified object is created. The functions `denotation` and `meta` return corresponding base-level object and meta-object, respectively; and the function `this-pe` returns a current processor ID. (l.7) The body of the expression is then evaluated in an extended environment. (l.8) Method `copy-back` is a user-defined method of the replicated metaobject, which writes values of instance variables in the original metaobject's for consistency management. (l.10) If the *pred* is false, it evaluates the body of the expression as usual.

Figure 9: Interpretation of replica-creating annotations

invocation.

```
(defmethod prefetch-eval do-method-call
        (type target method args meta-args env)
  ;; check whether the method is already
  ;; invoked in an annotation
  (let ((rbox (prefetched-method? env type
                target method args meta-args)))
    (if rbox
      ;; if it is invoked, do touch
      (touch rbox)
      ;; otherwise, invoke as usual
      (do-method-call super type target method
                      args meta-args env))))
```

Figure 11: Meta-level program for latency hiding (2)

5.3 Termination Detection

Here, we show that an automatic termination detection mechanism is implemented at the meta-level of ABCL/R3 using two layers of delegating evaluators. The first one defines each specific termination detection algorithm, and the second one defines the syntax commonly used in all termination detection algorithms. (Figure 12)

At the syntax layer, we define an evaluator class `TD-eval`, which simply dispatches forms

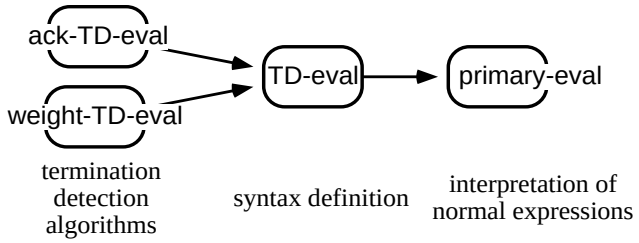

Figure 12: Delegation paths of evaluators for termination detection

`(fork/wait ...)` and `(fork ...)` to the methods `eval-fork/wait` and `eval-fork`, respectively.

At the leftmost layer, an evaluator class is defined for each termination detection algorithm. Here, we only present the simplest one in which an acknowledgment message is returned for each `fork` invocation. Other algorithms—e.g., the one using global weight[9, 14]—can be implemented in similar ways.

The overview of the algorithm is as follows (operations written in the slanted font are performed at the meta-level): (1) A method/function is invoked. (2) *A reply box is created for each child.* (3) A child (sub-computation) is forked. *The reply box is passed onto the child along with the invocation.* (4) *It waits for acknowledgment messages from all*

of its children. (5) Each child returns an acknowledgment message when it finishes. (6) When all acknowledgment messages are collected, it returns an acknowledgment message to its own parent.

The evaluator for this termination detection algorithm can be defined as follows. The method `meta-args` is customized to send a reply box along with base-level arguments; and the method `eval-entry` is customized to add special behavior at the beginning and the end of a base-level method/function execution.

```
(defmethod ack-TD-eval eval-entry (exp env)
  ;; create a location for created reply boxes
  (let* ((new-env
          (extend-env env :meta 'rboxes '()))
         ;; run the body of the method
         (result
          (eval-entry super exp new-env)))
    ;; wait for termination of children
    (dolist (rbox (lookup-meta 'rboxes new-env))
      (touch rbox))                          ; (4)
    ;; notify its parent of the termination
    (reply (lookup-meta env 'ack) t)         ; (5,6)
    result))

(defmethod ack-TD-eval meta-args
     (type method target args options env)
  (let ((margs (meta-args super type method
                 target args options env)))
    (cond ((eq type 'fork)
           ;; create a reply box
           (let ((rbox (make-reply-box))) ; (2)
             ;; and remember it in the environment
             (push-meta-env env 'rboxes rbox)
             (list* 'ack rbox margs)))     ; (3)
          (t margs)))) ; for the other forms
```

5.4 User-Level Scheduling

In Section 3.4, we have seen an example of user-level scheduling that is achieved by explicitly sending messages in application programs. Here, we show a similar scheduling mechanism that is achieved by *implicitly* communicating with the scheduler object. An evaluator class `sched-eval` and two methods are defined: (`do-method-call`) Instead of invoking a method, a message data is sent to the scheduler object. (`eval-entry`) At the end of each method execution, it notifies the scheduler to yield its thread of control. We assume that the scheduler object is bound to variable `*scheduler*` and has methods `request` and `run-next`.

```
;;; an evaluator class for user-customized scheduling
(defclass sched-eval ())

;;; send a request to the scheduler for method invocation
(defmethod sched-eval do-method-call
      (type target method args meta-args env)
 (let ((m (make-message
           target method args meta-args)))
   (request *scheduler*
            (cons m (get-priority env)))))

;;; yield the control to the scheduler
(defmethod sched-eval eval-entry (exp env)
  (eval-entry super exp env)
  (run-next *scheduler*)) ; notify end of execution
```

6 Discussions

Runtime Performance: Since reflective languages use meta-circular interpreters as their execution model, naively implemented reflective systems suffer from the *interpretation overhead.* Reducing this overhead is necessary to make reflection feasible for practical parallel programming.

To solve this problem, we have already proposed a compilation framework using partial evaluation[6]. This framework essentially removes the interpretation of the meta-level interpreter, and generates a program that includes the effect of user's modification at the meta-level. For example, from the definition of `product` with `{replicate ...}` annotation in Figure 8 and the extended meta-interpreter shown in Figure 9, our compiler can generate the following code, which does not involve interpretation:

```
(defun product (v1 v2)
  (let ((sum 0.0) (size (size-of v1)))
    (if (< 20 size)
      (let ((t001 (denotation
                   (copy-object (meta v2)))))
        (dotimes (i size)
          (setf sum
            (+ sum (* (nth-element v1 i)
                      (nth-element t001 i)))))
        (copy-back (meta t001))
        sum)
      (progn
        (dotimes (i size)
          (setf sum
            (+ sum (* (nth-element v1 i)
                      (nth-element v2 i)))))
        sum))))
```

This code is nearly as same as the optimal code that the user would manually write in order to implement replication requests.

The problem of this compilation technique is that the partial evaluation is not ultimate; a partial evaluator might generate an inefficient code for some meta-programs. In addition, our current compilation framework handles only evaluator objects. Making customized meta-objects efficient is a subject of future research.

Related Work There are several reflective systems for distributed programming such as AL-1/D and CodA, but few systems pay strong attention to the overhead of reflection.

AL-1/D[12] is a reflective language based on 'multi-model reflection framework' (MMRF). It is reported that performance improvement is achieved by using AL-1/D's meta-level architecture to implement and control object migration mechanisms[11]. However, this improvement seems to be worthwhile over the overhead incurred by reflection in environments where overhead of remote communication is substantially larger than the interpretation overhead, such as heterogeneous distributed computing.

In the CodA meta-level architecture[10], a meta-object is decomposed into several objects, so that user's meta-level programs are easily re-used and composed. However, efficient implementation of such an architecture is not obvious.

Recently, extensible compilers based on the meta-level architecture—so called *compile-time meta-object protocol (MOP)*—have been proposed, especially for parallel/distributed systems[1, 4, 13]. Most language constructs discussed in this paper could also be implemented with these MOPs. However, they may force different programming style at the meta-level; since they are compilers, programmers have to write their extensions as program transformation rules, in which programmers must distinguish run-time values and compile-time values.

7 Conclusion

This paper has presented the design of the meta-level architecture of ABCL/R3. Assuming the compilation technique using partial evaluation, our architecture allows flexible extensibility for the syntax and semantics of the base-level languages in an inexpensive manner. In addition, customization of meta-interpreters can be easily re-used by using standard object-oriented techniques such as delegation in the same way to sequential reflective object-oriented languages (e.g., CLOS) allow. Reflective annotations allow programmers to write directives to the meta-level programs as comments to a base-level program. The interpretation of annotations can also be customized by modifying the meta-interpreters.

This paper also describes how several common parallel programming strategies, including object replication, latency hiding, termination detection, and user-level scheduling, are straightforwardly realized as language constructs by using our architecture. These strategies can be incorporated into existing base-level programs without modifying their structures.

Acknowledgements

We would like to thank Jean-Pierre Briot, Shigeru Chiba, and Michiharu Takemoto for their helpful comments on the early drafts of the paper. We also thank the members of the Yonezawa's research group for fruitful discussions on our work.

References

[1] Shigeru Chiba. A metaobject protocol for C++. In *Proceedings of OOPSLA'95 (SIGPLAN Notices Vol.30, No.10)*, pages 285–299, 1995.

[2] Robert H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[3] High Performance Fortran Forum. High performance Fortran language specification. *Scientific Programming*, 2(1), June 1993.

[4] Yutaka Ishikawa. Meta-level architecture for extendable C++. Technical Report TR-94024, Real World Computing Partnership, 1994.

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

[6] Hidehiko Masuhara, Satoshi Matsuoka, Kenichi Asai, and Akinori Yonezawa. Compiling away the meta-level in object-oriented concurrent reflective languages using partial evaluation. In *Proceedings of OOPSLA'95 (SIGPLAN Notices Vol.30, No.10)*, pages 300–315, 1995.

[7] Hidehiko Masuhara, Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Object-oriented concurrent reflective languages can be implemented efficiently. In *Proceedings of OOPSLA'92 (SIGPLAN Notices Vol.27, No.10)*, pages 127–145, 1992.

[8] Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In *Proceedings of ECOOP'91*, 1991.

[9] F. Mattern. Global quiescence detection based on credit distribution and recovery. *Inf. Proc. Lett.*, 30(4):195–200, 1989.

[10] Jeff McAffer. Meta-level programming with CodA. In *Proceedings of ECOOP'95*, 1995.

[11] Hideaki Okamura and Yutaka Ishikawa. Object location control using meta-level programming. In Mario Tokoro and Remo Pareschi, editors, *Proceedings of ECOOP'94 (LNCS 821)*, pages 299–319, 1994.

[12] Hideaki Okamura, Yutaka Ishikawa, and Mario Tokoro. AL-1/D: Distributed programming system with multi-model reflection framework. In *Proceedings of IMSA'92 Workshop on Reflection and Meta-Level Architecture*, pages 36–47, 1992.

[13] Luis Rodoriguez Jr. A study on the viability of a production-quality metaobject protocol-based statically parallelizing compiler. In *Proceedings of IMSA'92 Workshop on Reflection and Meta-Level Architecture*, pages 107–112, 1992.

[14] K. Rokusawa, N. Ichiyoshi, T. Chikayama, and H. Nakashima. An efficient termination detection and abortion algorithm for distributed processing systems. In *International Conference on Parallel Processing*, volume I, pages 18–22, 1988. also published as ICOT-TR 341.

[15] Brian Cantwell Smith. Reflection and semantics in Lisp. In *Proceedings of POPL'94*, pages 23–35, 1984.

[16] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. ABCL/f: A future-based polymorphic typed concurrent object-oriented language – its design and implementation –. In G. Blelloch, M. Chandy, and S. Jagannathan, editors, *Proceedings of DIMACS Workshop on Specification of Parallel Algorithms*, pages 275–292. 1994.

[17] Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. In P. Maes and D. Nardi, editors, *Meta-Level Architecture and Reflection*, pages 111–134. Elsevier Science, North-Holland, 1988.

[18] Takuo Watanabe and Akinori Yonezawa. Reflection in an object-oriented concurrent language. In *Proceedings of OOPSLA'88*, pages 306–315, 1988. (revised version in [19]).

[19] Akinori Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System*. MIT Press, Cambridge, MA, 1990.