STUDY ON IMPLEMENTATION OF AN OBJECT-ORIENTED CONCURRENT REFLECTIVE LANGUAGE

自己反映計算の機能を持つ並列オブジェクト指向言語の実装に関する研究

by

Hidehiko Masuhara

増原 英彦

A Senior Thesis

卒業論文

Submitted to

the Department of Information Science

the Faculty of Science

the University of Tokyo

on February 6, 1992

in Partial Fulfillment of the Requirements

for the Degree for Bachelor of Science

Thesis Supervisor: Akinori Yonezawa 米澤 明憲

ABSTRACT

Computational reflection is the computational activity of a system whose targets of computation are its own structure and its own computation. Programming languages which support reflective capabilities facilitate a modular way of describing the meta-level features such as debugging and resource management — in conventional languages, these features were only available in an ad-hoc fashion. Operational semantics of a reflective language is usually given in terms of an infinite tower of interpreters (the so-called 'reflective tower.') For actual implementation of such languages, however, the tower itself must have some finite implementation. Optimization is also necessary to reduce the overhead of the interpretation. There have been several researches on such issues with sequential reflective languages, but none on concurrent reflective languages. The study of effective implementation of concurrent reflective languages is essential. Since resource management in concurrent computing systems is more dynamic and complex compared to that in sequential systems, the use of reflective facilities for the control of such management from within the language is even more beneficial. In this research, we study the implementation of ABCL/R2, an object-oriented concurrent reflective language. The operational semantics of ABCL/R2 is given in terms of two reflective towers; in order to have finite representations of these towers, we employ 'lazy creation' technique, which postpones the creation of the upper-levels until they are actually needed. We also explore optimizations, such as method compilation, for efficient execution.

論文要旨

計算システムが自分自身の構造・計算過程をモデル化して持ち、自分自身の計算の対象とすることを自己反映計算と呼ぶが、これによって従来ではアドホックにしか扱えなかったデバッグ・資源管理等の機能が、計算システム自身でより整合的に記述できる。自己反映的言語の意味モデルの多くは、無限階層のインタプリタ（リフレクティブタワー）によって与えられているが、実際の実装では、リフレクティブタワーを有限の範囲で表現しなければならない。また、インタプリタによる実行モデルから来るオーバーヘッドを減らすための最適化が必要となる。逐次言語における自己反映計算の機能を持った言語の実装に関しては、いくつか研究がなさているが、並列言語についてのものはない。ところが、並行計算システムでは、逐次システムにくらべ、スケジューリングや負荷分散等の動的でより複雑な資源管理が要求されるため、ユーザが言語レベルでこれらの制御を記述できることは有用である。これらの理由から、自己反映的並列言語の効率の良い実装は不可決であり、本研究では自己反映計算の機能を持った並列オブジェクト指向言語 ABCL/R2 の実装について扱う。リフレクティブタワーによって与えられている ABCL/R2 の実行モデルを有限の範囲で構築するために、上位の階層は必要になるまで作らない 'lazy creation' を行う。また、効率の低下に対しては、オブジェクトのメソッドのコンパイルをする等の最適化を可能にする設計を行う。

Acknowledgements

I am grateful to Professor Akinori Yonezawa for recommending me to study the implementations of the concurrent languages, and giving me many suggestions and comments. I would also like to thank Satoshi Matsuoka and Takuo Watanabe for many pieces of advice on the study of the reflection. Members of IBM Tokyo Research Laboratory offered us the platform of our implementation; the source code of the parallel version of Common Lisp running on the IBM TOP-1. I would like to thank them, especially Shigeru Uzuhara. Finally, I thank the members of Yonezawa Laboratory for the useful suggestions and the encouragements.

Table of Contents

Acknowledgements i

List of Figures iv

List of Tables v

1. Introduction 1

2. Related Work 4

2.1 3-Lisp 4

2.2 Brown and Blond 5

2.3 CLOS Metaobject Protocol 6

2.4 ABCL/R 6

2.5 Other Reflective Systems 7

3. Object-Oriented Concurrent Reflective Languages 8

3.1 Introduction to the Reflection 8

3.2 Object-Oriented Concurrent Reflective Architectures 9

3.2.1 Individual-Based Architecture 10

3.2.2 Group-Based Architecture 10

3.2.3 Hybrid Group Architecture 12

3.3 Semantic Model of ABCL/R2 12

3.4 Problems of Implementation and Efficiency 15
3.4.1 Breaking the Meta-Circularity Efficiently 15
3.4.2 Efficiency versus Reflective Capabilities 16

4. Our Approach **18**
4.1 Overview of the System 18
4.2 Low-Level Structure 20
4.3 Compilation 22
4.4 Light-Weight Object 24
4.5 Group Tower Construction 27
4.6 Individual Tower Construction 27
4.7 Non-Reifying Objects 29

5. Performance Measurements **32**
5.1 Non-Reifying Objects 32
5.2 Light-Weight Objects 34
5.3 Costs for Reflective Computation 35

6. Example of Reflective Programming **37**

7. Conclusion and Future Work **41**

References **42**

Appendix

A. Meta-Circular Definition of ABCL/R2 **45**

B. Code for Suppressing Concurrency Index **49**

List of Figures

3.1 Individual-Based Architecture in ABCL/R 11

3.2 Meta-Architecture of ACT/R 11

3.3 Reflective Architecture of ABCL/R2 17

4.1 Overview of the System 19

4.2 Forwarding Mechanism 20

4.3 Low-Level Scheduling Mechanisms 21

4.4 Definition of the Structure of an Object 22

4.5 Creation of the Objects at Meta Group 30

4.6 Sending a Message to a Conceptual Metaobject 31

5.1 Object for Testing Message Transmission 33

5.2 Object for Fibonacci Numbers (parallel version) 34

5.3 Object for Fibonacci Numbers (sequential version) 34

6.1 System Suppressing Concurrency Index 38

6.2 Concurrency Index for *fib*(10) 39

6.3 Concurrency Index for Quick Sorting 40

List of Tables

5.1 Execution Time for Message Transmissions 33

5.2 Execution Time for the *fib*(n) 33

5.3 Execution Time for Fibonacci Numbers 35

5.4 Comparison of 'Standard' Objects with Non-Reifying Objects . . . 36

Chapter 1

Introduction

Reflection is the process of reasoning about and acting upon the system itself[10, 9]. In conventional languages, the system computes upon the data which is external to the system. In reflective languages, the system computes upon not only the external data, but also the data which is internal to the system itself. In other words, reflective systems provide the ways of inspecting upon/modifying the data that implement the systems. Many types of languages that supports reflective computation have been proposed[5]. Those languages (so called *reflective languages*) are beneficial because they are able to add various programming language facilities from within the language itself. Reflection is a useful notion not only to languages. There are several studies that apply the notion of reflection to construct the flexible systems such as operating systems[14], and window systems[8].

In concurrent languages, the reflective computation is even more beneficial because[12, 16]: A concurrent system has many aspects that should be customized according to the system organization or the nature of the application programs, and those aspects are controlled only by exceptional and ad-hoc fashions in ordinary languages. On the other hand, a concurrent language, which supports reflective computations, is able to control such aspects dynamically. Thus, the reflection in

concurrent languages is more powerful and useful.

However, it is difficult to have an efficient implementation of reflective languages. Since reflective languages should have abstracted data that is mutually connected with the language's implementation: so called *Causally-Connected Self-Representation(s)* (CCSR). A semantic model of a reflective language is usually given by the infinite tower of meta-interpreters. For this reason, two problems should be solved to implement a reflective language: 1) How to break safely the meta-circularity in the semantic model. 2) How to decrease the overhead of interpretative execution, which results from the language's semantic model.

As far as we know, there have been no study on efficient implementation of concurrent languages with reflective computation. Therefore, this study aims at solving these problems on the concurrent languages supporting reflective computation. Particularly, we study the implementation of ABCL/R2[7] with parallel version Common Lisp on OMRON LUNA-88K, which has the shared-memory architecture. The language ABCL/R2 is an *Object-Oriented Concurrent Reflective* (OOCR) language, and is based on *hybrid group architecture* that consists of two kinds of reflective towers: *individual tower* and *group tower.*

For efficient execution, a script is compiled, where operations expected to be reflective are translated into explicit message passing for the interpretation, and the other operations are directly translated into Lisp expressions. The light-weight object, which has a lambda-closure instead of a message queue, provides efficient creation and execution for temporary objects. To break the meta-circularity, we construct the meta-level of a group *lazily* — the meta-group of a group is not created until the some object is created at the meta-level to that group. The metaobject for an object is also created *lazily.* Furthermore, a metaobject is implemented by a light-weight object until reflective operations are requested. Such

the object prevents unnecessary reification; the light-weight metaobject allows that a message sent to a base-level object from another metaobject is forwarded directly without causing reification. In the ABCL/R implementation, such the message causes reification.

In this study, we limit the scope of "efficient implementation" around reflection, and traditional optimizing techniques may not be considered.

The rest of this paper is organized as follows: Chapter 2 describes previous studies that implement sequential/concurrent reflective languages. In chapter 3 the object-oriented concurrent reflective architecture is briefly introduced, and implementation problems with semantic model of ABCL/R2 are pointed out. Chapter 4 discusses our approach to the efficient implementation, and chapter 5 gives the results of our performance measurement and chapter 6 gives an example of reflective programming that shows favorable result. Chapter 7 concludes this thesis and mentions the future work.

Chapter 2

Related Work

In this chapter, previous studies, that are related with the implementation for reflective languages, are reviewed. There have been several studies of the implementation of the sequential reflective languages. However, unfortunately, there have been few studies of the efficient implementation of concurrent reflective languages, as far as we know.

2.1 3-Lisp

A schema of the efficient implementation for sequential reflective languages is described by des Rivières and Smith[3]. This schema is based on *level-shifting processor,* and showed an implementation of a particular dialect of Lisp called 3-Lisp[10]. In his schema, ordinary (non-reflective) expressions are executed by the *implementation processor* (IP) that is a real, active processor, not a program for a processor. When IP encounters a reflective expression, implicit state of IP is collected, then the expression is executed as if a program for a processor has been running with the state that is equal to the one collected from IP. (In other words, IP *reifies* itself into the state for a program.) Once the execution of the expression is ter-

minated, IP *reflects* the explicit state on its internal state, and executes rest of the expressions directly. This method is optimal because the number of reification is minimized and the IP never executes the program that generates a behavior equivalent to IP.

Since one or more activities are running simultaneously in a concurrent language, it is difficult to obtain a state of the system at one instant. Consequently, Smith's schema cannot be directly applied for the concurrent reflective languages.

2.2 Brown and Blond

Wand and Friedman described a semantic model of Brown, that is a sequential reflective language like 3-Lisp[11]. Since the description is based on the denotational semantics, this model is also an implementation in Scheme. The model gives the semantic account of reflective towers without the use of towers, using *metacontinuations.*

A metacontinuation is a data structure that represents the state of the upper interpreter, and that of the tower above it. The reification process is that of packing the state of the interpreter into the metacontinuation, and the reflection process is that of re-installing the state of the interpreter. Though this is infinite structure, it can be constructed in finite steps (by fixpoint operator).

Blond[2], like Brown, is a sequential reflective language, whose model is written in denotational semantics. Contrary to Brown, Blond first assumes the tower of interpreters, then makes it reflective.

2.3 CLOS Metaobject Protocol

PCL (Portable Common Loops)[4] is one of a prototype CLOS (Common Lisp Object System) implementation .

In the semantic model of PCL, the class system is implemented by the PCL's class object, and method invocation steps are described by the PCL's methods. For example, a class named `standard-class` defines the data structure for the classes, and a method named `compute-class-precedence-list` computes a class precedence list to determine what method should be applied for the an object. Thus the reflection in PCL is operated as follows: (1) create a subclass of `standard-class` and (2) re-define some methods (such as `compute-class-precedence-list`) for the new class. Then, the method invocation steps to an instance of the new class are held by the user-defined methods.

PCL has practical efficiency because: (1) Since its reflective capabilities are limited to the operations related with object-oriented facilities, the other operations are executed with enough efficiency. (2) Changes to the meta-level procedures are — as the method invocation steps are divided into many methods in meta-level, the amount of methods, that should be modified for some reflective operation, can be small, and the other steps are executed by the default and highly optimized methods.

2.4 ABCL/R

ABCL/R is an object-oriented concurrent reflective language[12]. Its semantic model is described by the tower of metaobjects in which each metaobject interprets an object in lower level, and this model has good similarity to that of ABCL/R2. There is a prototype implementation of ABCL/R in ABCL/1[15]. A key to the

ABCL/R implementation is the "lazy creation" technique — the metaobject is created when the metaobject is accessed.

Our implementation uses a technique like this to build the reflective towers of ABCL/R2. Chapter 4 discusses this subject.

2.5 Other Reflective Systems

Besides programming languages, several reflective systems have been proposed.

Silica[8] is a window system based on reflection. In Silica, the data structure that implements a window can be changed at meta-level. For example, a new window structure that contains sub-windows can be defined from user programs. The reflective architecture of this system has only two levels; the base-level (the window object) and the meta-level (the implementation of the window object). In other words, this system is not meta-circular.

Muse[14] is an operating system, which is based on reflection. Execution of an object in Muse is supported by the objects, such as memory managers and device drivers, in the meta-level. Though the reflective architecture of Muse is the meta-circular one, the prototype system is implemented by limiting the number of meta-levels; objects at meta-meta level are representing the machine-dependent functions.

Chapter 3

Object-Oriented Concurrent Reflective Languages

This chapter describes the background of our study. First, an introduction to the reflection and a rough description of the OOCR architectures is shown. Then the semantic model of ABCL/R2 — our implementation target — is discussed in more detail. Finally, problems for implementing such language are explained.

3.1 Introduction to the Reflection

Reflective languages can access the abstracted data that represents the implementation of the system itself. For example, in traditional languages, a user program has no way to access its executing environment (data storing the binding information) except for the pre-defined methods. This is because the environment is implementation-dependent for the traditional languages. On the other hand, sequential reflective languages (such as 3-Lisp and Brown) can define reflective procedures to access such data. A reflective procedure is invoked with the three arguments; a processing expression, an environment and a continuation for the expression. The programmer can describe explicit access (introspection/modification)

to the environment through these arguments. In other words, reflective languages provide a modular way to introspect/modify its internal state, or modify its course of processing, and these subjects are considered to be implementation-dependent in traditional languages.

Such abilities of reflective systems are useful to construct implementation independent debuggers, to describe communication between outside world, and to providing opportunity for dynamic optimization. Recently, several practical applications are developed such as PCL (a CLOS implementation that has reflective abilities) [4]. Not only the programming languages, there are also attempt to use the notion of the reflection for constructing the flexible system.

To provide the access to its internals, a reflective language has data that represents meta-level structural and computational aspects of itself. Such data is called the *Causally-Connected Self-Representation(s)* (CCSR). The CCSR must have following features: When the internal state of the language system is changed, the change affects CCSR, and when the modification on the CCSR (from user program) is *reflected* to the actual computational state of the user program.

To contain the CCSR, semantic model of reflective languages is often given by the infinite tower of meta-interpreters (so called reflective tower). In such model, a CCSR of some level is the explicit implementation data at the meta-level of that level. Therefore, the access to the CCSR of a level is taken place in the meta-level.

3.2 Object-Oriented Concurrent Reflective Architectures

Recently, the benefits of computational reflection in concurrent languages have been claimed[6]. This is because a concurrent system embodies multitudes of aspects that do not arise in sequential computing: scheduling, communication,

load-balancing, and so on. Since concurrent systems are much more complex and more immature as compared to sequential ones, it is favorable for constructing modular system and experimenting algorithms to control those aspects from within the languages. Reflection in concurrent languages is beneficial to encompass such aspects within the programming language framework.

Several languages that have the *Object-Oriented Concurrent Reflective* (OOCR) architecture are proposed: ABCL/R[12] is based on the *individual-based architecture,* ACT/R[13] is based on the *group-wide architecture,* and the evolution of ABCL/R, the language ABCL/R2[7] is based on the *hybrid group architecture.* Below, characteristics of each architectures are discussed by examples.

3.2.1 Individual-Based Architecture

An example of this architecture is ABCL/R[12]. Each object in the system has its own metaobject $\uparrow x$ that governs its computation and can be accessed with a special form `[meta` x`]`. The structural aspects of x — a set of state variables, a set of scripts, a local evaluator, and a message queue — are part of state variables of $\uparrow x$. The metaobject $\uparrow x$ also has its own metaobject $\Uparrow x$ and so on (Figure 3.1).

The arrival of a message M at object x is represented as an acceptance of the message `[:message` M R S`]` at $\uparrow x$, where R and S are the reply destination and the sender, respectively. Reflective computation in ABCL/R is through message transmissions to its metaobject and other objects in the tower.

3.2.2 Group-Based Architecture

An example of this architecture is ACT/R[13]. In this architecture, the behavior of an object is not governed by a single particular metaobject; rather, the collective behavior of a group of objects is represented as the coordinated actions of a group

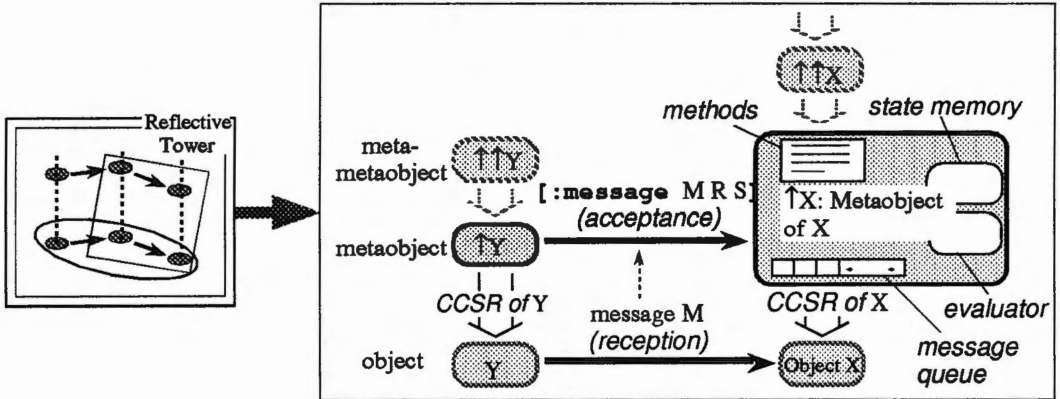

Figure 3.1: Individual-Based Architecture in ABCL/R

of meta-level objects, that constitute the *meta-group* (Figure 3.2).

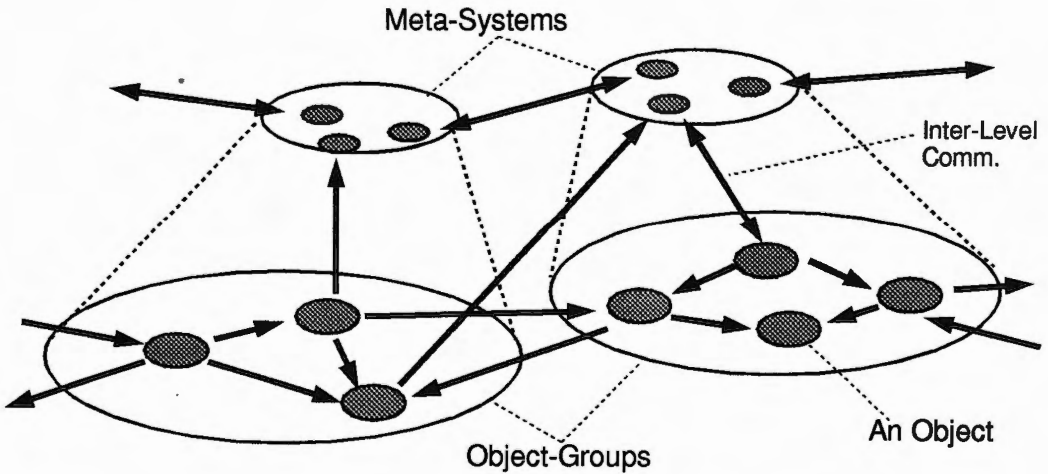

Figure 3.2: Meta-Architecture of ACT/R

The essence of this architecture is the lack of metaobjects, because the behavior of a single object is realized at the meta-level by coordinated action of multiple meta-level objects. Notice that a change to the meta-level object affects to all base-level objects in the group.

3.2.3 Hybrid Group Architecture

In real-life, concurrent systems have various resources such as computational power, communication, storage, I/O, etc., and the availabilities of these resources are limited; for example computational resource is limited by the number of the CPUs in the system. To formulate the management of such limited resources at the programming language level, *Hybrid Group Architecture*(HGA) has been proposed[7]. In languages based on HGA, such "limited resources" are shared at the meta-level as the objects, while each object has its own reflective tower.

The language ABCL/R2 — the target language of our implementation — is based on the HGA. The semantic model of ABCL/R2 is discussed in the next section.

3.3 Semantic Model of ABCL/R2

This section describes the features and semantic model of ABCL/R2.

The key features are follows:

- Heterogeneous Object Group and Group Shared Resources
- Meta-groups and Individual/Group Reflective Towers
- Non-reifying Objects (objects that do not have metaobjects)

An object in ABCL/R2 is always belongs to some *group*. Objects in some group share computational resources as *group kernel objects* at meta-level. Groups can be created dynamically, and the group creation process is given by a concrete metacircular definition with ABCL/R2. As a result, not only that we have the tower of metaobjects, but we also have the tower of *meta-groups*. There are some distinctions between these two kinds of towers:

- The *individual tower* of metaobjects mainly determines the structure of the object, including its script. Thus, 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, the changes to have different behaviors of the same script are in the domain of the group tower.

Figure 3.3 illustrates the reflective architecture of ABCL/R2.

The structure and the computation of a group are defined at the meta- and higher levels of the group, by the group kernel objects. In figure 3.3, objects labelled "Eval," "Metaobject Generator," and "Group Manager" are the standard group kernel objects. The characteristics of these objects are follows [1]:

Group Manager: The group manager represents and 'manages' the group. Also, the identity of a group is that of the group manager object. Following form is an example of the group creation:

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

When this form is evaluated, a new group named `concurrency-controlled-group` is created. At the meta-level, a group manager with two group kernel objects is created: `priority-meta-gen` as a metaobject generator and `priority-queue-eval` as an evaluator.

A new object can be created at a particular group by explicit designation of the group in the object creation form:

---

[1] Complete definition is appeared in appendix A as the meta-circular definitions.

```
[object x
  (group concurrency-controlled-group)
                   ⋮
```

The evaluation of this form creates an object named `x` at the group `concurrency-controlled-group`.

Metaobject generator: An object creation process is realized at meta-level as the creation of a metaobject. A metaobject generator is the object who creates the metaobject at the meta-level. When a form `[object ...]` is evaluated, a message `[:new` *StateSpec LexEnv ScriptSet Evaluator GMgr*`]` is sent to the metaobject generator[2].

A metaobject created by a standard metaobject generator has four state variables: a message queue, a set of scripts, a set of state variables, and a pointer to the evaluator. When the metaobject receives a message `[:message` *M R S*`]`, the metaobject put the message into the queue object in its state variable. Then, the metaobject searches a script matches to a message on top of the queue, and sends a message to the evaluator for executing the script.

Evaluator: The computational resource, shared by the group members, is the evaluator. An evaluator governs the script execution for the members of the group. Its typical behavior is as follows: it receives a message `[:do` *Exp Env Id Gid Eval*`]` with reply destination *C*, then evaluates an expression *Exp* under the environment *Env*, then sends the result to *C*. The arguments *Id* and *Gid* is used for references of the pseudo variables `Me` and `Group` that denotes

[2]More precisely, the evaluator sends a message `[:new` *StateSpec LexEnv ScriptSet*`]` to the group manager, and the group manager sends to the metaobject generator adding *Evaluator* and *GMgr* to the message.

the object itself and the group of the object, respectively. The argument *Eval* is for executing sub-expressions generated during the execution of some expression. That is, the evaluator can delegate the execution of the sub-expressions to an other evaluator.

In ABCL/R2, the user can create an object that runs more efficiently at the sacrifice for the loss of reflective capabilities. Such object is defined by the following form:

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

When this form is evaluated, a non-reifying object named `x` is created. The behavior of `x` is almost same as the one of a standard object created without the designation `(meta-gen non-reifying-meta)`. The difference is that reflective operations are disallowed.

A the non-reifying object does not have its metaobject. Consequently, the extensibility is lost; however, non-reifying objects execute much more efficiently compared to the standard ones.

3.4 Problems of Implementation and Efficiency

3.4.1 Breaking the Meta-Circularity Efficiently

The semantic model of ABCL/R2 is given by the hybrid group architecture. To realize the architecture, the implementation must obey following features:

- Modifications through a metaobject — reflective operations on an individual tower of an object — should be reflected only to the object.

- Modifications through the group kernel objects — reflective operations on the group tower — should be reflected to all members at the base level of the group tower.

Moreover, the actual system must be created by finite steps. Consequently, a problem is arisen: how to break the meta-circularity of the semantic model efficiently.

3.4.2 Efficiency versus Reflective Capabilities

Our interest is the implementation way that has practical efficiency. The semantic model considers the efficiency by introducing the non-reifying object, and the user categorizing objects into reflective or not-reflective. However, it is also important that the standard objects (not non-reifying objects) have enough efficiencies.

There are opportunities for the efficiency of the standard objects to apply such categorization to the operations of standard objects' executions. The operations that appear in the execution of an object can be classified into two groups: the ones we have interest of reflection; for example, operations message transmission and object creation. The other ones we have no interest to be reflective; for example, parsing and arithmetic operations. It is efficient that the latter types of operations can be executed without interpretation like ones in a non-reifying object.

The second problem is: how to translate an expressions into a code where the some operations are indirectly executed, and others are directly executed.

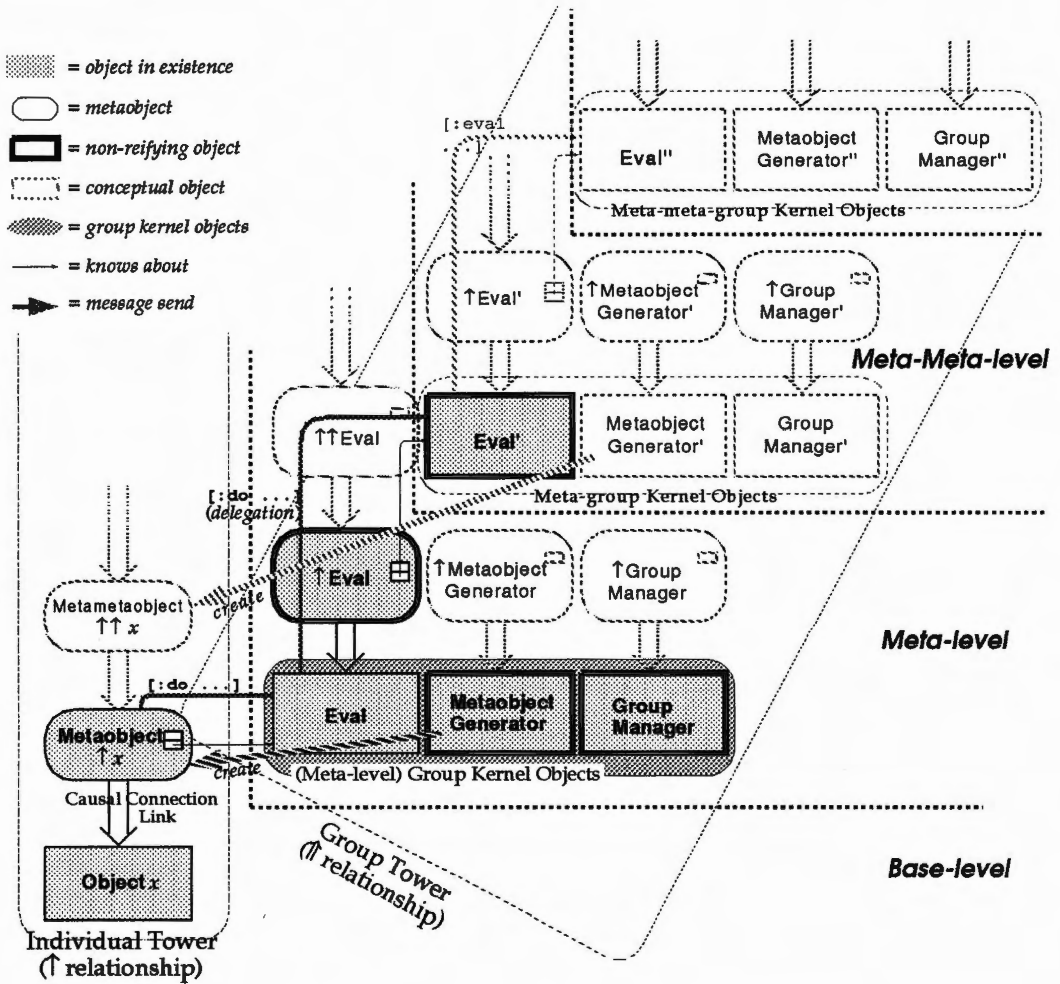

Figure 3.3: Reflective Architecture of ABCL/R2

Chapter 4

Our Approach

A parallel implementation of ABCL/R2 is build in the multi-thread version of Common Lisp, which is running on OMRON LUNA-88K, the shared-memory computer.

In this chapter, the implementation design of our system is shown.

4.1 Overview of the System

Figure 4.1 is an overview of the system. Each box in the figure represents an object, which can be a target of message sending. A "conceptually existing object" do not exist initially, but the access to such the object causes a creation of an actual object (so called *lazy creation*). To maintain the causal-connection between object and metaobject, a individual tower has at most one *executable object.* An executable object is executed directly by Lisp functions, and scheduled when it has any message to process (this state is called to be *active*).

Rest of objects in the individual tower are not executable, but the messages to such the object are handled by the *forwarding mechanism.* The forwarding mechanisms are as follows:

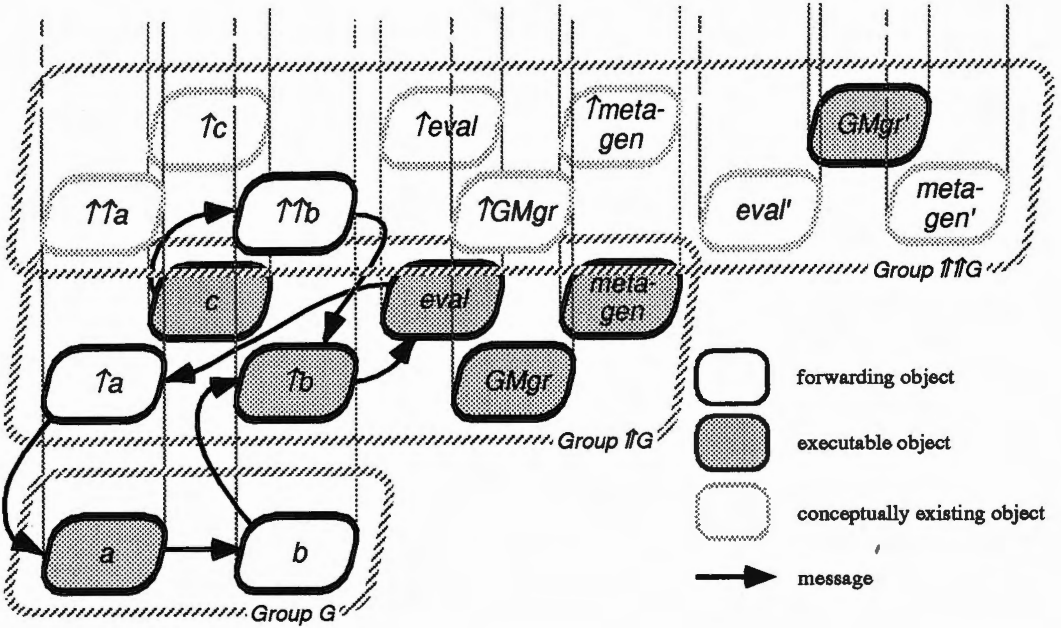

Figure 4.1: Overview of the System

- When a message is sent to an object O in lower than an executable object, O forwards the message by sending a "message arrival" message to its metaobject. (Figure 4.2–a)

- When a "message arrival" message is arrived to an object O' in upper than an executable object, O' sends the message part of the arrival message to its denotation. (Figure 4.2–b)

It might sound strange if a non-executable object *executes* the forwarding mechanisms. Since the forwarding is held by the message sender, not by the message receiver, however, it is not necessary to schedule non-executable object.

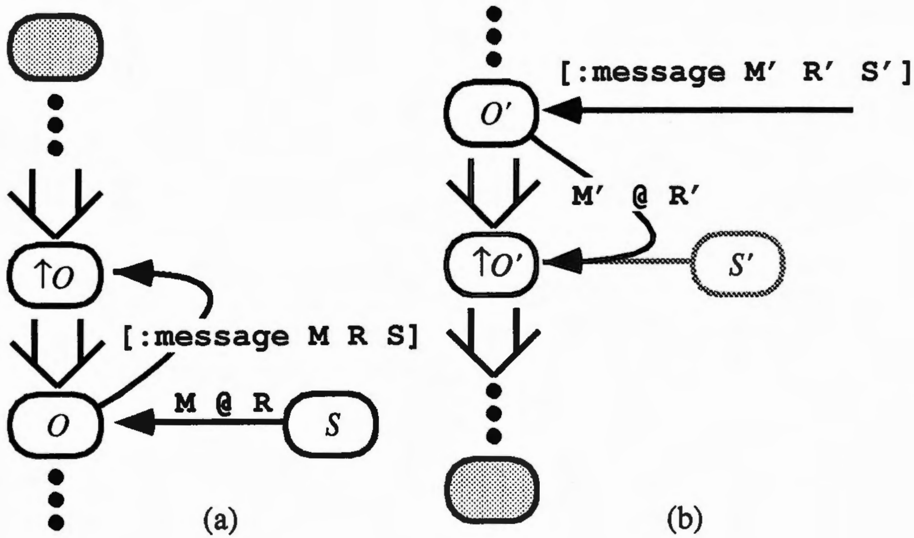

Figure 4.2: Forwarding Mechanism

4.2 Low-Level Structure

Before proceeding, let us describe the low-level structure of the system. Figure 4.3 illustrates the scheduling mechanisms for executable objects. An executable object is implemented by a Common Lisp's structured data. (Figure 4.4.) The entry port is a message queue for the object. "Thread" is a control activity for Lisp programs, and our system assumes a fixed number of threads for executing the executable objects[1] When a message is sent to an idle object, the object is turned to be active, and it is put into the waiting queue. Then, a thread is assigned to the object, and the thread executes a Lisp function that corresponds to a script for the message. After termination of the function, the object returns to the waiting queue when it is still active (i.e., the port is not empty), or it becomes idle when it has no more message (i.e., the port is empty).

[1]The system can work in a pseudo-parallel manner in conventional Common Lisp, which is regarded as a single-thread Lisp. However, some restrictions for non-reifying objects are placed.

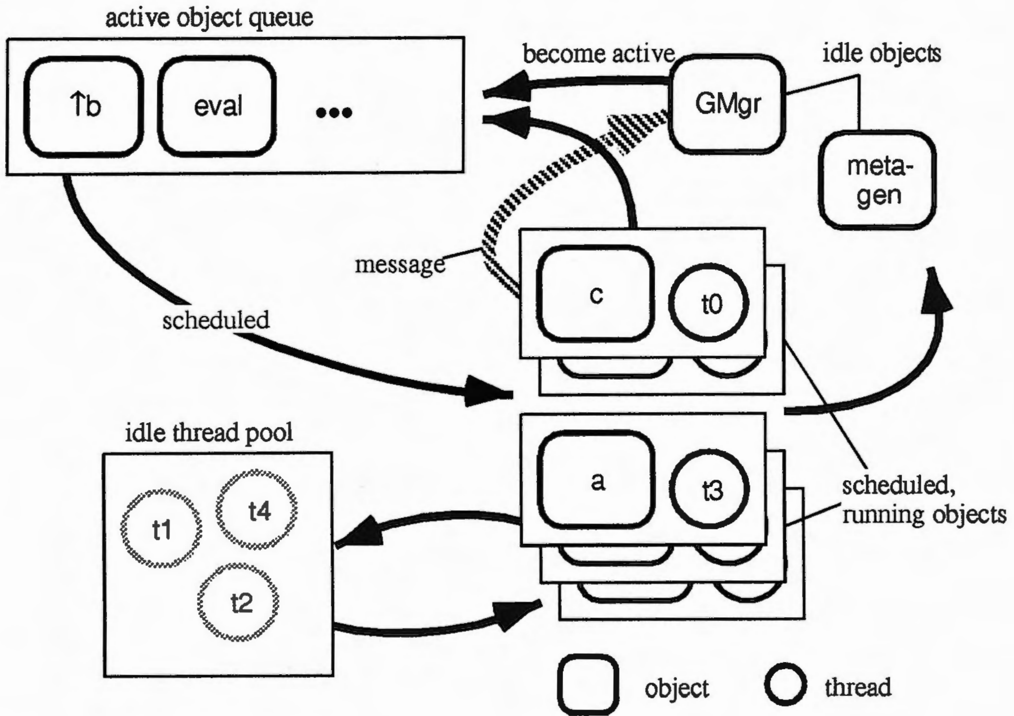

Figure 4.3: Low-Level Scheduling Mechanisms

Default group kernel objects (the group manager, the metaobject generator, and the evaluator) and metaobjects created by the default metaobject generator are also the executable objects. As if a non-reifying object, such an object has a Lisp function that gives equivalent behavior to the meta-circular definitions. (The meta-circular definitions for default group kernel objects appear in appendix A.) Such an object also has a set of compiled reflective script of meta-circular definitions for the *self-reification*: When a reflective operation is requested to the metaobject of such the object, the metaobject becomes a default metaobject with the compiled script as a state variable.

```
(defstruct object-data
  port          ; message port
  meta          ; metaobject
  den           ; denotation
  gid           ; (meta) group manager
  local         ; object's local information
  type
  status)
```

Figure 4.4: Definition of the Structure of an Object

4.3 Compilation

As is mentioned in section 3.4, our interest in reflection of ABCL/R2 is focused on the operations related to the object-oriented concurrent computations, such as scheduling and resource management. In other words, we have few interests in the reflection of the operations related only for sequential computations. Thus, we propose a schema for efficient execution: While operations that are not expected to be reflective are compiled, other operations are reflective.

In this schema, operations are categorized into *reflective operations* and *non-reflective operations,* then:

- Consecutive expressions representing non-reflective operations are translated into a Lisp function.
- The evaluator has a script for invoking the Lisp function for non-reflective operations. Thus, non-reflective operations are initiated by sending a message to evaluator.
- The evaluator has scripts that execute reflective operations, and expressions representing a reflective operation is translated into a message for the evaluator.

- When expressions representing a reflective operation are appeared in ones representing non-reflective operations, a code, that sends a message to evaluator explicitly, is embedded into the lambda-closure for the non-reflective operations.

Though what operations should be reflective and what should not be is arbitrary choice, we choose following operations to be reflective:

- Reference to the variables (including the references to lexical environments, and pseudo variables '`Me`' and '`Group`')
- Message sending (either now type or past type)
- Object creation
- Group creation
- Sequential executions of list of expressions
- Sequential evaluations for list of expressions

The last two operations are 'sugar' for the meta-level programming.

The following example shows how the expressions are compiled:

The expression `[x <= (* y 2) @ z]` means to send the value of `(* y 2)` to the value of `x` with its reply destination to be the value of `z`. As the reflective operations in this expression are message sending and variable references, the compiler generates a code like followings:

```
#'(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]]])
```

Where the arguments `C`, `Env`, `Id`, `Gid`, and `Eval` mean the continuation, the environment, the object ID, the group ID for the object, and the evaluator for the sub-expressions, respectively. The tag `:do-evlis` means the evaluation of list of expressions.

Notice that the multiplication `(* y 2)`, which is not reflective operation, is embedded into the compiled code, while the reference to the variable `y` is achieved by sending an expression `[:variable 'y]` to the evaluator.

4.4 Light-Weight Object

In ABCL/R2, many objects are created at meta-level during the execution of an expression. However, most of those are temporary and have simple functions (hereafter, we call such objects *simple objects*) — for example, a continuation for an expression, or a reply destination for a now type message passing — and constructing these objects as same as complex objects causes considerable overheads: (1) A message for an object is put into a message queue, but the simple objects are created for only processing single message. Therefore, creating a queue and the steps for queueing should be the overhead. (2) Compared with the complexity of the processes for standard objects, that of the processes for simple objects are fine-grained. Thus, the system mostly may devote its computation for the scheduler with ordinary scheduling methods.

To decrease these overhead, we implement such a simple objects as a *light-weight object*. A light-weight object has same structure as an ordinary (heavy) object, except for the message queue and the state variables. The most distinct

feature is that instead of the message queue, light-weight object has a Lisp function, that is translated from the scripts of the object. When a message is sent to a light-weight object, the sender of the message executes the function with the message, instead of queueing the message.

The activation and execution steps for light-weight objects are as follows:

1. An object S starts sending a message M to a light-weight object L.

2. S gets a function from the slot for L's message queue, and invokes function that is stored L's message queue slot.

3. The activity for S executes the L's script with its argument M.

4. When the function execution terminates, S resumes its own execution.

Notice that the execution of the message sender is suspended until the termination of the execution of the light-weight objects. This means that the use of light-weight objects does not increase concurrency, while decreasing the overhead of scheduling. On the other hand, an ordinary object is activated by following steps:

1. An object S starts sending a message M to an object O.

2. S gets O's message queue, and puts M into it.

3. O is put into the active object queue if O has been dormant.

4. Then a thread picks O up from the queue, and starts execution for O.

5. A message M from the O's message queue is processed.

6. A script corresponding to M is executed.

A light-weight object has no state variables, therefore, it is created more efficiently than an ordinary object. This is favorable property for the temporary, and dynamically created objects.

However, light-weight objects have several restrictions:

- No now type messages — consider the following situation: When a light-weight object L has a script that causes a now type message to an object O, a past type massage form O to L causes a deadlock. This is because L waits the reply from O, and O waits the termination of L's execution. (Notice that a now type message to another light-weight object does not cause deadlock.)

- No state variables — light-weight object may be executed simultaneously by one or more objects because it has no message queue. Thus, the access to the state variables without mutual exclusion may lead unexpected behavior.

Despite these restrictions, many types of objects can be constructed with the light-weight object; the light-weight metaobject that appears in section 4.5 is good example.

Our implementation opens up this light-weight object for application programmers. An expression

```
[cont Pattern Expressions]
```

creates a light-weight object whose behavior is same as the one of an object created by the expression:

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

4.5 Group Tower Construction

The lazy creation technique, that is used for construction of ABCL/R, can not be used for ABCL/R2 directly because ABCL/R2's architecture has two kinds of reflective towers. In our implementation, this technique is extended to these two kinds of towers. First, this section describes the method to build group tower.

1. Any object has its group manager of the group that the object belongs to except for the case 2.
2. Initially, a default group manager does not have its group manager. When the empty group manager slot of a default group manager is accessed, a new default group manager is created and set to the slot.
3. A default group manager does not have its evaluators and metaobject generator initially, and these objects are not created until they are requested (Figure 4.5). Precisely, when the evaluator/group manager is requested (a), a group manager G creates its own group manager $\Uparrow G$ (b), then G creates an evaluator/group manager whose group manager is $\Uparrow G$ (c).

Any resources for new object's creation (i.e., metaobject generator, and evaluator) are acquired through group managers or given at the creation. Thus, any object created at any level satisfies the condition 1.

4.6 Individual Tower Construction

This section describes the method to build an individual tower. A prototype of ABCL/R is implemented with lazy creation technique, but the technique does not concern about efficiency, and creates too many metaobjects. In our implementa-

tion, the technique is extended to suit the hybrid group tower architecture, and to be efficient.

First, inefficiency in prototype implementation of ABCL/R is discussed. In this implementation, metaobjects are created in the lazy way. A metaobject $\uparrow x$ is actually created when the access to $\uparrow x$ takes place — when the evaluator first evaluates an expression `[meta x]` [12]. Consider that an object S is sending a message M to an object T, and S has its metaobject and T does not. (Figure 4.6–a.) Since S is executed by its metaobject $\uparrow S$, $\uparrow S$ try to send a message `[:message` M R S`]` (R is a reply destination of the message) to a metaobject for T. The access to metaobject for T causes a creation of $\uparrow T$, then T is reified into an object that is indirectly executed by $\uparrow T$. (b) After this, T is executed much slowly. However, the operation needed is to send M to T, and reifying T is unnecessary if $\uparrow S$ can send M directly to T.

Our implementation uses more elaborate method that reduces unnecessary reification:

- Initially, an object does not have its metaobject. (Its metaobjects are existing conceptually.)
- When an expression `[meta` x`]` is evaluated, where x does not have the metaobject, a light-weight metaobject $\uparrow x_L$ is created as x's metaobject. Since $\uparrow x_L$ is constructed with the light-weight object, x remains directly executable. (c)
- The $\uparrow x_L$ have ability only to receive messages for x (messages matching with the pattern `[:message` M R S`]`), and forward it to x.
- When a message requesting some reflective operations (a message not matching to the pattern `[:message` M R S`]`) arrived to $\uparrow x_L$, x is reified (i.e., x

becomes indirectly executed object) and $\uparrow x_L$ becomes a default metaobject $\uparrow x$, which is directly executed. After this, the reflective operation is executed by $\uparrow x$.

- The access to the metaobject of $\uparrow x_L$ creates a light-weight metaobject $\uparrow\uparrow x_L$, and x still remains to be directly executed.

4.7 Non-Reifying Objects

As is mentioned in section 3.3, non-reifying objects are not reflective. Thus, these objects can be compiled into Lisp code, as is done in ABCL/1[15]. In our implementation, scripts of non-reifying objects are translated into Lisp code — an expression that computes a sequential operation is directly converted to a Lisp's expression, and an expression such as message sending or object creation is converted into a library function calling. Since our system assumes multi-thread environment, non-reifying objects do not switch their context for the reply to the now type message passing.

Though a metaobject of a non-reifying object is conceptual existence, the message transmission to the non-reifying object through the meta-level should be allowed. For this reason, our implementation allow non-reifying objects to have light-weight metaobjects.

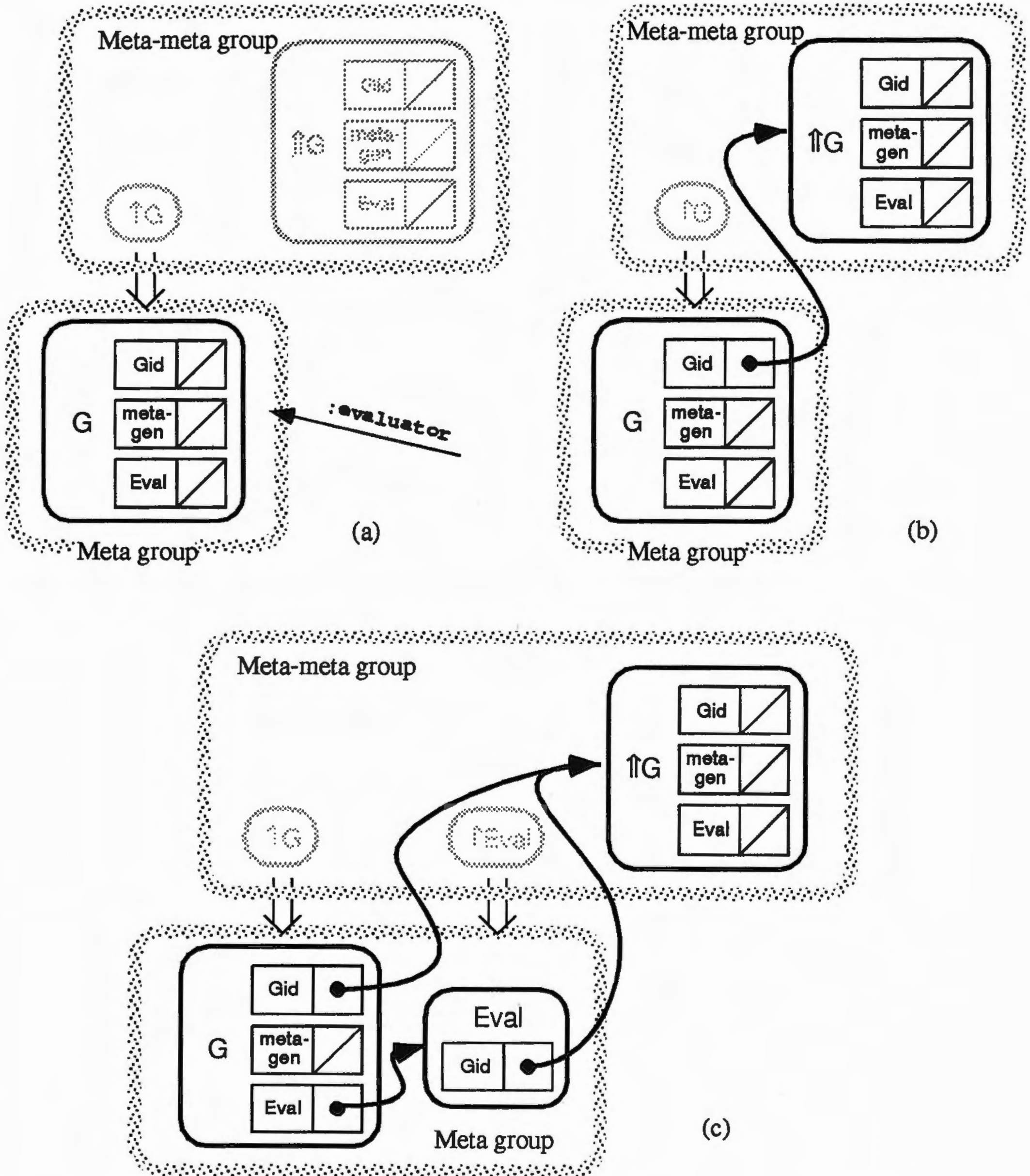

Figure 4.5: Creation of the Objects at Meta Group

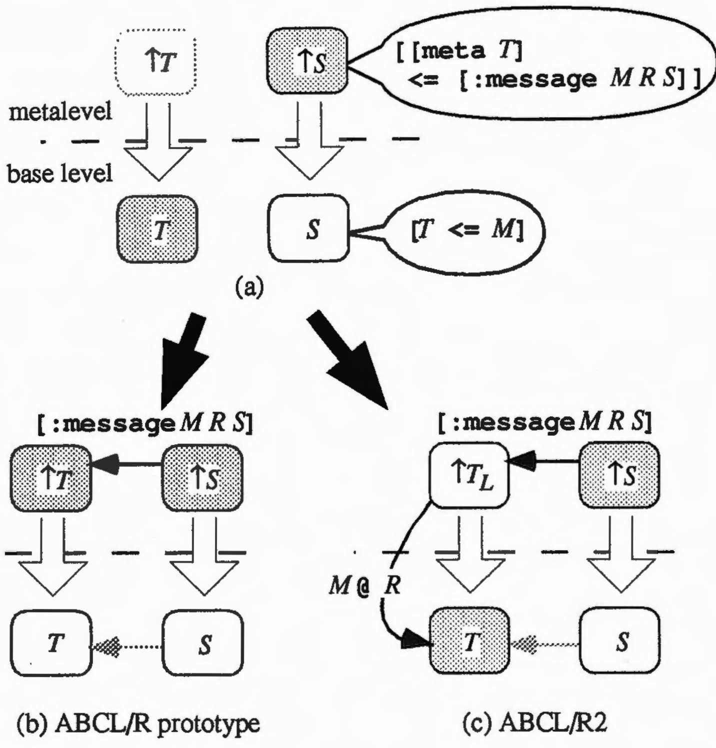

Figure 4.6: Sending a Message to a Conceptual Metaobject

Chapter 5

Performance Measurements

In this chapter, the efficiency of our ABCL/R2 implementation is measured with respect to the following points: (1) the non-reflective features, (2) the light-weight objects, and (3) the cost of reflective computation.

5.1 Non-Reifying Objects

The performance of the non-reifying objects is measured. As a non-reifying object is executed without interpretation, it is thought to have a comparable efficiency to the one for an object in a non-reflective language. For this reason, the efficiency of the non-reifying objects in ABCL/R2 and that of the ordinary objects in ABCL/1, an implementation of the object-oriented concurrent language[15], are compared[1].

The first test is a simple message transmission. Two objects, whose definition is given in figure 5.1, are created and exchange messages given n times between them. The table 5.1 shows the result.

The second test is the parallel computation of Fibonacci numbers. For each computation of $fib(n)$, two sub-objects that compute $fib(n-1)$ and $fib(n-2)$

[1]In this chapter, both the ABCL/R2 system and the ABCL/1 system have been executed in KCl (Kyoto Common Lisp) on SparcStation1+ in pseudo parallel manner.

are created. Table 5.2 shows the times elapsed for the computation.

```
[object ping
  (meta-gen non-reifying-meta)    ; ABCL/R2 only
  (state [pong := nil])
  (script
   (=> [:set p] [pong := p])
   (=> 0 !:finished)
   (=> n @ R
       [pong <= (1- n) @ R]))]
```

Figure 5.1: Object for Testing Message Transmission

n	10,000	50,000	100,000
ABCL/1	2.5	13	26
ABCL/R2	4.5	24	49

(unit: second)

Table 5.1: Execution Time for Message Transmissions

n	18	19	20	21	22
ABCL/1	36	78	198	500	1247
ABCL/R2	73	118	196	319	519

(unit: second)

Table 5.2: Execution Time for the *fib*(n)

From the results of these tests, we can see that non-reifying objects in ABCL/R2 have comparable efficiency to the ABCL/1's.

```
[object fib-gen
  (script
    (=> :new
      ![object fib
         (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]
              [[fib-gen <== :new] <= (- n 1) @ Me]
              [[fib-gen <== :new] <= (- n 2) @ Me]))]))]
```

Figure 5.2: Object for Fibonacci Numbers (parallel version)

5.2 Light-Weight Objects

The effectiveness of light-weight objects is measured. As is mentioned in section 4.4, the user program can create a light-weight object by `[cont ...]` form. We show the comparison by the sequential computation of Fibonacci numbers.

```
[object fib
  (meta-gen non-reifying-meta)          ; ABCL/R2 only
  (script
   (=> 0 !0)
   (=> 1 !1)
   (=> n @ R
       [fib <= (- n 1)
            @ [cont fib-n-1
                    [fib <= (- n 2)
                         @ [cont fib-n-2
                                 [R <= (+ fib-n-1 fib-n-2)]]]]]))]
```

Figure 5.3: Object for Fibonacci Numbers (sequential version)

Figure 5.3 shows the definition of the object **fib**, which is used for our mea-

surement. For each computation of $fib(n)$, the object `fib` creates two temporary objects to receive the values $fib(n-1)$ and $fib(n-2)$. These temporary objects are implemented by the light-weight objects in ABCL/R2, while they are implemented by ordinary objects in ABCL/1[2].

n	18	19	20	21	22
ABCL/1	7.0	11	19	30	48
ABCL/R2	4.6	7.4	12	20	32

(unit: second)

Table 5.3: Execution Time for Fibonacci Numbers

Table 5.3 shows the result of the measurement. Although the slowness of the ordinary operations shown in the previous section, the execution in ABCL/R2 is approximately 30% faster. This is important because the light-weight objects are frequently created by the evaluator during the script execution.

5.3 Costs for Reflective Computation

Though a standard object (not non-reifying object) provides reflective capabilities at the cost of execution efficiency, it would be more favorable that the execution is more efficient. This section compares the efficiency of the standard objects with that of the non-reifying objects. Table 5.4 shows comparison by the results of the the tests appeared previous sections. From the results, the execution of a standard object is 5–20 times slower than that of non-reifying objects.

[2] In ABCL/1, instead of the form `[cont` *Pattern Body*`]`, the form `[object (script (=>` *Pattern Body*`))]` is used.

	Parallel Fibonacci			Message Transmission		
	12	13	14	10,000	50,000	100,000
Standard object	22	45	67	100	494	988
Non-reifying object	4	6	10	4	24	49

(unit: second)

Table 5.4: Comparison of 'Standard' Objects with Non-Reifying Objects

Chapter 6

Example of Reflective Programming

This chapter shows a programming example suppressing the concurrency index. The concurrency index is the number of objects existing in the system at a given time during the execution of a program. In particular, the concurrency index is limited by the total number of resources in a system at given time[1]. Therefore, a scheduling method, that can control the concurrency index according to the number of resources available at the given system, is more favorable for a same computation.

The system is organized as in figure 6.1. Each application object is created with a parameter showing "degree of progress" in its computation. A computation of an application object at the base-level is invoked by a message from the metaobject to the evaluator at meta-level. In addition to the standard message, the metaobject sends the "degree of progress parameter" to the evaluator, and the metaobject of the evaluator schedules by the parameter. Giving a higher priority to an object that is expected to terminate earlier, the concurrency index is suppressed. The definitions for these metaobjects and the application objects will appear in appendix B.

For the application objects, the computation for the Fibonacci numbers and

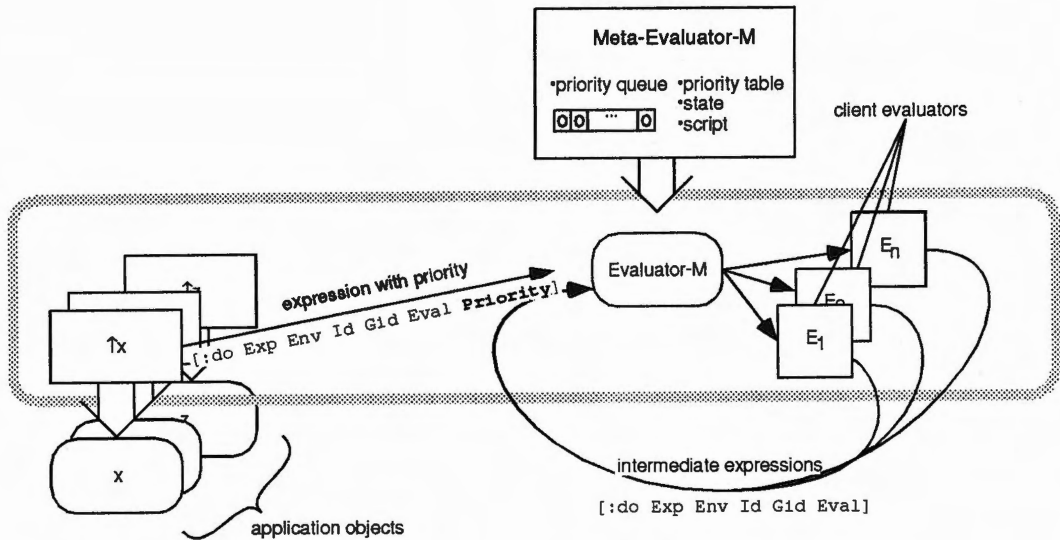

Figure 6.1: System Suppressing Concurrency Index

the quick sorting is used. Figures 6.2 and 6.3 shows the concurrency graphs for the computation of $fib(10)$ and sorting an array of 1,000 elements. Solid lines mean the concurrency indices with the priority-based scheduling, and dashed lines mean the ones scheduled without priorities (FIFO scheduling). The horizontal axis means the number of expressions processed by the evaluator. Both examples are executed with four client evaluators.

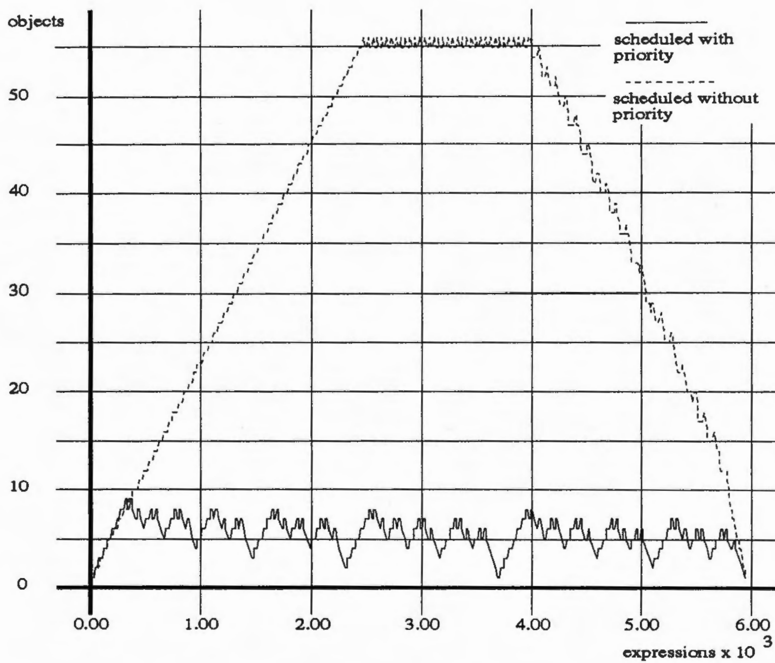

Figure 6.2: Concurrency Index for *fib*(10)

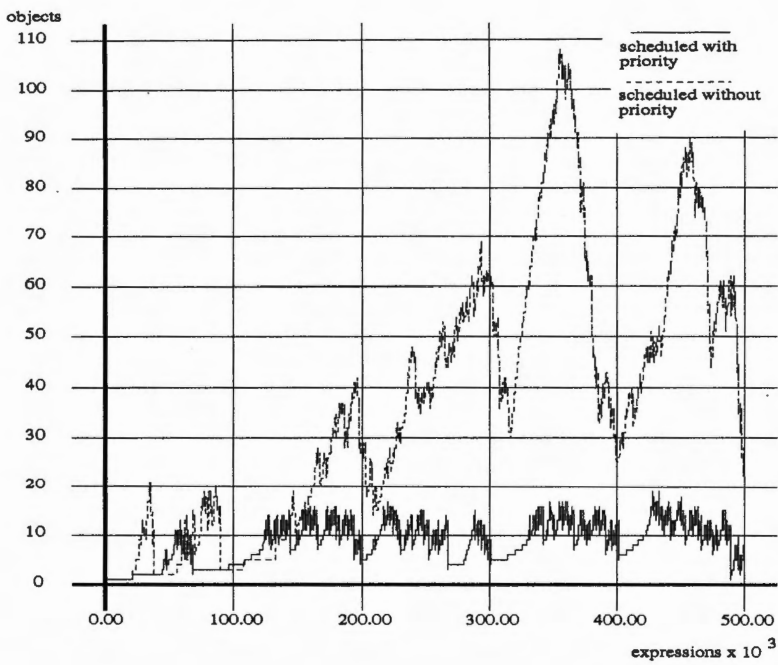

Figure 6.3: Concurrency Index for Quick Sorting

Chapter 7

Conclusion and Future Work

The language ABCL/R2 is implemented with the lazy creation scheme of metagroups and metaobjects. The system reduces 'unnecessary reification' using the light-weight metaobjects. The light-weight object also provides good efficiency for the computation that frequently uses continuation objects. Efficient script execution is achieved by the partial compilation mechanism: only selected operations are interpreted, and other ones are compiled.

Many attractive examples, such as the dynamic optimization and debugging, have not been programmed. We would do these in future. In addition, more examples should be programmed. This is because many programming experiences would give us the direction for more efficient implementation, and more sophisticated reflective architectures. One approach to more efficient implementation would be: reducing interpretations reflecting the changes at the meta-level on the compiled code at the base-level.

References

[1] Agha, G., "Concurrent Object-Oriented Programming," *Comm. ACM*, vol. 33, no. 9, pp. 125–141, 1990.

[2] Danvy, O. and K. Malmkjær, "Intensions and Extensions in a Reflective Tower," in *Porc. Conference on Lisp and Functional Programming (LFP)*, (Snowbird, Utah), ACM, July 1988.

[3] des Rivières, J. and B. C. Smith, "The Implementation of Procedurally Reflective Languages," in *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, 1984.

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

[5] Maes, P., "Issues in Computational Reflection," in *Meta-Level Architecture and Reflection*, pp. 21–35, Elsevier Science, 1988.

[6] Matsuoka, S., T. Watanabe, Y. Ichisugi, and A. Yonezawa, "Object-Oriented Concurrent Reflective Architectures," in *Workshop on Object-Based Concurrent Programming*, (Geneve, Swizerland), July 1991.

[7] Matsuoka, S., T. Watanabe, and A. Yonezawa, "Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming," in *Proc. European Conference on Object-Oriented Programming (ECOOP)*, 1991.

[8] Rao, R., "Implementational Reflection in Silica," in *Proc. European Conference on Object-Oriented Programming (ECOOP)*, pp. 251–266, 1991.

[9] Rivières, J. D., "Control-Related Meta-Level Facilities in LISP," in *Meta-Level Architectures and Reflection*, pp. 101–110, Elsevier Science, 1988.

[10] Smith, B. C., "Reflection and Semantics in Lisp," in *Conference Record of 11th ACM Symposium on Priciples of Programming Languages (POPL)*, pp. 23–35, 1984.

[11] Wand, M. and D. P. Friedman, "The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflecitve Tower," in *Meta-Level Architecture and Reflection*, pp. 111–134, Elsevier Science, 1988.

[12] Watanabe, T. and A. Yonezawa, "Reflection in an Object-Oriented Concurrent Language," in *Proc. ACM Conference on Object-Orienetd Programming Systems, Languages, and Applications (OOPSLA)*, (San Diego CA.), pp. 306–315, ACM, September 1988. (Revised version is in [15]).

[13] Watanabe, T. and A. Yonezawa, "An Actor-Based Metalevel Architecture for Group-Wide Reflection," in *Proc. REX School/Workshop on Foundation of Object-Oriented Languages (REX/FOOL)*, 1990.

[14] Yokote, Y. and M. Tokoro, "Muse: An Operating System for Next Grenerации Computing Environment," in *Computer System Symposium*, 1991. (in Japanese).

[15] Yonezawa, A., ed., *ABCL: An Object-Oriented Concurrent System.* Computer Systems Series, Cambridge, Massachusetts: MIT Press, 1990.

[16] Yonezawa, A. and T. Watanabe, "An Introduction to Object-Based Reflective Concurrent Computations," in *Proc. 1988 ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pp. 50–54, 1989.

Appendix A

Meta-Circular Definition of ABCL/R2

Following codes are the meta-circular definitions for the primary group kernel objects.

```
;;; Group Manager
[object Group
   (state [meta-gen := meta-gen]
          [evaluator := eval])
   (script
    (=> :meta-gen !meta-gen)
    (=> :evaluator !evaluator)
    (=> [:new StateSpec Env Script] @ C
        [meta-gen <= [:new StateSpec Env Script evaluator Me] @ C])
    (=> [:new-group StateSpec Env Script] @ C
        [Group  ; ' Group' is bound to the group manager of this object.
         <= [:new StateSpec Env Script]
         @ [cont meta-GMgr
               [[den meta-GMgr] <= :initialize]
               [C <= [den meta-GMgr]]]])
    (=> :initialize
       ;; do nothing
      ))]


;;; Metaobject Generator
[object meta-gen
 (script
  (=> [:new StateSpec LexEnv Scripts Eval GMgr]
      [object Metaobject
       (state [queue := [queue-gen <== :new]]
              [state := nil]
              [scriptSet := scripts]
              [evaluator := Eval]
```

```
        [Group-manager := GMgr]
        [mode := ':dormant])
  (script
   (=> [:message Message Reply Sender]
       [queue <= [:enq [Message Reply Sender]]]
       (when (eq mode ':dormant)
         [mode := ':active]
         [Me <= :begin]))
   (=> :begin
       (match [queue <== :deq]
         (is [Message Reply Sender]
             (match (find-script Message Reply scriptSet)
               (is [Bindings ScriptBody]
                   [evaluator
                     <= [:do-progn
                         ScriptBody
                         [env-gen <== [:new Bindings state]]
                         Me Group-manager evaluator]
                     @ [cont _
                        [Me <= :end]]])
               (otherwise
                (warn "~S cannot handle the message ~S"
                      [den Me] Message))))))
   (=> :end
       (if (not [queue <== :empty?])
         [Me <= :begin]
         [mode := ':dormant]))
   ;; reflective operations
   (=> :state !state)
   (=> [:set-state newState]
       [state := newState])
   (=> :queue @ R
       [queue <= :listify @ R])
   (=> :dequeue @ R
       [queue <= :deq @ R])
   (=> [:find-script Message]
       !(search-script Message nil scriptSet))
   (=> [:add-script new-script]
       [scriptSet := (cons new-script ScriptSet)])
   (=> [:remove-script Message]
       [scriptSet := (remove-script Message nil ScriptSet)])
   (=> [:enqueue item]
       [queue <= [:enq item]]))]
  ;; initialization for the state variables
  [Eval <= [:do-evlis (stateSpec-initforms StateSpec)
                      LexEnv Metaobject GMgr Eval]
        @ [cont initValues
           [Metaobject
             <= [:set-state
                 [env-gen
                   <== [:new (pairlis (stateSpec-varNames stateSpec)
```

```
                                                 initValues)
                                       LexEnv]]]]
                 !Metaobject]]))]


;;; Evaluator
[object eval
  (script
   (=> [:do Exp Env Id Gid Eval] @ C
       (match Exp
         (is [:compiled scriptBody]
             (funcall scriptBody C Env Id Gid Eval))
         (is [:variable Var]
             (match Var
               (is 'Me ![den Id])    ; pseudo variable
               (is 'Group !Gid)      ; pseudo variable
               (otherwise [Env <= [:value-of Var] @ C])))
         (is [:self-evaluate form]
             [C <= (eval form)])
         (is [:send-past Target Message Reply]
             [C <= nil]
             (if (not (null Target))
               [[meta Target] <= [:message Message Reply [den Id]]]))
         (is [:send-now Target Message]
             (if (not (null Target))
               [[meta Target] <= [:message Message C [den Id]]]))
         (is [:object-def Name Meta-gen GMgr State Script]
             (let ((Group-for-obj (or GMgr Gid))
                   (set-name
                    [cont object
                          (if Name
                              [Env <= [:set-local Name [den object]]
                                   @ C]
                              [C <= [den object]])]))
               (if Meta-gen
                   [Group-for-obj
                    <= :evaluator
                    @ [cont evaluator-for-obj
                            [Meta-gen <= [:new State Env Script
                                                 evaluator-for-obj
                                                 Group-for-obj]
                                      @ set-name]]]
                   [Group-for-obj
                    <= [:new State Env Script] @ set-name])))
         (is [:group-def Name StateSpec Script]
             [Gid <= [:new-group StateSpec Env Script]
                  @ (if Name
                        [cont GMgr
                          [Env <= [:set-local Name GMgr] @ C]]
                        C)])
         (otherwise
```

```
       (warn "Illegal expression ~S" Exp))))
;; sequential execution
(=> [:do-progn [] Env Id Gid Eval] @ C
    [C <= nil])
(=> [:do-progn [LastExp] Env Id Gid Eval] @ C
    [Eval <= [:do LastExp Env Id Gid Eval] @ C])
(=> [:do-progn [FirstExp . RestExp] Env Id Gid Eval] @ C
    [Eval <= [:do FirstExp Env Id Gid Eval]
          @ [cont _
              [Eval <= [:do-progn RestExp Env Id Gid Eval] @ C]]])
;; list evaluation
(=> [:do-evlis [] Env Id Gid Eval] @ C
    [C <= nil])
(=> [:do-evlis [FirstExp . RestExp] Env Id Gid Eval] @ C
    [Eval <= [:do FirstExp Env Id Gid Eval]
          @ [cont FirstVal
              [Eval <= [:do-evlis RestExp Env Id Gid Eval]
                    @ [cont RestVals
                        [C <= [FirstVal . RestVals]]]]]]))]
```

Appendix B

Code for Suppressing Concurrency Index

Codes for suppressing concurrency index are shown. Expressions surrounded by the boxes are inserted ones for this example.

A metaobject for an application object is created with the priority parameter. When the application object starts the execution of its script, the metaobject sends compiled expressions with its priority to the evaluator.

```
;;; Metaobject generator for application objects
[object priori-meta-gen
  (script
   (=> [:new StateSpec LexEnv Scripts Eval GMgr Priori] @ R
       ;; A metaobject for an application object
       [object Metaobject
         (meta-gen non-reifying-meta)
         (state [queue := [queue-gen <== :new]]
                [state := nil]
                [scriptSet := scripts]
                [evaluator := Eval]
                [Group-manager := GMgr]
                [mode := ':dormant]
                [priority := Priori])
         (script
          :
          (=> :begin
              (match [queue <== :deq]
                (is [Message Reply Sender]
```

```
                    (match (find-script Message Reply scriptSet)
                      (is [Bindings ScriptBody]
                          [evaluator
                           <= [:do-progn ScriptBody
                                 [env-gen <== [:new Bindings state]]
                                 Me Group-manager evaluator
                                 priority]   ; with priority
                           @ [cont _
                                  [Me <= :end]]])
                      (otherwise
                       (warn "~S cannot handle the message ~S"
                             [den Me] Message)))))
          :
          )]
        :
)]
```

Object **Evaluator-M** is an entry to multiple evaluators; its function is to distribute given expressions to its client evaluators. Exceptionally, **Evaluator-M** handles the expressions for the object creation with priority parameter. This is because that each client evaluator is a primary evaluator, and does not know to handle such expressions.

```
;;; An evaluator with multiple clients
[object Evaluator-M
  (meta-gen Evaluator-M-meta-gen)
  (state [eval-list := el]       ; el: list of client evaluators
         [current-eval := el])  ; next client pointer

  (script
   ;; object creation with priority
   (=> [:do [:object-def Name Meta-gen GMgr State Script :priority priority]
            Env Id Gid Eval] @ C
            [(or GMgr Gid)
             <= [:new State Env Script :priority priority]
             @ [cont object
                     (if Name
                         [Env <= [:set-local Name [den object]] @ C]
                         [C <= [den object]])]])
   ;; normal expression
   (=> any @ C
       [(first current-eval) <= any @ C]        ; delegate to a client evaluator
       [current-eval := (rest current-eval)]   ; move the next client pointer forward
       (if (null current-eval)
           [current-eval := eval-list]))))]
```

The scheduling is held in the object **Meta-Evaluator-M** — the metaobject of the object **Evaluator-M** — whose definition is the followings. The object **Meta-Evaluator-M** picks up the priority parameter from the message for its denotation as the scheduling key.

```
;;; A metaobject for the evaluator
[object Meta-Evaluator-M
  (meta-gen non-reifying-meta)
  (state [pqueue :=  ⟨create priority queue object⟩]
         [priority-alist := nil]
         [state := ⟨initialize denotation's state variables⟩]
         [scriptSet := scripts]
         [evaluator := Eval]
         [Group-manager := GMgr]
         [mode := ':dormant])
  (script
   ;; An expression with priority
   (=> [:message [Tag Exp Env Id Gid Eval Priority]
                 Reply Sender]
       [pqueue <= [:enq [Priority
                         [[Tag Exp Env Id Gid Eval]
                          Reply Sender]]]]
        ;; record sender's priority
       (pushnew (cons Id Priority) priority-alist :test #'equal)
       (when (eq mode ':dormant)
         [mode := ':active]
         [Me <= :begin]))
   ;; expression without priority
   (=> [:message [Tag Exp Env Id Gid Eval] Reply Sender]
       (temporary
        ;; search priority value from table
        [Priority := (cdr (assoc Id priority-alist))] )
       (if (null Priority) [Priority := 0])
       [pqueue <= [:enq [Priority
                         [[Tag Exp Env Id Gid Eval]
                          Reply Sender]]]]
       (when (eq mode ':dormant)
         [mode := ':active]
         [Me <= :begin]))
   (=> :begin
       ⟨almost same as primary metaobject⟩
       )
   (=> :end
       ⟨same as primary metaobject⟩
       ))]
```

The group pcontrol is for the application objects; objects created in this group has a metaobject with priority value, and their scripts are executed by the Evaluator-M at the meta-level.

```
[group pcontrol
  (meta-gen priori-meta-gen)
  (evaluator ⟨create Evaluator-M⟩)]
```

Following two forms are the definitions for application objects. The former one is for the Fibonacci numbers, and the latter one is for the quick sorting. Both codes are executed under the same meta-level objects shown previously.

The object creation form with the designation (priority n) means the creation of an object with its priority value n. Actually, the evaluation of this form produces an expression [:object *Name Meta-Gen GMgr State Script* :priority n].

```
;;; Parallel computation for the Fibonacci numbers
[object fib-gen
  (group pcontrol)  ; object creation in group pcontrol
  (priority 0)
  (script
   (=> [:new progress]
       ![object fib
          ;; declare the given number as a priority
          (priority progress)
          (state [sub-node-value := nil]
                 [reply := nil])

          (script
           (=> [:ans n]
               (if sub-node-value
                   [reply <= [:ans (+ sub-node-value n)]]
                   [sub-node-value := n]))
           (=> 1 ![:ans 1])
           (=> 2 ![:ans 1])
           (=> n @ R
               [reply := R]
               [[fib-gen <== [:new (- n 1)]] <= (- n 1) @ Me]
               [[fib-gen <== [:new (- n 2)]] <= (- n 2) @ Me]))]))]

;;; Parallel version of quick sorting
```

```
[object qsort-gen
  (group pcontrol)
  (priority 0)
  (script
   (=> [:new array left right]
       ;; the quick sort object
       ![object qsort
          ;; declare the length of the sub-array as a priority
          (priority (- right left))
          (state [pivot := nil] [i := nil] [j := nil]
                 [reply := nil] [wait := t])
          (script
           (=> :start @ R
               (if (< left right)
                   (progn
                     ;; parameters initialization
                     [pivot := (aref array (floor (+ left right) 2))]
                     [i := left] [j := right] [reply := R]
                     [Me <= :scan-left])
                   ;; no need to divide, immediate termination
                   (progn !:fin
                          [p-monitor <= :object-dead])))
           ;; scan left pointer to exchange
           (=> :scan-left
               (if (< (aref array i) pivot)
                   (progn [i := (1+ i)] [Me <= :scan-left])
                   [Me <= :scan-right]))
           ;; scan right pointer to exchange
           (=> :scan-right
               (if (> (aref array j) pivot)
                   (progn [j := (1- j)] [Me <= :scan-right])
                   [Me <= :test-swap]))
           (=> :test-swap
               (when (<= i j)
                  ;; swap elements
                 (let ((tmp (aref array i)))
                   (setf (aref array i) (aref array j))
                   (setf (aref array j) tmp))
                 [i := (1+ i)] [j := (1- j)])
               (if (<= i j)
                   ;; continue scanning
                   [Me <= :scan-left]
                   ;; the division is finished
                   (progn
                     ;; create two subobjects
                     [[qsort-gen <== [:new array left j]] <= :start @ Me]
                     [[qsort-gen <== [:new array i right]] <= :start @ Me])))
           ;; wait for the termination of the subobjects
           (=> :fin
               (if wait
```

```
;; wait another subobject
[wait := nil]
;; notification of the termination
[reply <= :fin])))]))]
```