Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently*

Hidehiko Masuhara **Satoshi Matsuoka** **Takuo Watanabe**‡
Akinori Yonezawa
Department of Information Science, The Universtiy of Tokyo†

Keywords and Phrases:
Object-Based Concurrency, Object Groups, Reflection,
Hybrid Group Architecture, Meta-level Compilation

Abstract

Computational reflection is beneficial in concurrent computing in offering a linguistic *mechanism* for incorporating user-specific *policies*. New challenges are (1) how to implement them, and (2) how to do so efficiently. We present efficient implementation schemes for object-oriented concurrent reflective languages using our language ABCL/R2 as an example. The schemes include: efficient lazy creation of metaobjects/meta-groups, partial compilation of scripts (methods), dynamic progression, self-reification, and light-weight objects, all appropriately integrated so that the user-level semantics remain consistent with the meta-circular definition so that the full power of reflection is retained, while achieving practical efficiency. ABCL/R2 exhibits two orders of magnitude speed improvement over its predecessor, ABCL/R, and in fact compares favorably to the ABCL/1 compiler and also C + Sun LWP, neither supporting reflection.

*To be presented at ACM OOPSLA'92, Vancouver, Canada, Oct. 1992

†Physical mail address: 7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan. Phone +81-3-3812-2111 ex. 4108. E-mail: {masuhara, matsu, yonezawa}@is.s.u-tokyo.ac.jp

‡Currently with School of Information Science, Japan Advanced Institute of Science and Technology, Hokuriku, Japan. E-mail: takuo@jaist-east.ac.jp.

1 Introduction — Why Reflection and How?

Reflection is the process of reasoning about and acting upon the system itself[1] [16, 10, 25]. Contrary to the misconception that 'reflection' is some difficult-to-understand/impractical concept, we believe it is a practical and general scheme in the context of OO(Object-Oriented)-systems as a new methodology in constructing malleable, large-scale systems such as programming languages[9], operating systems[22, 21], and window systems[14]. Reflection is similar to inheritance in that it offers a *mechanism* for incorporating user-level *policies* into the system; that is, the user is offered with a clean interface with which he can customize his system according to his requirements. Furthermore, reflection can augment inheritance in OO-languages for coping with dynamic aspects of the system where inheritance is less effective.

We claim that reflection is especially beneficial in concurrent systems that require customization according to the system organization or application program characteristics for achieving efficiency, ro-

[1] To expose, or *reify* its internals, a reflective system embodies *reifiable* data that represents or implements the structural and computational aspects of itself within itself at the meta-level. Such data must be dynamically self-accessible and self-modifiable by the user program. Furthermore, modification by the user must be 'reflected' to the actual computational state of the user program — this property is termed as *causal-connection*.

bustness, etc. In ordinary (non-reflective) languages, those aspects are controlled only in exceptional and ad-hoc manners; examples are scheduling, communication, and load-balancing, etc. By contrast, reflective languages expose 'concurrent computations' and allow the programmer to control the computations with the same uniform interface as manipulating some data structure — this has been demonstrated in our past work in OOCR (Object-Oriented Concurrent Reflective) languages, such as ABCL/R and ACT/R[19, 20].

The major challenges of reflective languages and systems have been (1) how to implement it, and (2) how to do so efficiently. In order to maintain the causal-connection, the semantic model of a reflective language is usually given with an infinite tower of metacircular interpreters. Since it is impossible to literally implement the 'infinite' tower, most implementations 'bottom out' at some level with non-reflective version of the language, but this scheme is inefficient when the higher levels are interpreted. The novel idea in the initial work of 3-Lisp by Rivières and Smith[7] was to employ a single interpreter and 'shift up' only when reflective procedures are invoked, instead of always performing metacircular interpretations. Since then, there have been few efforts on efficient implementation, which have been (1) only in the context of sequential languages, and/or (2) drastically limited the reflective capabilities to simple ones that could be efficiently implemented, e.g., message dispatch, and/or (3) have only proposed the overall idea of how it might be possible, but without concrete methodologies or actual implementations [15, 4, 3, 6].

As far as we know, there have been no study on efficient implementation of concurrent reflective languages; this is probably due to the difficulty in obtaining the true CCSR of the current state computation due to inter-level concurrency and the time delays incurred in inter-level communication[17]. Thus, for example, Rivières-Smith approach is not entirely applicable, because it does not account for concurrency, i.e., it assumes that at any given time only one level is running. Furthermore, because their scheme is still interpretive, the resulting language will not be able to compete with compiled non-reflective languages when running standard programs. Thus, an alternative way of efficiently breaking the meta-circularity in OOCR languages without sacrificing the lucidity of the reflection must be devised.

This paper presents an efficient implementation scheme of OOCR languages, with the language ABCL/R2[11] as a target. ABCL/R2 features the *hybrid group architecture* that consists of the *individual tower* and the *group tower*. Contrary to its predecessor, ABCL/R[19], whose implementation was extremely slow due to interpretation that bottomed out with ABCL/1[24], ABCL/R2 is compiler-based, is independent from ABCL/1, and in fact competes with the ABCL/1 compiler for speed. ABCL/R2 runs on top of a parallel version of Common Lisp on OMRON LUNA-88K, a shared-memory Mach machine. Benchmarks have shown that the execution speed of ABCL programs on ABCL/R2 (1) is nearly or over two-orders of magnitude faster than that on ABCL/R, and (2) closely compares with or even exceeds the speed on the publically distributed version of our ABCL/1 compiler[24] and also C + Sun Light-Weight Processes (LWP) library, neither of which support reflection.

The schemes we have developed for efficient implementation of OOCR languages are: (1) Efficient Lazy Creation of Metaobjects and Metagroups, (2) Partial Compilation of Scripts (Methods), (3) Dynamic Progression of Degree of Reflectivity, (4) Self-Reification of Group Kernel Objects, (5) Non-reifying Objects (User-level) and Light-weight Objects[2]. We have also managed to *integrate* the optimized components of the system, so that the system remains faithful to the meta-circular definition of ABCL/R2[11]. For example, compilation coexist with reification, and objects of various degrees of reflectivity can send messages to each other. The obvious test is whether the meta-circular definition of ABCL/R2 in [11] runs on top of our system, and in fact it does.

[2] We make a brief note that, in this study, we limit the scope of "efficient implementation" to reflection; traditional compiler optimizing techniques were also incorporated but are not presented here.

2 ABCL/R2 — A Hybrid Group Architecture Language

ABCL/R2[11] is based on the *Hybrid Group Architecture* (HGA), which combines the characteristics ABCL/R[19], which is based on the *individual-based architecture,* and ACT/R[20], which is based on the *group-wide architecture*. Its key features are (1) Heterogeneous object group and group shared resources, (2) Meta-groups and individual/group reflective towers, and (3) Non-reifying objects (objects that lack individual-based reflective capabilities). ABCL/R2 aims to model meta-level encapsulation of control of limited resources shared within object groups, such as computational power, communication, storage, etc. Such "limited resources" have meta-level representation as objects, and are shared among the base-level objects in a group. At the same time, each object has its own *reflective tower*, allowing meta-operations to be customized on a per-object basis. (See [11] for metacircular definition of ABCL/R2.)

2.1 Object Groups and Reflective Towers in ABCL/R2

An object in ABCL/R2 always belongs to some *group*. Objects in a group share computational resources which are represented as *group kernel objects* at the meta-level. Groups can be created dynamically, whose process is defined metacircularly with ABCL/R2. As a result, we have both the tower of metaobjects called the *individual tower* per each object, and the tower of *meta-groups* called the *group tower* per each group. The distinctions between the two towers are as follows:

- The *individual tower* of metaobjects mainly determines the structure of the object, including its script (i.e., method). Thus, for example, reflective operations to alter the script is in the domain of the individual tower.

- The *group tower* of meta-groups mainly determines the group behavior, including the computation (evaluation) of the script of the group members. Thus, group-wide operations that share group resources, such as scheduling, are in the domain of the group tower.

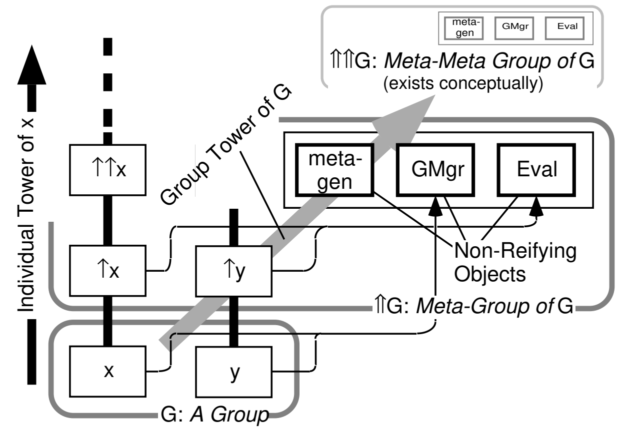

Figure 1: Reflective Architecture of ABCL/R2

Figure 1 illustrates the reflective architecture of ABCL/R2. The structure and the computational behavior of a group are defined at the meta- and higher levels of the group, by the *group kernel objects*. In Figure 1, objects labeled "`meta-gen`" (*metaobject generator*), "`GMgr`" (*group manager*), and "`Eval`" (*evaluator*) are the standard group kernel objects. Additional group kernel objects can be defined by the user to represent other types of shared resources. The characteristics of these objects are follows:

Group Manager: The group manager represents and 'manages' the group. The identity of a group is that of the group manager object — thus, any messages sent to the group is actually forwarded to the group manager. Newly created objects become the member of the same group as its creator; alternatively, a new object can be created at a particular group by explicit designation of the group identity in the object creation form. For example, evaluation of the form below creates a new group `concurrency-controlled-group`:

```
[group concurrency-controlled-group
  (meta-gen meta-gen-with-priority)
  (evaluator eval-with-priority)]
```

At the meta-level, a group manager is created with two customized group kernel objects: `meta-gen-with-priority` (metaobject generator), and `eval-with-priority` (evaluator). The customizations cause all the objects that are members of `concurrency-controlled-group` to be automatically subject to priority-queue based scheduling, even though their base-level programs remain the same.

Metaobject generator: Upon object creation, the metaobject generator creates a metaobject at the meta-level. When a form `[object ...]` is evaluated by the evaluator, a message `[:new` *StateSpec LexEnv ScriptSet Evaluator GMgr*`]` is sent to the metaobject generator (the arguments contain the necessary information for creation of the new object in the group.). A metaobject created by the standard metaobject generator contains: a message queue object, a set of scripts (methods), a set of state variables (i.e., instance variables), references to the evaluator and the group manager, and the execution mode. The reception of a message M by a base-level object corresponds to its metaobject receiving a message `[:message` M R S`]`. The metaobject then places the message M into the message queue object it has. The metaobject also searches for a script that matches a message at the top of the queue, and if it finds one, sends a message to the evaluator to execute the script.

Evaluator: The evaluator is the computational resource shared by the (base-level) members of a group. Its role is to evaluate the scripts of member objects. It is no longer a stand-alone, private object as is with ABCL/R, but interacts with other group kernel objects and metaobjects for group management, such as scheduling. Its typical behavior is as follows: it accepts a message for evaluation of an expression `[:do` *Exp Env Id Gid Eval*`]`, with reply destination designated to a *continuation object* C. Upon receipt of the message, it evaluates the expression *Exp* under the environment *Env*, then sends the result to C. The arguments *Id* and *Gid* are mapped to references of the pseudo variables `Me` and `Group`; the former denotes the object itself and the latter the group of the object, respectively.

```
[object Evaluator
 (script
  (=> [:do Exp Env Id Gid Eval] @ C
   (match Exp
     (is [:variable Var] ;; Variables
       (match Var
         (is 'Me ![den Id]) ;; pseudo variables
         (is 'Group !Gid)
         (otherwise
           [Env <= [:value-of Var] @ C])))
     ;; Past-Type Message Transmisson
     (is [:send-past Target Message Reply]
       [C <= nil] ;; the value of this expression
       (if (not (null Target))
         [[meta Target]
          <= [:message Message Reply [den Id]]]))
     ;; Creation of a New Object
     (is [:object-def Name Meta-gen GMgr
                                    State Script]
       (temporary [Env := initialize state variables])
       ;; delegate to the group manager
       [GMgr <= [:new State Env Script]
           @ [cont ...
          :
```

Figure 2: Metacircular Definition of the Evaluator

The argument *Eval* is for executing the sub-expressions generated during the execution of an expression. This allows the evaluator to delegate the execution of the sub-expressions to another evaluator in a multi-evaluator environment.

2.2 Non-reifying Objects

In ABCL/R2, users can create objects that run more efficiently compared to standard objects by sacrificing individual-based reflective capabilities. When the form below is evaluated, a *non-reifying object* named `x` is created:

```
[object x
  (meta-gen non-reifying-meta)
     :
```

A non-reifying object has a special metaobject that does not allow meta-level operations. Furthermore, the metaobject is also defined to be a non-reifying object. The resulting behavior of `x` is almost same as the one for the standard object (created *without* the designation '`(meta-gen non-reifying-meta)`'), except for faster execution speed and the non-availability of individual-based reflective operations (It is still OK, say, to send messages directly to other meta-level objects, and to be subject to group-wide meta-level control, etc.).

2.3 ABCL/R2 Reflective Programming

Reflective programming in ABCL/R2 is performed in two ways. In order to affect the behavior of an individual object, one sends messages to and/or customizes its metaobject in the same way as ABCL/R, as described in[19]. In order to affect the group-wide behavior such as resource sharing, one sends messages to its group, `[group-of` x`]` (which is delivered to its group manager object), and/or customizes the group kernel objects. The two schemes are not contradictory; in practice, a combination of both schemes is effectively used.

3 Implementation Issues for OOCR Languages

As introduced in Section 1, the two issues in efficient implementation of reflective languages are: how to break the meta-circularity safely, and how to do so efficiency. Since ABCL/R2 is based on the hybrid group architecture, the implementation must always guarantee the proper maintenance of the following causal-connection properties:

- Reflective operations on the individual tower, i.e., message sends to a metaobject to invoke reflective operations, and/or dynamic customization of a metaobject (via the meta-metaobject), must be reflected properly and solely to the object it represents (*localization*).
- Reflective operations on the group tower, i.e., message sends to the group manager object and/or dynamic customization of the group kernel objects, must be reflected properly to all members of group (at the base-level) of the group tower.

Moreover, the actual system must be finitely constructible and executable in real-life. Thus, the problem is, how do we safely break the meta-circularity of the two towers.

Efficiency issues are more problematic. In order to achieve practical efficiency, the implementation cannot remain solely interpretive. Instead, it must include a *compiler* which emits code that allows reflection, thereby achieving practical efficiency. Too much compilation, however, would not allow for reflection to occur. Interfacing of reflective and non-reflective portions in compiled code is another problem; although mixture of interpretive and compiled code has been done on a per-function basis in Lisp compilers, our case is more intricate due to the existence of both (1) reflection and (2) concurrency. Finally, the object-level problem is how to integrate the mixture of group kernel objects and individual metaobjects and meta-metaobjects...etc., that have been subject to various optimizations such as lazy creation of the individual tower, so that they are able to send messages to each other and remain consistent with the metacircular definition of ABCL/R2.

4 Implementation Scheme of ABCL/R2

4.1 Overview of our Efficient Implementation Scheme

Currently, a new version of ABCL/R2 is implemented on top of the multi-threaded version of Common Lisp, running on Omron Luna-88k, a shared-memory computer. As mentioned earlier, the execution speed of this version compares closely with, and sometimes exceeds that of the original ABCL/1 compiler, with appropriate user intervention.

Our implementation scheme is structured as follows: First there is an underlying *low-level kernel* that provides the basic primitives for (non-reflective) object execution. Specifically, it maps the underlying Lisp threads to objects. Secondly, the compiler performs *partial compilation* of scripts (i.e., methods). The compiled code is such that it allows co-existence of reflective and non-reflective code, so that execution efficiency is maintained while retaining reflective capabilities. Default system objects are further optimized by providing a pair of non-reflective and reflective compiled codes for *self-reification*. The compiled code is augmented with *light-weight objects*, that serves as fast replacements for normal objects in special roles such as *continuation objects* and

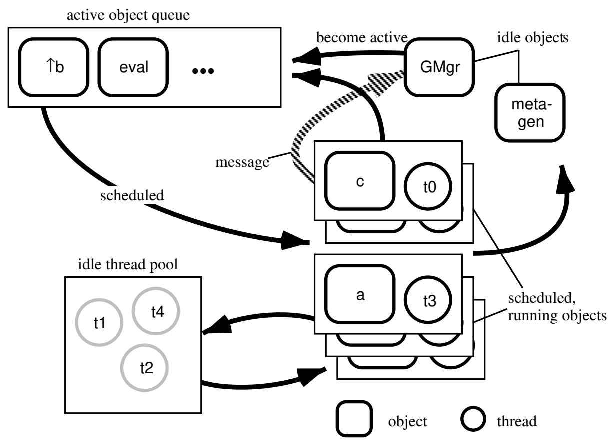

Figure 3: The Low-Level Kernel

light-weight metaobjects. The individual tower is constructed lazily with *dynamic progression* techniques, in which full-fledged creation of metaobjects is avoided until its capabilities are absolutely necessary. Interfacing of various objects with different degrees of reflectivity (including non-reifying objects, which are essentially fully compiled) become possible with *inter-level message forwarding*, which has the effect of avoiding unnecessary reification. Finally, lazy creation of meta-groups along the group tower is made possible with lazy creation of group managers and self-reification techniques.

4.2 The Low-Level Kernel

The *low-level kernel* has the task of providing the basic primitives for (non-reflective) object execution. Based on the underlying multi-threaded Lisp environment, Lisp threads are mapped to objects to provide primitive computational resources. It can be regarded as a simplified operating system kernel in that it provides primitive message transmission and scheduling capabilities. Figure 3 illustrates the structure of the low-level kernel.

For primitive message transmission, when a message is sent to an object, a triplet $\langle$*message body*, *reply destination*, *message sender*$\rangle$, is placed into the message queue of the receiver object. Then, if the object has been 'dormant', the object is made 'active' and placed into the *active object queue*.

The kernel employs a standard technique of creating a pool of fixed number of Lisp threads, from which a thread is acquired and assigned to objects for execution. A thread dequeues an object from the active object queue, then executes the compiled Lisp lambda-closure associated with the object. The lambda-closure describes the basic behavior of the object — namely, remove one message from its message queue, then execute the script matching the message pattern. After the execution of the script terminates, the object returns to the active object queue if it is still active (i.e., the message queue is non-empty), or becomes idle when it becomes dormant (i.e., the message queue is empty). The thread then starts the execution of another active object in the active object queue.

4.3 Partial Compilation of Scripts

The primary focus of reflection in ABCL/R2 is the ability to perform coordinated resource management, such as object scheduling and group consistency maintenance. For this purpose, 'what is to be reflected' concentrates on the concurrency aspect of the object-oriented concurrent computation; in other words, primitives in intra-object sequential computations — say, primitive arithmetic expressions — would serve little purpose if it could be reflected upon. (This is similar in spirit to CLOS Metaobject Protocol[9], where only the features added by CLOS, such as generic function dispatch, could be reflected upon, whereas Common Lisp features are basically hard-wired).

From such a perspective, we categorize operations into *reflective operations* and *non-reflective operations*. In ABCL/R2, the following operations can be reflected upon:

- Reference to the variables (including the references to lexical variables, and pseudo variables '`Me`' and '`Group`').
- Message sending (both 'past type' (asynchronous) and 'now type' (RPC-style)).
- Object creation and group creation.

As indicated above, other primitive operations such as arithmetic operations are not subject to reflection, and are thus compiled. Here, in order for reflective and non-reflective operations to coexist, the following scheme is adopted:

- Consecutive expressions representing non-reflective operations are compiled into a simple Lisp lambda-closure (which is, in turn, compiled into native code by the Lisp compiler).
- Other expressions are also similarly compiled into a lambda-closure. The difference is that, when reflective operations appear within expressions that are non-reflective, a code to send a message to evaluator explicitly is embedded into the lambda-closure. The evaluator in turn receives the message and executes the reflective operations.
- Conversely, the evaluator can also receive a compiled lambda-closure as a message for direct execution. Thus, non-reflective operations embedded within reflective ones can be initiated by sending a message to evaluator.

Abridged compilation rules are shown in Appendix A; here, as an example, let us consider the compilation of an expression `[x <= (* y 2) @ z]`, meaning "send the value of `(* y 2)` to the value of `x` where the reply destination is the value of `z`." Due to the lack of side effects, we evaluate the expression in the following order: (1) Reference the values of `x`, `y`, and `z`. (2) Compute the value of `(* y 2)`. (3) Send the value of `(* y 2)` to the value of `x`, with the value of `z` as the *reply destination*. Since the variable references and the message sending are reflective, the complied code has expressions explicitly requesting those operations to the evaluator; more specifically, the resulting compiled code is as follows: (1) send a message for variable reference to the evaluator, (2) receive the values of the variables by a newly created continuation, (3) compute the value of the sub-expression `(* y 2)`, then (4) send a message for message sending to the evaluator. Following is the Lisp code generated by the ABCL/R2 compiler (For reader clarity, we write [x <= y @ z] for the actual Lisp expression generated (`send-message` x y z), meaning a message send to x with y, with reply destination z.):

In the compiled code shown in Figure 4, arguments `C`, `Env`, `Id`, `Gid`, and `Eval` denote the continuation, the environment, the object ID, the group ID of the object, and the evaluator for the sub-expressions, respectively. `:Do-evlis`, `:variable`, and `:send-past` are message tags that request the evaluator to evaluate a list of expressions, reference a variable, and perform the past-type message transmission. The form `[cont ...]` creates a *continuation object* (explained later), to which the reply from the evaluator is sent.

```
#'(lambda (C Env Id Gid Eval)
    [Eval
     <= [:do-evlis
          [[:variable 'x]
           [:variable 'y] [:variable 'z]]
          Env Id Gid Eval]
     @ [cont [x-value y-value z-value]
         [Eval
          <= [:do [:send-past x-value
                              (* y-value 2) z-value]
                  Env Id Gid Eval]
          @ C]]])
```

Figure 4: Compilation of `[x <= (* y 2) @ z]`

4.4 Self-reification of Default System Objects

In addition to the partial compilation of normal objects, the default group kernel objects (the group manager, the metaobject generator, and the evaluator) and metaobjects of regular objects each has a pair of scripts that are pre-compiled; one is the bottomed-out script and the other is the meta-circularly defined script for *self-reification*: Such objects initially execute with their bottomed-out (compiled) script (Figure 5(a)). Then, when their metaobjects are accessed, to which some reflective operation is requested, the former script creates a (default) metaobject that embodies the set of compiled scripts for that particular object.

When the metaobject created, it is initialized so that the metaobject could continue execution from the point where the reflective operation was requested in the bottomed-out script (this can only be realized with explicit argument passing rather than environment reification because the latter is not possible with bottomed-out, compiled script). Then the execution is delegated to the newly created metaobject, which starts the execution of the requested reflective operation (Figure 5(b)).

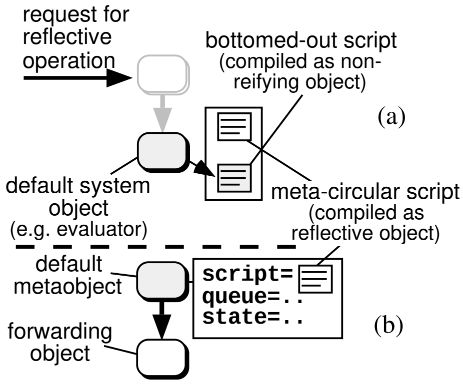

Figure 5: Self-Reification Mechanism

4.5 Light-weight Objects

Normal objects are heavyweight in the sense that it must support every conceivable operations allowed for an object. Here, the optimization strategy we employ is to sacrifice some of the capability of normal objects in trade for efficiency, and use them where appropriate. For this purpose, we introduce *light-weight objects*. The light-weight object does not have a message queue, but has a compiled script as a lambda-closure which is directly executable. When an ordinary object sends a message to a light-weight object, the sender executes the script of the receiver by directly invoking the lambda-closure of the object. It also does not have state variables; thus, it cannot have its own internal state. Execution overhead is reduced for two reasons: (1) its script is executed without scheduling overhead, and (2) its creation cost is smaller compared to normal objects due to the lack of state variables and the message queue.

Because of its limited functionality, objects which could be light-weight must satisfy the following requirements:

stateless, because it does not have state variables,

receive a single message at a time, because it does not have a message queue

only perform simple operations, because complex operations would otherwise lead to deadlocks.

Light-weight objects are used extensively in ABCL/R2 to reduce the overhead of execution: one major use common to all levels is the *continuation object*; in the meta-level, it is also employed as the *light-weight metaobject*. Here, we defer the description of the latter until Section 4.6, and concentrate on its general use as a continuation object.

OOCP languages typically employ a programming style whereby continuation objects are created for delegating the result to, or synchronizing controls[1]. The meta-circular definition of metaobjects is a typical example: when a metaobject sends an expression to the evaluator for execution, the metaobject creates a continuation object, which receives a reply from the evaluator, and in turn notifies the end of evaluation by sending a `:end` message to the metaobject. This allows script execution and message reception to occur simultaneously[19]. In this manner, efficient implementation of continuation objects is crucial in achieving higher overall performance.

In ABCL/1, creation of continuation object was actually a syntactic macro that created a normal object. An expression:

[cont *Pattern Expressions*]

was equivalent to the following expression:

[object
  (script (=> *Pattern Expressions*))]

In ABCL/R2, continuation objects are implemented with light-weight objects, since the ways in which continuation objects are employed satisfy the abovementioned requirements: (1) stateless (temporal), (2) receives one and only one message, (3) its script is simple and terminates within finite steps, usually delegating the computed result to the next continuation object with a past type message send. The effectiveness of the light-weight objects is shown in Section 5.

4.6 Creation of Individual Tower via Dynamic Progression of Reflectivity

To construct the individual tower without unnecessary reification (i.e., creation of metaobjects), we employ the *dynamic progression* technique, which is an extension of lazy creation.

Figure 6 illustrates the implementational structure of an individual tower. To maintain the causal-connection, at any given time an tower has

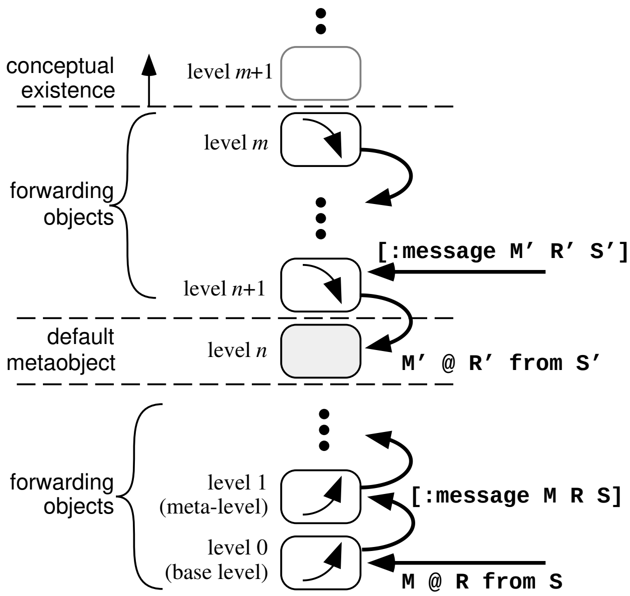

Figure 6: Snapshot of an Individual Tower

one and only one active object, the *default metaobject*, that could be directly executed by the low-level kernel (the shaded object at level n). All the other objects in the tower are either (1) *forwarding objects* (the white objects with solid border), or (2) of 'conceptual' existence i.e., does not physically exist within the system (the white object with hatched border). The level at which the default metaobject resides indicates the level of reflective operations that has been requested: in the figure, the reflective operation has been requested at level $(n-1)$ since the default metaobject resides at level n. Metaobjects above level n up to level m have been accessed in the past with the `[meta ...]` form. Objects above level m do not exist as they have not been accessed yet.

When the evaluator evaluates a message sending expression of a metaobject at level n, it attempts to send the message to the metaobject of the destination object at the same level, n. When the level of the metaobject of the receiver does not match that of the sender, the message is *forwarded* either up or down the tower to the default metaobject. For example, when the base-level object receives a message `M`, it forwards the message to its metaobject by sending a message `[:message M R S]` (`R` and `S` are the reply destination and the sender of `M`, respectively). Conversely, when the forwarding object at level $(n+1)$ receives a message `[:message M' R' S']`, it forwards the message by sending `M'` to the object at level n.

Since forwarding is a simple form of delegation within the tower, the forwarding metaobjects are implemented using the light-weight objects. We call such metaobjects the *light-weight metaobjects*. By employing the light-weight objects, forwarding along the tower need not be true message passing, which would incur great overhead; rather, the message is correctly delivered to the default metaobject by successive invocations of the lambda-closures of the light-weight metaobjects.

Here, the reader may become concerned with the fact that multiple messages could arrive at the object simultaneously, which could be a problem because light-weight objects do not have message queues. Fortunately, it is not a problem since forwarding is programmed to be pure functional, and multiple incoming messages are eventually buffered by the message queue of the default metaobject.

Forwarding augmented with light-weight objects is much more efficient compared to naive implementation of lazy creation of metaobjects. Let us contrast the two approaches and see why: in the latter approach, a metaobject $\uparrow x$ is created when the access to $\uparrow x$ occurs, that is, when the evaluator first evaluates an expression `[meta x]`[19]. Here, suppose that object S is sending a message M to object T, where S already has a metaobject while T does not. (Figure 7(a).)

Since the execution of S is already governed by its metaobject $\uparrow S$, $\uparrow S$ tries to send a message `[:message` M R S`]` (R is a reply destination of the message) to the metaobject of T. Here, the access to the metaobject of T by $\uparrow S$ causes the lazy creation of $\uparrow T$; T then becomes an object that is indirectly executed by $\uparrow T$ (Figure 7(b)). As a consequence, execution of T becomes comparatively slower. In our current approach, messages are directly forwarded to T for faster execution (Figure 7(c)).

We refer to the following notion as the *dynamic progression of degree of reflectivity*[12] — the facility to realize the reflective functionality is progressively made more elaborate as more powerful ones are requested. This is a generalization of the lazy

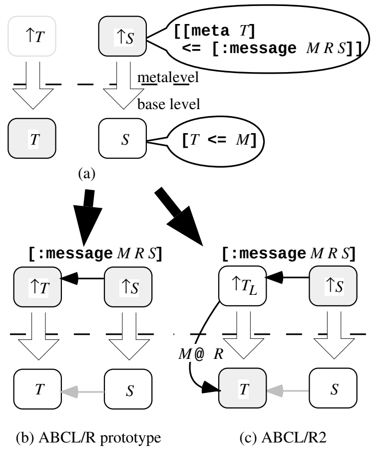

Figure 7: Comparison of the Naive Approach and the Forwarding Approach

creation mechanism, in a sense that not only unnecessary reification is avoided, but also reflective features are restricted, allowing for efficient execution until full-fledged reflective operations become necessary. Such progression is performed automatically by the system in the following way:

- Initially, an executable object does not have its metaobject. (i.e., its metaobject(s) exist only conceptually.)
- When an expression `[meta` x`]` is evaluated, where x does not have its metaobject, a light-weight metaobject $\uparrow x_L$ is created as the metaobject of x. X remains directly executable as mentioned above (Figure 7(c)).
- When a message requesting some reflective operations (a message that does not match to the pattern `[:message` M R S`]`) arrives at $\uparrow x_L$, x is automatically reified, and $\uparrow x_L$ becomes the default metaobject $\uparrow x$. Hereon, the reflective operation is executed by $\uparrow x$.
- The access to the metaobject of $\uparrow x_L$ (i.e., the meta-metaobject of x) merely creates a light-weight metaobject $\uparrow\uparrow x_L$. X still remains to be directly executable by the forwarding mechanism.

The alternative strategy would have been to perform the appropriate reification/reflection of the message at the sender's end (actually, the evaluator), and let it decide the proper level of the receiver to which the message should be sent. This is not desirable, however, for several reasons: (1) it would break the object encapsulation (in the current scheme, the sender does not need to know the status of the individual tower of the receiver), (2) if such reification/reflection was manifest in the evaluator code, it would be complicated, (3) the code for dynamic progression, which requires concurrency control due to simultaneous message arrival, would be inefficient because the sender must explicitly 'lock' the receiver object, and (4) in a distributed implementation, it would incur several round-trip message sends to check the status of the receiver, and then sending the message, whereas our current scheme only requires a single message transmission.

4.7 Compilation of Non-Reifying Objects

Since non-reifying objects sacrifice reflective capabilities for faster execution, the script of non-reifying objects can be almost entirely be compiled into a lambda-closures of Lisp, and be executed without interpretation. The compiled code accepts messages of the form `[:message` M R S`]`, and executes the compiled script matching M. Expressions that are internal to the object and execute sequentially are directly translated into Lisp expressions. Expressions relevant to OOC-computing, such as message sending or object creation, are converted into function calls to the low-level kernel. For example, the expression `[x <= (+ 1 y) @ z]` is converted into a Lisp expression `(send-message x (+ 1 y) z)` where `send-message` is a primitive for message transmission provided by the low-level kernel.

Even though the script of non-reifying object is not subject to reification, the problem is that some messages may be sent to the (non-existing) metaobject(s) of the non-reifying objects due to the abovementioned encapsulation. For this purpose, a message to a metaobject of a non-reifying object automatically creates a light-weight metaobject in the same manner as is with the implementation of individual tower for normal objects. This solves the problem of interfacing non-reifying objects with reflective parts of the system.

4.8 Lazy Creation of Meta-Groups

The construction of the group tower is achieved in finite steps via lazy creation of group managers. In addition, the self-reification code of group kernel objects contains the necessary reification code for the group tower. Below is the overview of how the group tower is constructed:

1. All objects except the group manager object is created with a reference to the group manager object of the group of which it is a member.

2. The group manager is initially created without a reference to *its* group manager, i.e., the group manager of the meta-group. The latter is created lazily by the system when it is referenced.

3. Upon creation of the group manager, other group kernel objects (evaluator, metaobject generator, etc.) are not initially created. Their creation occurs when they are referenced via the group manager; at the same time, the group manager G references its own group ID, which results in the creation of $\Uparrow G$ via 2 (provided it is non-existent). The created group kernel object becomes a member of $\Uparrow G$ by receiving the reference to $\Uparrow G$ from G.

4. Acquiring a reference to the group kernel objects is initially possible only via the group manager (The reference can be freely passed around once it is acquired). Thus, all objects (including the members of user defined groups) of all levels satisfy the condition 1 (except the group manager).

Figure 8 illustrates the creation process of a new group. When the object `a` creates a new group `B` with the `[group ...]` form, the evaluation of this form is sent to the evaluator (the `[:do ...]` message). The evaluator, in turn, sends the `[:new-group ...]` message to the group manager, which is forwarded to the group manager of the meta-group, $\Uparrow$`G` (the `[:new ...]` message). Since the group kernel objects of the new group `B` become members of $\Uparrow$`G`, the metaobject generator of $\Uparrow$`G` must be created, which successively results in the creation of $\Uparrow\Uparrow$`G` as explained in 3. (Note that this object exists for consistency purpose only, and does not execute any code unless higher-level reflective operations are requested.) The newly created metaobject generator `meta-gen'` in turn creates the group kernel objects of group `B`.

5 Performance Measurements

We ran several benchmark programs for performance evaluation of ABCL/R2. Specifically, we tested performance of (1) non-reflective features, (2) the light-weight objects, and (3) the maximum cost of reflective computation.

For comparative purposes, we have performed the same benchmark on our ABCL/1[24] compiler (which does not support reflective features), which is currently being publically distributed. (For details, contact abcl@camille.is.s.u-tokyo.ac.jp.) We have also programmed the same algorithm in C with the Sun Light-Weight Processes (LWP) library, assigning a thread per object. The ABCL/R2 system we employ in this section is not the version on Luna-88k, but instead the pseudo-parallel version running on top of KCL (Kyoto Common Lisp) on SparcStation1+ (except for the lowest-level thread scheduler, the two implementations are identical.). The reason for this is that the current ABCL/1 system only supports pseudo-parallel execution on standard Common Lisp. (The examples in the next section were executed on Luna-88k).

Although the benchmarks were performed for several programs, for brevity the one we present in this paper is the computation of Fibonacci numbers. In the parallel version `parallel-fib`, for each computation of $fib(n)$, two sub-objects that compute $fib(n-1)$ and $fib(n-2)$ are created. In the recursive (sequential) version, for each computation of $fib(n)$, the object `recursive-fib` creates two continuation objects to receive the values $fib(n-1)$ and $fib(n-2)$.

Figure 10 shows the result of the measurement. Some of the implications of these results are as follows:

- For the parallel version, the two corresponding lines in the middle indicate that, ABCL/R2

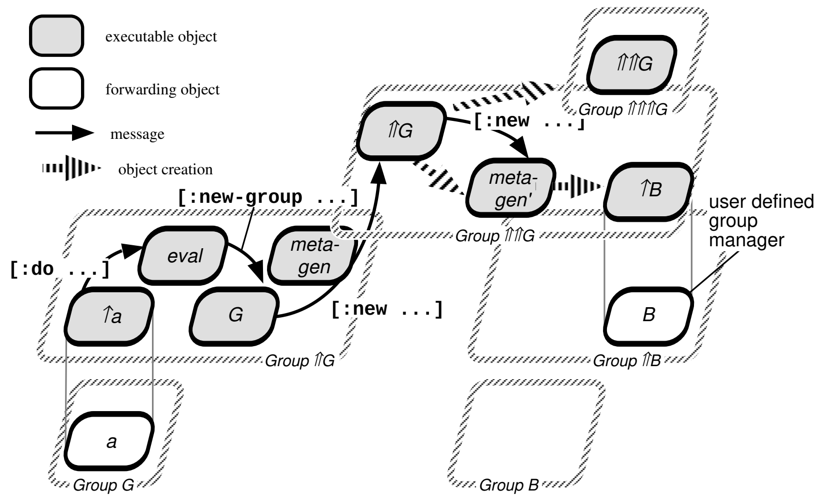

Figure 8: Creation of a New Group

exhibits comparable performance to ABCL/1 for non-reifying objects.

- When light-weight objects are employed (as continuation objects), execution in ABCL/R2 is consistently faster (by approximately 30%). As we have noted earlier, this fact is important because the light-weight objects are heavily created and employed by the evaluator during the execution of scripts, and are also employed (transparently) in user programs.

- Normal objects still pay some cost of reflective execution, but by a factor of less than 10. This is astonishingly smaller compared to ABCL/R, by nearly or over two orders of magnitude: the computation of parallel $fib(12)$ takes over *12 minutes* on ABCL/R executing on an identical software/hardware platform, whereas it takes only 22 seconds for normal objects and 4 seconds for non-reifying objects on ABCL/R2.

- The C + Sun LWP version become drastically slow when the number of object increases, probably due to the overhead of stack allocation and context switching. (In fact, it was not possible to compute beyond $n = 16$ due to lack of memory. We also make a note that the measurement does not include the paging overhead.) Furthermore, non-reifying objects of ABCL/R2 exhibit comparable performance even when the number of objects is small.

Furthermore, by our programming experience thus far, the execution speed of actual programs on ABCL/R2 is often quite comparable to that on ABCL/1, even if reflective operations are employed. The reason for this is that, in practice, the user would employ the mixture of non-reifying objects and normal objects, and attempts to localize the reflective portion of his programs to normal (reflective) objects. As a result, a large portion of the user code runs with non-reifying objects (and light-weight objects), and meta-level execution would be localized to where it is really needed. Even if execution is with normal objects (which have metaobjects), the meta-level execution overhead is at a comparable level to ABCL/1 due to our optimization schemes (unlike ABCL/R, which is more than two orders of magnitude slower compared to ABCL/1).

6 Examples of Reflective Programming in ABCL/R2

Our example is controlling of explosion of parallelism. Programs in Actor-like languages are usually written in a style such that the maximum available parallelism in the algorithm is exploited. However, too much parallelism wastes system resources,

```
[object parallel-fib-gen  ; the parallel version
 (meta-gen non-reifying-meta)
 (script
   (=> :new              ; '!' returns the evaluated
    ![object fib         ; expression
       (meta-gen non-reifying-meta) ; ABCL/R2 only
       (state [reply := nil] [sub-value := nil])
       (script
        (=> [:ans x]
             (if sub-value
                [reply <= [:ans (+ sub-value x)]]
                [sub-value := x]))
        (=> 0 ![:ans 0])
        (=> 1 ![:ans 1])
        (=> n @ R
            [reply := R]
            [[parallel-fib-gen <== :new]
              <= (- n 1) @ Me]
            [[parallel-fib-gen <== :new]
              <= (- n 2) @ Me]))]))]
[object recursive-fib
  (meta-gen non-reifying-meta)  ; ABCL/R2 only
  (script
   (=> 0 !0)
   (=> 1 !1)
   (=> n @ R
       [recursive-fib <= (- n 1)
        @ [cont fib-n-1
             [recursive-fib <= (- n 2)
              @ [cont fib-n-2
                   [R <= (+ fib-n-1 fib-n-2)]]]]]]))]
```

Figure 9: Object Definitions for Fibonacci Numbers

and as a result, has a negative effect on performance. Of course, one could perform user-level programming in order to control the number of objects created, etc.[1], but the resulting user code would have the base-level algorithm and the control algorithm heavily intermixed without proper encapsulation, and as a result, hampers program development, portability, and re-use. Rather, if such control could be encapsulated in the meta-level, not only that the user code need not contain provisions for control, but the same meta-level code could be re-used for a variety of concurrent programs in a portable manner.

In order to control the parallelism, we limit the number of objects created in the system. In particular, if the number of executable objects could always be suppressed so that it is comparable to or little higher than the total number of processing resources in the system, we obtain an ideal balance.

The system is organized as in Figure 11. Each application object embodies within itself a parameter indicating the "degree of progress" in its computation.[3] Such the object is created by the following form:

```
[object object-name
 (group pcontrol)
 (priority p)
 (state ...)
 (script ...)]
```

When an evaluator accepts this form, it sends an object creation message, which contains a "degree

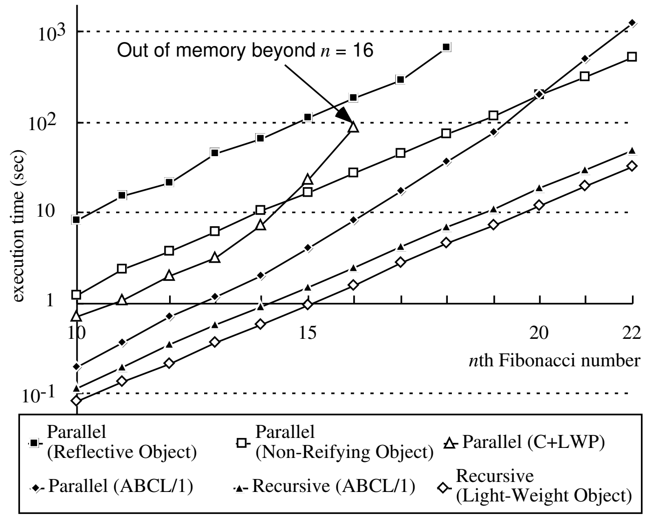

Figure 10: Performance Measurements for Fibonacci Numbers on ABCL/R2 and ABCL/1

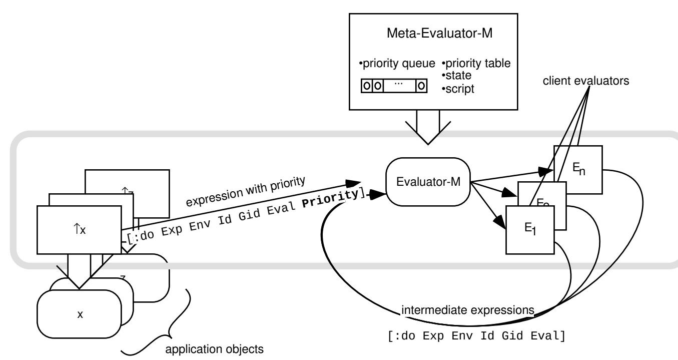

Figure 11: Controlling Explosion of Parallelism

[3] For example, in quicksort it is the length of the list to be sorted.

```
[object Meta-Evaluator-M
 (meta-gen non-reifying-meta)
 (state [pqueue := aPriorityQueue]
        [state := newEnvironment]
        [scriptSet := givenScriptSet]
        [evaluator := anEvaluator]
        [Group-manager := aGroupManager]
        [mode := ':dormant])
 (script
  ;; arrival of a message with priority
  (=> [:message [Tag Exp Env Id Gid Eval Priority]
                Reply Sender]
           ;; enqueues the message in the
   [pqueue ;; prioritalized message queue
    <= [:enq [Priority [[Tag Exp Env Id Gid Eval]
                        Reply Sender]]]]
   (when (eq mode ':dormant)
     [mode := ':active]
     [Me <= :begin]))
  ;; start processing of a message
  (=> :begin
   ;; a message with the highest priority is taken out
   (match [pqueue <== :deq]
     (is [_ [Message Reply Sender]]
       (match (find-script Message Reply scriptSet)
           :
```

Figure 12: Definition of Meta-Evaluator-M

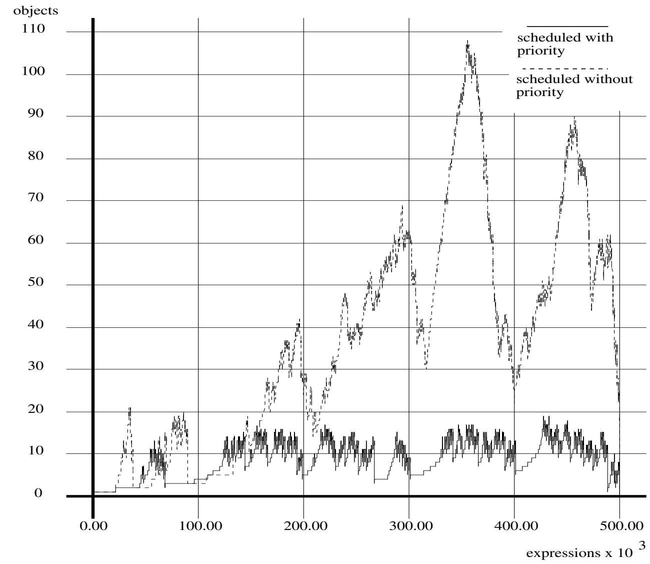

Figure 13: Controlled and Uncontrolled Parallelism for Quicksorting

of progress" parameter p as a priority value in addition to the standard creation messages, to a customized metaobject generator through the group manager pcontrol. Then the metaobject generator creates a customized metaobject with that priority value. As indicated earlier, the computation of an application object at the base-level is performed by the evaluator at the meta-level, which is triggered by a message from the metaobject. This message contains the priority value. Hence, the metaobjects created by the customized metaobject generator send the priority value to the evaluator in addition to the standard parameters. Here, in order to exploit the meta-level programmability of our system, we let the metaobject of the evaluator (Meta-Evaluator-M) prioritize the message according to this parameter. Figure 12 shows the outline of the definition of Meta-Evaluator-M. By prioritizing (evaluation requests of) objects that are expected to terminate their computation faster, the number of executable objects can be suppressed. The evaluator (Evaluator-M) then distributes the task to the client evaluators ($E_1 \ldots E_n$), which represent actual processors.

We applied the above of scheme of parallelism control to two different parallel programs: the parallel version of computation of Fibonacci numbers in the previous section, and a parallel quicksort algorithm. To adapt to different algorithms, the only customization is to declare what to employ as the parameter. We created four client evaluators to coincide with the number of processors that LUNA-88K has (which is 4). Figure 13 is the graph of the executable object count for parallel quicksorting of 1,000 elements (the graph for Fibonacci is essentially the same in shape). The solid line indicates the object count when the parallelism is controlled, and the dashed line indicates no control. The horizontal axis indicates the number of expressions processed by the evaluator. As we can see, parallelism is appropriately controlled via suppression of excessive object creation (which is a user policy) with appropriate meta-level encapsulation (which is a system mechanism). (We present a more elaborate meta-level encapsulation of *Time Warp Scheduling* in the Appendix.)

7 Conclusion

We have presented schemes for implementing OOCR languages efficiently using a target language ABCL/R2. The extended lazy creation scheme of meta-groups and metaobjects reduces the meta-level interpretation overhead. Efficient script execution is achieved by the partial compilation of non-reflective operations, and allowing for the mixture of reflective and non-reflective operations. The light-weight objects provide efficient execution of continuation objects and light-weight metaobjects. The system reduces 'unnecessary reification' via dynamic progression scheme using the light-weight metaobjects and the forwarding mechanism. Self-reification of group kernel objects and compilation scheme of non-reifying objects allow full compilation of object scripts. The basic run-time performance of ABCL/R2, as a result, compares favorably to or sometimes even exceeds that of the ABCL/1 compiler and also conventional threads programming using C + Sun LWP in non-reflective programming, and the overhead of reflective computation is reduced by orders of magnitude compared to our previous OOCR language prototypes. This allows us to achieve practical execution efficiency for typical programs that are mixtures of non-reflective and reflective code,

The implementation of ABCL/R2 described in this paper is running on the TOP-1 Common Lisp[18] on Omron Luna-88k, a shared-memory computer with four 88000 CPUs running Mach. A pseudo-parallel version that runs on Kyoto Common Lisp and CMU Common Lisp was also created. The latter version is available via anonymous ftp on the Internet from camille.is.s.u-tokyo.ac.jp (133.11.12.1) so that researchers all over the world can experience the joys and intricacies of OOCR programming. We hope that it will also serve as a platform for experiments in concurrent programming, e.g., compare several concurrency control algorithms, since such algorithms can usually be encapsulated easily in the meta-level and above as we demonstrate in this paper.

Our next big challenge is to reduce the cost of reflective operations further by the use of more elaborate compilation schemes. Since we have achieved comparable speed to non-reflective version of the language for non-reflective operations, if we could 'collapse' and compile away much of the reflective code, we would be constantly able to achieve comparable (or greater) speed for reflective programs. In particular, the most difficult problem would be how to 'reflect' the dynamically user-customized meta-level code into base-level compilation. For this purpose, we need to concentrate on three research areas:

1. The current version of ABCL/R2 allows too much freedom in the reflective programming of both the individual tower and the group tower. We should instead (1) divide the meta-level into smaller sub-functional parts (e.g., divide the evaluator into sub-evaluators) and (2) devise appropriate Metaobject Protocols[9] that allow only valid customization, thereby giving the compiler more a-priori information.

2. Develop a practical partial evaluation scheme so that the amount of compilable code could be maximized. For example, one could collapse some of the evaluator code into the metaobject so that evaluation request to the evaluator could be eliminated. (The use of partial evaluation in reflection has been suggested in [6, 20, 4], but to our knowledge, no actual reflective languages exist that have actually implemented it.)

3. Integrate an on-line compiler into the system, which is used for dynamic compilation of objects that had been dynamically modified by the user via reflection. For this purpose, various compiler technologies developed for SELF (e.g., [5], among many others) could be applicable, but many other technologies specific to OOCR architectures would have to be developed.

We are also experimenting with ABCL/R2 in programming of many other interesting examples, such as the dynamic optimization, deadlock detection, and concurrent debugging. Such programming experiences would guide us towards more efficient implementation schemes as well as more sophisticated OOCR architectures.

In conclusion, we stress that it is not the language but the language architecture as a whole (that manifests itself with reflection) that provides the *mechanism* for integrating various user *policies* into the system for solving user-specific problems, and, such malleable system architectures are especially valuable for parallel and distributed computing.

Acknowledgements

We would like to thank the members of IBM Tokyo Research Laboratory (TRL) for providing us a parallel version of Common Lisp (TOP-1 Common Lisp) which was ported from TOP-1 (a prototype shared-memory architecture) to LUNA-88K with the helpful aid of Shigeru Uzuhara[4] of TRL.

References

[1] Gul Agha. Concurrent object-oriented programming. *Communications ACM*, 33(9):125–141, 1990.

[2] Christopher Burdorf and Jed Marti. Non-Preemptive Time Warp Scheduling Algorithm. *Operating Systems Review*, 24(2):7–18, April 1990.

[3] Roger M. Burkhart. Reflective functions for the C language. In *Proceedings of ECOOP/OOPSLA '90 Workshop on Reflective and Metalevel Architectures in Object-Oriented Programming*, Ottawa, Canada, October 1990.

[4] Craig Chambers. Towards Efficient Implementation of Computational Reflection. In *Proceedings of the OOPSLA'91 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1991.

[5] Craig Chambers and David Ungar. Making pure object-oriented languages practical. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 1–15, Phoenix, Arizona, October 1991. Published as SIGPLAN Notices 25(11), November, 1991.

[6] Olivier Danvy. Across the bridge between reflection and partial evaluation. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 83–116. Elsevier Science, North-Holland, 1988.

[7] Jim des Rivières and Brian Cantwell Smith. The implementation of procedurally reflective languages. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, 1984.

[4] Currently with Keio University, Japan.

[8] David R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

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

[10] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings of OOPSLA'87*, volume 22, pages 147–155. SIGPLAN Notices, ACM Press, October 1987.

[11] Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In *Proceedings of ECOOP'91*, number 512 in Lecture Notes in Computer Science, pages 231–250. Springer-Verlag, 1991.

[12] Satoshi Matsuoka and Akinori Yonezawa. Metalevel solution to inheritance anomaly in concurrent object-oriented languages. In *Proceedings of the ECOOP/OOPSLA'90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1990.

[13] Jayadev Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, 18(1):39–65, March 1986.

[14] Ramana Rao. Implementational reflection in Silica. In *Proceedings of ECOOP'91*, number 512 in Lecture Notes in Computer Science, pages 251–267. Springer-Verlag, July 1991.

[15] John R. Rose. A Minimal Metaobject Protocol for Dynamic Dispatch. In *Proceedings of the OOPSLA'91 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1991.

[16] Brian C. Smith. Reflection and semantics in Lisp. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 23–35. ACM Press, 1984.

[17] Brian C. Smith. What do you mean, meta? In *Proceedings of the ECOOP/OOPSLA'90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1990.

[18] Tomoyuki Tanaka and Shigeru Uzuhara. Multiprocessor Common Lisp on TOP-1. In *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*, 1990.

[19] Takuo Watanabe and Akinori Yonezawa. Reflection in an object-oriented concurrent language. In *Proceedings of OOPSLA'88*, volume 23, pages 306–315. SIGPLAN Notices, ACM Press, September 1988. (Revised version in [24]).

[20] Takuo Watanabe and Akinori Yonezawa. An actor-based metalevel architecture for group-wide reflection. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages (REX/FOOL), Noordwijkerhout, the Netherlands*, May 1990. also number 489 in Lecture Notes in Computer Science. Springer-Verlag, 1991.

[21] Yasuhiko Yokote. The muse reflective operating system: The concept and its implementation. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1992. To appear.

[22] Yasuhiko Yokote, Fumio Teraoka, and Mario Tokoro. A reflective architecture for an object-oriented distributed operating system. In Stephen Cook, editor, *Proceedings of ECOOP'89*, pages 89–106. Cambridge University Press, 1989.

[23] A. Yonezawa, H. Matsuda, and E. Shibayama. Discrete event simulation based on an object oriented parallel computation model. Technical Report C-64, Dept. of Information Science, Tokyo Institute of Technology, 1984.

[24] Akinori Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System*. Computer Systems Series. The MIT Press, 1990.

[25] Akinori Yonezawa and Takuo Watanabe. An introduction to object-based reflective concurrent computations. In *Proceedings of the 1988 ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, volume 24, pages 50–54. SIGPLAN Notices, ACM Press, April 1989.

A Appendix — Specification of the ABCL/R2 Compiler (Abridged)

Function $\mathcal{C}$ compiles an ABCL/R2 expression to a Lisp form which is processed by the evaluator. The form consists of a pair of tag and data. When the evaluator receives a form `[:compiled` f`]`, the evaluator calls f with the following arguments: the continuation `C`, the environment `Env`, the object id `Id` and the group id `Gid` of the object being processed, and the `Eval` itself (i.e., this is equivalent to `(funcall` f `C Env Id Gid Eval)`.)

The abridged description below does not describe various minor optimizations, such as constant folding. In addition, the notations used in the RHS, `[`e_1 `<=` e_2 `@` e_3`]` and `[`e_1 $e_2 \ldots e_n$`]`, are syntactic shorthands for `(send-message` e_1 e_2 e_3`)` and `(list` e_1 $e_2 \ldots e_n$`)`, respectively.

```
C[[id]] = [:variable id]                    (Variable lookup)
```

```
C[[(e1 <= e2 @ e3)]]                        (Message sending)
=[:compiled
  #'(lambda (C Env Id Gid Eval)
     [Eval <= [:do-evlis [C[[e1]] C[[e2]] C[[e3]]]
                          Env Id Gid Eval]
      @ [cont [v1 v2 v3]
          [Eval <= [:do [:send-past v1 v2 v3]
                        Env Id Gid Eval] @ C]]])]
```

```
C[[[id := e]]]                                  (Assignment)
=[:compiled
  #'(lambda (C Env Id Gid Eval)
      [Eval <= [:do C[[e]] Env Id Gid Eval]
        @ [cont v [Env <= [:set id v] @ C]]])]
```

```
C[[(op e1 e2...en)]]                  (Non-reflective operator)
=[:compiled
  #'(lambda (C Env Id Gid Eval)
      [Eval <= [:do-evlis [C[[e1]] C[[e2]]...C[[en]]]
                 Env Id Gid Eval]
        @ [cont [v1 v2...vn]
               [C <= (op v1 v2...vn)]]])]
```

B Appendix — Meta-level Encapsulation of Time Warp Scheduling

Our previous paper outlined how ABCL/R2 facilitates the *Time Warp* algorithm[5][8] (also known as the Virtual Time scheme, employed in parallel discrete event simulation) to be encapsulated in the meta-level of the user program, and how its scheduling algorithm could be encapsulated in the meta-level of the Time Warp algorithm itself (Figure 14). Scheduling in Time Warp is known to be important, because there could be substantial change in the execution speed due to the difference in number of rollbacks, etc[2].

More specifically, we define a *TimeWarp group*,

[5] Here is a quick overview of the Time Warp algorithm: first, message sends/reception model the events in the simulation. The Time Warp algorithm then serves to maintain the temporal consistency among the events. Consistency management is distributed and optimistic; each object has its own *Local Virtual Time (LVT)* (i.e., there is no global clock), and the messages are timestamped to be compared with the LVT of the recipient. When a conflict is detected, the object performs automatic *rollback* by sending *anti-messages* until it reaches the time just prior to the conflict occurrence.

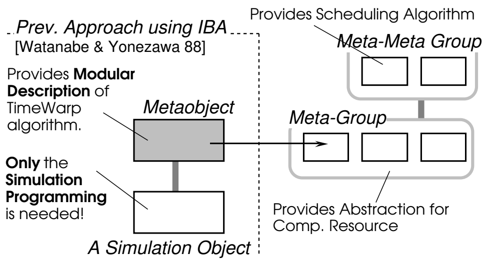

Figure 14: Meta-level Encapsulation of the Time Warp Algorithm

whose members are specialized with their individual metaobjects so that they coordinate in running of the Time Warp algorithm. By specialization of the metaobject generator, the Time Warp algorithm itself is encapsulated in the meta-level, and group membership automatically dictates Time Warp behavior. (The actual definitions of the Time Warp group are given in [11].) Messages sent within the group or to destinations within other Time Warp groups must be of the form:

[*target* `<=` *message*
    `@` *reply-destination* `:vrt` *virtual-send-time*]

For encapsulation of scheduling in the meta-meta-level, we employ a similar scheme to the example given in Section 6. We introduce the *TimeWarp scheduler* object labeled `Scheduler` in Figure 15. It is responsible for controlling the allocation of the computational resource within a Time Warp group. For meta-meta-level encapsulation of scheduling, the scheduler does not directly interact with the evaluator of the Time Warp group, but rather, interacts with the *metaobject* of the evaluator that is customized so that the evaluation request message is first sent to the scheduler. The metaobject then asks the scheduler for the next evaluation job as determined by the algorithm of the scheduler. With this scheme, the same Time Warp algorithm runs irrespective of the presence of/difference in the scheduling algorithm at the meta-meta-level. Furthermore, it would be easy to extend the Time Warp group to add inter-scheduler communication, and/or to transparently adapt to a distributed environment.

Figure 16 is the result of the car-wash simulation[13, 23] on ABCL/R2: cars in the incoming queue are washed by multiple attendants with different washing speed. When (1) no scheduling was performed, and (2) meta-meta-level scheduling was performed in FIFO or-

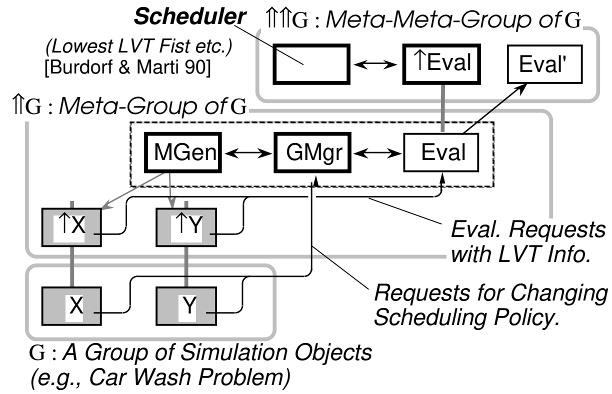

Figure 15: Implementation of the Time Warp Scheduling in the Meta-meta Level

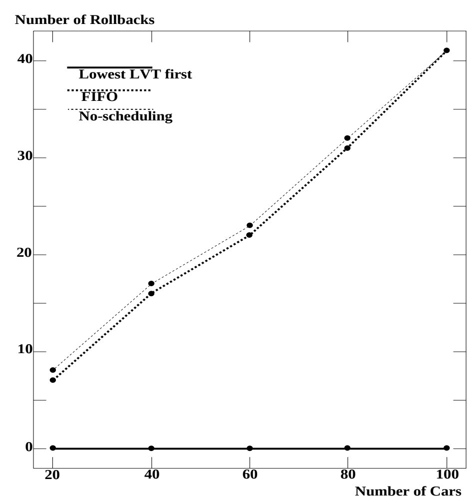

Figure 16: The Result of Meta-meta-level Scheduling of the Time Warp Algorithm applied to the Car Wash Problem

der, the number of rollbacks increased proportionally to the number of cars. When the scheduling algorithm was changed to (3) 'lowest LVT first', the rollbacks were eliminated[6]. Due to this, the execution speed of (3) was consistently several % faster than (1), despite the overhead of execution at the meta-meta-group level.

[6] This seemingly 'ideal' result is probably due to centralized scheduling; when ABCL/R2 is extended to a distributed system, rollbacks should occur (albeit few in number).