学 位 論 文

A Reflective Object-Oriented Concurrent Language for Distributed Environments

分散環境におけるリフレクティブな並列オブジェクト指向言語

平成５年３月博士（理学）申請

東京大学大学院理学系研究科

情報科学専攻

一杉裕志

学 位 論 文

A Reflective Object-Oriented Concurrent Language for Distributed Environments

分散環境におけるリフレクティブな並列オブジェクト指向言語

平成５年３月博士（理学）申請

東京大学大学院理学系研究科

情報科学専攻

一杉裕志

A Reflective Object-Oriented Concurrent Language for Distributed Environments

Doctor Thesis by
Yuuji Ichisugi

Submitted to Department of Information Science
Faculty of Science
The University of Tokyo
on Mar 25, 1993
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Science

Abstract

In a computational system, reflection is the process of reasoning about and acting upon the computational system itself. Reflection is a scheme that realizes highly flexible and malleable systems. A reflective system can manipulate data called *Causally-Connected Self Representation*(CCSR) that represents the current state of its own computation. In a reflective system, the user program can manipulate its CCSR in order to change the future course of its computation. The user can define new language features or change the representation of data structures of the language within the same language framework. This thesis proposes a reflective object-oriented concurrent language RbCl which has no run-time kernel. All the behavior of RbCl except for what is restricted by the operating system and hardware can be modified/extended by the user. RbCl runs efficiently in a distributed environment and is intended for practical use. The execution of an RbCl program is performed by a *metasystem* that consists of *metalevel objects*. All the features of RbCl including concurrent execution, inter-node communication, and even reflective facilities themselves are realized by the metalevel objects, which are modifiable and extensible. Important metalevel objects are called *system objects*, that are registered in *system object tables*. The user can change the behavior of the metasystem by replacing elements of system object tables with user-defined objects. RbCl also provides a novel feature called *linguistic symbiosis* for metalevel objects. All the metalevel objects in the initial RbCl metasystem are actually C++ objects, but the linguistic symbiosis enables the user to manipulate metalevel C++ objects just as ordinary RbCl objects. Even reflective schemes and facilities themselves are realized by system objects that can be modified/extended by the user. Therefore, debugging of reflective programs and experiments on reflective schemes and facilities can be expressed and performed within the RbCl language framework.

Acknowledgments

I am grateful to Professor Akinori Yonezawa for serving as my thesis supervisor. I would like thank him for so many hours he spared for having technical discussions and teaching me how to write papers.

I would also like to thank Satoshi Matsuoka, Takuo Watanabe, Hidehiko Masuhara and Masahiro Yasugi for their numerous helpful advices and discussions about computational reflection.

I wish also thank for Professor Kei Hiraki, Yutaka Ishikawa and Hideaki Okamura for their invaluable advices and help through discussions.

Finally, I thank the members of Yonezawa Laboratory for the useful suggestions and the encouragements.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Recently, object-oriented languages are recognized its usefulness and widely spread. Also, research on distributed concurrent languages becomes very active in order to make full use of network environments, or to improve performance of applications. Usually, systems for object-oriented languages and concurrent languages contain many *run-time routines*. For example, many object-oriented languages have run-time routines such as dynamic memory management and class systems. Distributed concurrent languages should have run-time routines which support many facilities such as scheduling and inter-node communications.

The behaviors of the run-time routines affect the efficiency of program execution. For example, a scheduling based on some priority scheme may improve performance rather than the scheduling using a simple FIFO queue. It is, however, impossible to decide the best suited execution policy of run-time routines before execution. In some traditional systems, run-time routines are designed to be customizable in restricted manner (e.g., changing parameters). In distributed concurrent languages, the customization with a restricted manner is not adequate because they have to accommodate to various applications and execution environments. Consequently, flexible language systems are required whose run-time routines are easily customizable with

fewer restriction [1].

On the other hand, researchers of programming languages also require flexible language systems that can be used as a platform of research for language facilities and implementation techniques. Suppose that a researcher wants to make an experiment with a new distributed garbage collection algorithm. If there are no platforms for the experiment, the researcher must implement an almost complete distributed language system, or analize a huge amount of the source code of an existing language system to incorporate the new algorithm into it. If there is a language system whose implementation is modular, well documented and interactively modifiable, the pains for the experiment will be dramatically reduced.

Reflection is a scheme that allows us to realize such a highly flexible and malleable system. In a computational system, reflection is the process of reasoning about and acting upon the computational system itself [20, 13, 28]. A reflective system can manipulate data called *Causally-Connected Self Representation*(CCSR)[13] that represents the current state of its own computation. Reflective systems maintain causally connection between CCSR and actual status of computation. In other words, when CCSR is modified, the actual status of computation is also modified at the same time. In a reflective system, the user program can manipulate its CCSR to change the subsequent course of its computation. The user can define new language features or change the representation of data structures of the language within the same language framework.

There are many advantages of reflective systems in contrast with traditional customization schemes, such as specifying parameters or modifying the language system's source code directly.

1. The user can use the high level language for customization rather than low level languages such as C. It is especially useful for prototyping of run-time routines.

2. It is easy to construct flexible applications that adapt its behavior for dynamically changed execution environments such as the load of the system or the

[1] One of the interesting applications is to dynamically change the scheduling policy of object execution [1] in distributed simulation based on the Time Warp Scheme [10].

network traffic. Re-compilation and restart of the entire system are not necessary after changing the behavior of run-time routines.

3. The state of computation is easy to manipulate because it is abstracted as CCSR. Especially, in object-oriented systems, CCSR can be represented as objects, that can be accessed through a uniformed manner: message sending. Differential programming using inheritance or delegation also makes the system easy to modify/extend.

1.2 Previous Work

B. Smith gave the definition of a dialect of sequential Lisp, called 3-Lisp[20] which has substantial reflective capabilities. User programs of 3-Lisp are usually executed at Level 0. The behavior of the Level 0 language is defined by an interpreter at Level 1, and Level 0 and Level 1 are completely same language. This interpreter is called a *metacircular interpreter*. When a *reflective procedure* is called, it is executed at Level 1. The user can execute reflective procedure at Level 1, then it is executed at Level 2. In this way, the execution of Level 0 is implemented as an infinite tower of interpreter. This tower is called a *reflective tower*. In 3-Lisp, a triple of code, an environment and a continuation represent a computation state. Smith indicates that the user can define new language construct such as catch and throw using reflective procedures.

Following Smith's tradition, P. Maes proposed a reflective object-oriented architecture, called 3-KRS[13]. In 3-KRS, all the objects are defined by its metaobjects. The user can change the behavior of an object by modifying its metaobject. A metaobject itself is also an object, so it has another metaobject, and this chain continues to infinity. In contrast with 3-lisp, metaobjects, that are the CCSR of 3-KRS objects, contain more rich information about objects. In addition, the user can manipulate the CCSR with uniformed manner, message sending. Since 3-KRS have only one name space of global object names, it should be done carefully to change the system's global behavior for keeping the system's consistency. For example, redefinition

of metaobject named `object`, that is super class of all objects in the system, will change all behaviors of objects in the system, including the `object` itself.

CLOS is an object oriented language based on Common Lisp. CLOS-MOP(Metaobject Protocol)[11] is an object-oriented system that provides metaobject facilities to modify/extend the behavior and implementation of objects. The metaclass of a class is actually an object which creates the class. The user can modify/extend the system by customizing metaclasses and specifying metaclass at the time of creation of objects. Since CLOS-MOP have only one name space of class as well as 3-KRS, modifying the existing class dynamically is not recommended.

ABCL/R[23] is also object-oriented language and it is the first reflective concurrent language. Objects of ABCL/R have its own metaobject, as well as 3-KRS. Metaobjects are also concurrent objects. Because there are no CCSR that represent global behavior of the system, ABCL/R user can not change the scheduling policy.

ABCL/R2[15, 14] introduced group tower to manipulate computational resources such as computing power. In ABCL/R2 scheduling policy is can be customized by manipulating a metaobject that defines scheduling policy.

AL-1/D [9, 17] is an object-oriented concurrent language. AL-1/D is based on the *multi-model* reflection framework. AL-1/D provides multiple CCSRs that are suitable for various purposes in modifying the behavior of the interpreter, resource management, etc. Current AL-1/D does not provide any facilities to define new models of objects.

Apertos [25, 26] is a reflective object-oriented operating system. Apertos runs efficiently in distributed environments, and the user can modify/extend almost all parts of the system. Apertos objects at the baselevel are represented as raw data at the metalevel. In contrast with many other reflective systems, Apertos does not provide infinite reflective tower. A small kernel called *MetaCore* manages the primitives of reflection. The behavior of the MetaCore cannot be manipulated by the user so the range of customization is restricted (although Apertos allows some customization of reflective behaviors by subclassing new *reflector* classes).

Rose[19] proposes a metaobject protocol for dynamic dispatch that is efficient, powerful and language independent. However, Rose's metaobject protocol only sup-

ports a dynamic dispatch mechanism, and does not support other facilities such as concurrent execution.

1.3 Motivations

The main motivation of this research is to develop a practically useful concurrent programming system that is easy to customize and become adaptable. Previous reflective systems have the following problems:

- All previous reflective language systems are implemented on top of their *run-time kernels*, that cannot be manipulated by the user. For example, the reflective tower of 3-Lisp[20][18] is implemented by code of hundreds of lines which acts as the run-time kernel. Although the semantics of the language may be altered using the reflective capabilities, the behavior of the run-time kernel cannot be changed by the user. This is a serious problem for the language users who are keen to efficiency: for example, a user may want to tune the language system to adapt to a specific application to improve its efficiency. The existence of the run-time kernel, however, will restrict such modifications. Also, the underlying scheme of reflection, which is implemented by the run-time kernel, cannot be modified/extended by the user within the language framework.

- In all previous reflective systems, CCSRs that can be manipulated by the user does not expose the entire aspects of system implementation; rather, the CCSRs are abstracted so that the user can easily manipulate the system behavior. This restricts the aspects of computation the user can manipulate: for example, if the representation of garbage collection mechanisms is not included in the CCSR, the user cannot change the garbage collection scheme.

- In all previous reflective systems which have a *reflective tower*, the reflective tower observed by the user actually does not exist; rather, the run-time kernel makes the system act as if there is an infinite reflective tower. Because the reflective tower is far from the actual implementation, the user must understand the

behavior of the run-time kernel to predict the amount of the CPU power and memory used by reflective programs. Furthermore, complex implementation techniques are required for efficient implementation in spite of their relatively simple metacircular definitions. This may cause various implementation problems for large systems (e.g., whether or not the semantics of the reflective tower is properly preserved).

1.4 Research Contributions

In this thesis, we propose a new reflective architecture and implementation techniques that alleviate the above problems. Our language RbCl (*Reflection Based Concurrent Language*) [6, 8] is an object-oriented concurrent language with a reflective architecture, and runs efficiently in a distributed environment. Specifically, our RbCl system has the following characteristics to alleviate the problems.

- A simple mechanism called *system object tables* are introduced to remove the fixed run-time kernel at the level of the implementation language. In other words, all the run-time routines comprising the language system can be replaced by the user-defined ones. In the case of RbCl, the implementation language is C++; the user, therefore, can redefine the entire C++ program code to change the behavior of the system up to the restriction imposed by the operating system and hardware. Every possible run-time facility can be provided as libraries or applications written in RbCl.

- A novel facility called *linguistic symbiosis with implementation language* is introduced to make the CCSR completely be in accordance with the actual implementation. The linguistic symbiosis enables the user to manipulate objects of the implementation language in the same manner as ordinary RbCl objects. All language facilities initially realized by C++ objects, including concurrent execution, inter-node communication, program code management, memory management, etc., can be subject to modifications by the user. Even the reflective schemes themselves can be modified/extended by the user. Therefore, RbCl is

highly useful as a platform for experimenting new language facilities and implementation techniques. Furthermore, the linguistic symbiosis allows efficient implementation of the reflective system itself.

- In RbCl, the reflective tower can be regarded as actually existing; that is to say, the reflective tower of RbCl is *the infinite tower of the direct implementation*. However, the *metasystem*, which is a system that realizes the execution of a system on each level, is created in a lazy manner. This creation of metasystems can be achieved without using the run-time kernel, by using the characteristic of *direct implementation*. Because the behavior of the reflective tower is completely defined by the CCSR of RbCl, the user can easily predict the efficiency of the reflective programs and can also modify/extend the behavior of the entire reflective tower.

- The metasystem of RbCl (that is, the CCSR of RbCl system as mentioned above) is designed based on an *object-oriented* and *layered* architecture, so that the user can easily modify/extend its behavior in an encapsulated manner.

- In previous reflective object-oriented concurrent languages such as ABCL/R2 and AL-1/D, meta objects are also concurrent objects and they run in parallel. In RbCl, default metasystem is executed in sequential. (However, the user can make use of concurrent execution facility in metalevel programming.) This metasystem has the following merits.

  - The behavior of baselevel is completely defined by the metasystem because the execution of the metasystem is deterministic. In other words, the user can customize the detailed scheduling policy of the baselevel.
  - The mutual exclusion mechanisms of metalevel objects are not necessary.
  - The message sending between metalevel objects can be implemented by efficient function calls using stack.
  - When the user manipulates a metalevel object, it is not necessary to be careful about interference from other metalevel object that runs in parallel.

- The reflective facilities of RbCl are implemented using only simple mechanisms of the implementation language: the only special mechanism required is the coroutine mechanism. The implementation techniques employed by RbCl can be easily applied to a wide range of language systems.

1.5 Possible Applications of the Reflective Capability

There are a number of possible applications of RbCl. Some of them are listed belows:

- Extending language features such as:

  Message sending and synchronization mechanisms,
  Characteristics of objects
  (Persistent, Replicable, ...),
  Language constructs,
  Data types,
  Reflective facilities, etc.

- Dynamically changing implementation schemes such as:

  Scheduling,
  Inter-node Communication protocols,
  Distributed garbage collection,
  Load balancing,
  Fault tolerance, etc.

These are realized by manipulating the parts of the metasystem. The detailed schemes to realize some of these applications are described in Chapter 4.

1.6 Thesis Flow

The remainder of this thesis is structured as follows. In Chapter 2, we will explain language features of RbCl. In Chapter 3, the reflective architecture provided by RbCl are explained. In Chapter 4, we will describe how the RbCl facilities are implemented and can be modified by the user. In Chapter 5, we consider performance of RbCl's reflective architecture. Finally, we summarize our work in Chapter 6. In Appendix A, a number of example programs of RbCl (including ones that extend the reflective facilities themselves) are presented. To illustrate a concise overview of the RbCl system, in Appendix B, we will give a full program list of *Rscheme*, that is a kernel-less language on Scheme based on a reflective architecture like that of RbCl.

Chapter 2

Language RbCl

Since RbCl is a kernel-less system, all the features described here can be altered if the user desires to do so. We refer to the language initially provided for the user as the *plain RbCl* to distinguish it from a modified/extended RbCl. Although this thesis describes the characteristics of the plain RbCl in the strict sense of the word, we simply use the name "RbCl" except for the cases where we need to make the distinction.

2.1 Plain RbCl

The computation model on which RbCl is based is similar to that of ABCL/1[27, 3] except for reflective facilities. In this section, we describe the language features of plain RbCl except for its reflective facilities.

2.1.1 Objects

In RbCl, objects are the units of concurrency and the operations performed within an object are executed based on a sequential imperative computation model. Currently, we use the a subset of Common Lisp to describe the sequential behavior of each object. Objects are dynamically created and they interact each other only by sending messages.

The default RbCl object has only the minimum functions. For example, the facility that guarantees the uniqueness of object identity is not provided at the default system. In other words, there are no methods to determine if two object pointers denote the same objects. It is not trivial to implement this facility in distributed environments, and it usually introduces an extra overhead. Such facilities must be supplied as libraries written with reflective facilities of RbCl. Plain RbCl does not support inheritance mechanism. Classes are only generators of objects.

There are no global variables except for *system object table* described at Section 2.1.4.

Fig. 2.1 shows an object definition. `Defclass` defines a template of the objects. Expression (`new` <*class-name*>) generates an instance of the class. When an object is generated, it starts initialization immediately, then waits for messages sent from other objects. The `script` macro form specifies the method that the object can accept. The syntax of `script` is almost equivalent to ABCL/1's one. This object accepts three messages, `:empty?`, `:push` and `:pop`. The behavior of the method is described using Common Lisp form.

! *value*

means sending a return value to the caller object.

2.1.2 Message Sending

RbCl provides three types of message passing:

1. *Past type* message passing does not wait for a reply message. In RbCl, this type of message passing form is written as:

   ```
   [T <= M]    or    [T <= M @ R]
   ```

   where `T`, `M` and `R` are the target object, message and *reply destination*, respectively. The reply destination is an object to which the receiver can send a reply message. If the reply destination is not specified, sending a reply to the sender will report an error.

```
(defclass stack
  ;; Initialize a state variable.
  (let ((contents nil))
   ;; Method definitions.
   (script
    (=> [:empty?]
        ! (null contents))
    (=> [:push x]
        (setq contents (cons x contents)))
    (=> [:pop]
        ! (car contents)
        (setq contents (cdr contents))))))
```

Figure 2.1: An object definition of RbCl.

2. *Now type* message passing blocks the execution of the sender object until it receives the reply. In RbCl, this type of message passing form is written as:

   ```
   [T <== M]
   ```

   A now type message passing looks similar to an ordinary remote procedure call.

3. *Future type* message passing does not wait for a reply message as well as past type. The return value can be accessed when it becomes necessary. In RbCl, this type of message passing form is written as:

   ```
   [T <= M $ F]
   ```

   where `F` is a future object created by a `make-future` function. The return value of the future type message passing is extracted by the following function:

   ```
   (next-value F)
   ```

   The caller of this `next-value` will block if the reply is not reached at the future object yet.

RbCl lacks some linguistic facilities of ABCL/1 such as the `wait-for` statement and the *express mode* message passing.

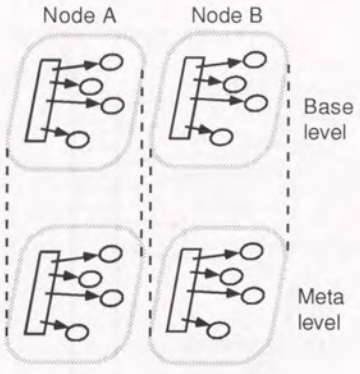


Figure 2.2: Each level of each node has its own system object table.

2.1.3 Linguistic Symbiosis

RbCl provides a novel facility called *linguistic symbiosis* with C++ objects. It hides the implementation gap between RbCl objects and C++ objects by allowing transparent inter-communication within the same memory space. An RbCl object can regard a C++ object as an RbCl object, and conversely, a C++ object can regard an RbCl object as a C++ object. The user need not be conscious of the difference of these languages. When communicating between C++ and RbCl, each uses its own communication protocols: an RbCl object communicates with a C++ object by the RbCl message passing protocol, while a C++ object communicates with an RbCl object via C++ virtual function invocation. The implementation scheme of the linguistic symbiosis is described in Section 4.5. As described in Section 4.4.1, the linguistic symbiosis plays an important role in implementing the infinite reflective tower with finite computing resources without any run-time kernel.

2.1.4 System Object Tables

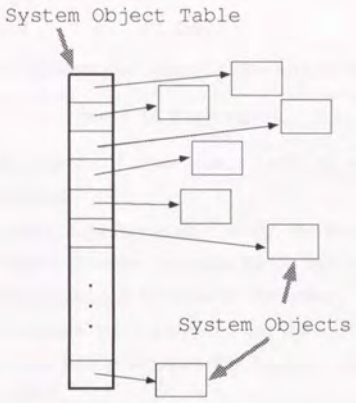


Figure 2.3: System object table.

Each level of each *node* (RbCl system consists of RbCl nodes connected by the network) has its own *system object table* (Fig. 2.2), which plays a crucial role in realizing the reflective facilities of RbCl. A system object table realizes a name space of *system objects*. System objects are important metalevel objects that determine the basic behavior of the baselevel, and are recorded in the system object table of the metalevel(Fig. 2.3). For example, the metalevel objects that determine the global behavior of the baselevel, such as schedulers, active queues and the network manager, are all system objects. The *generators* (described at Section 3.3.2) are also system objects that generate metalevel objects such as parts of baselevel objects, or primitive data element such as `cons` cells. System objects are referred by name from other objects within the same node and level. The user can change the behavior of the baselevel program on a node by replacing elements of the system object table of the metalevel on the node.

In RbCl, objects can access an element `foo` of the system object table using the

following expression:

```
(G foo)
```

The following expression replaces the element of the system object table:

```
(setf (G foo) value)
```

The *value* should be an object. If other datum (such as an integer) is specified, runtime error will be reported.

System objects are parts of metasystem. Usually, the baselevel programs do not access to the system objects provided by plain RbCl, but the metalevel programs access to them to modify/extend the behavior of the system. All the system objects provided by plain RbCl is actually C++ objects, but the linguistic symbiosis enables the user to manipulate them just as ordinary RbCl object. Fig. 2.4 shows the list of system objects in plain RbCl.

2.1.5 Inter-Node Communication

In plain RbCl, nodes can be started dynamically and can join to another RbCl system. A connection between two nodes is explicitly made by the user program. The current system identifies RbCl nodes by pair of host address and port number.

```
(setq encoder (connect (node-address "123.45.67.89" 6666)))
```

The first argument of the function `node-address` is a host address and the second argument is a port number. Return value of `node-address` can be applied to the function `connect` that makes a connection to a specified RbCl node. The return value of `connect` is called *encoder*, that is communication point to the RbCl node (The role of encoders are described at Section 4.3).

When an object sends a message, the object should have a reference to the target object. Immediately after connection is established, there are no reference between each node. System objects at the baselevel are accessible from arbitrary connecting nodes. The following function returns specified system object at the node connected through the specified encoder:

```
(remote-system-object 'name encoder)
```

There is an example program using this function in Appendix A.1.

```
Generators of first class Data:
        Linteger
        Lcons
        Lsymbol
        Lpackage

Generators of parts of RbCl objects:
        Lobject
        Llocal_pointer
        Lport
        make_local_object

Scheduling module:
        Lscheduler
        Lc_object
        Lactive_queue
        LCthread
        current_object
        current_thread
        scheduler
        active_queue
        scheduler_thread

Linguistic symbiosis module:
        Lr2c_pointer
        Lc2r_pointer
        Lr2c_startup
        Lc2r_rep_des

Inter-node communication module:
        Lnetwork_manager
        Lwakeup_baselevel
        Ltcpip_node_address
        Ltcpip_connection_request_handler
        Ltcpip_read_connection
        Ltcpip_write_connection
        Lio_buffer
        Lglobal_pointer
        Lencoder
        Ldecoder
        network_manager
        current_sender_node

Reflective tower module:
        Lglobal_var
        Llevel_manager
        Lmetalevel_level_manager
        Lsystem_object_table
        baselevel_system_object_table
        default_system_object_table
        level_manager
        metalevel_level_manager
        metasystem_generator
```

Figure 2.4: System objects in plain RbCl.

2.2 Execution environments

2.2.1 Machine Architectures

The language specification of plain RbCl assumes a distributed environments that consists of processors connected by a network. Most parts of the metasystem of RbCl are designed not to depend on a specific network architecture.

When one node or one network is down, the entire system will be halted. That is the simplest implementation of the distributed system.

2.2.2 RbCl Nodes

The RbCl system consists of *nodes* which are units of resource sharing such as CPU power and memory. In plain RbCl, each node has its own reflective tower. The metasystem is executed sequentially; concurrent execution of the baselevel objects becomes pseudo-parallel. In programming the baselevel, the user need not be aware of the distributed nature of the architecture. However, the design of the metalevel takes node boundaries into account. In the metasystem of the plain RbCl, there are no inter-node references between metalevel objects. The transparent inter-node communication between baselevel objects is realized at the metalevel using explicit remote communications.

2.2.3 Front-End Processor

The front-end processor reads source codes of RbCl and translates them into the intermediate code, that is machine independent. The intermediate code is written to an output file. Currently, the front end processor is written in Common Lisp. (Although it will be rewritten in RbCl code in the future.) The user can use the powerful macro facilities provided by Common Lisp.

2.2.4 Execution of Programs

The user of the system starts up the front end processor and enters the program. Then, start up the RbCl node from shell specifying the file name of the intermediate code generated by the front end processor. This is also called as *bootstrap code* for the RbCl node.

Plain RbCl does not support the interactive execution environments. Top-level read-evel-print-loop can be provided as one of standard library for plain RbCl. There may by various problems to design the top-level for distributed concurrent languages. For example, the semantics and the implementation of the name space of the global variables and global functions are not trivial. There may be various styles of top-level. The experiments for design and implementation of such top-levels are one of the aim of using RbCl. Plain RbCl has an ability to load other program code dynamically. The user can build up his own top-level using RbCl reflective facilities.

2.2.5 I/O

Plain RbCl provides read/write facilities from files or terminals. When an object blocks its execution because of reading, other runnable object can run. The standard library functions such as `print` outputs strings to the terminal on which the RbCl node starts. When many objects output strings simultaneously, outputs will be mixed. I/O library can be provided with more sophisticated and useful style for users. RbCl enables the user to define such libraries using the reflective facilities.

Chapter 3

Reflection in RbCl

In this chapter, we will explain the reflective architecture provided by RbCl. Example programs using the reflective facilities of RbCl are presented in Appendix A.2.

3.1 Baselevel and Metalevel

A client RbCl program resides at the *baselevel*. The *metasystem* is a *metalevel* system that realizes the execution of the baselevel RbCl program (Fig. 3.1). The metasystem consists of *metalevel objects*. Likewise the execution of the metalevel objects is realized by the meta meta system and so on ad infinitum, forming a *reflective tower*.

Although the plain RbCl metasystem consists of only C++ objects, the behavior of each object is carefully designed so as to be independent of C++ language features as much as possible. For example, global variables and global functions of C++ are not used; rather, system objects are employed for encapsulating global data and operations.

The user can replace arbitrary system objects with user-defined ones, that may be either C++ objects or RbCl objects (Fig. 3.2). The user need not be conscious of the differences between these languages thanks to the linguistic symbiosis. When the user defines RbCl system objects, the user can use all RbCl features such as concurrent execution, inter-node communication.

The RbCl metasystem is constructed in a layered manner. In other words, the

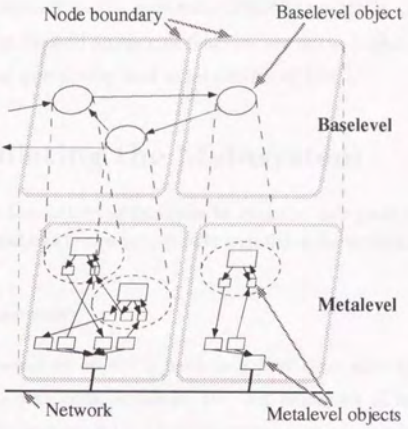


Figure 3.1: The baselevel and the metalevel in RbCl.

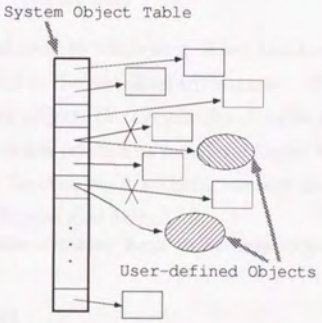


Figure 3.2: The user can replace all system objects.

metasystem consists of several layers corresponding to the degree of abstraction. Program modules that depend on the hardware architecture are in the lower layers, and program modules that depend on specific features are in the higher layers. The layered architecture enhances portability and extensibility of RbCl.

3.2 Manipulating the Metasystem

Plain RbCl provides the following facilities to execute user code at the metalevel.

The function `metalevel-new` is the only primitive for reflection in current plain RbCl.

```
(metalevel-new class-name)
```

The generated metalevel object is used as a *first class data* at the baselevel. (In plain RbCl, integers, cons cells, symbols, etc. are examples of first class data. The exact definition is in Section 4.1.) Any *operations* on the created first class data occurs message sendings to the metalevel object. How this metalevel object can be manipulated is described in Section 4.1.

A macro form `meta-lambda` is defined using `metalevel-new`.

```
(meta-lambda (... args ...) ... body ...)
```

This expression is evaluated to a function. When this function is called, the body of the function is evaluated at the metalevel environment. When an expression in the `body` accesses to a system object, the system object table at the metalevel is used. Arguments of the function are referred as metalevel object within the body.

Actually, calling this function send a `:call` message to a metalevel object that have a method `:call` with specified body.

`Metalevel-exec` is also a macro form to execute expressions at the metalevel environment.

```
(metalevel-exec ... body ...)
```

This is equivalent to

```
(funcall (meta-lambda () ... body ...))
```

The function `reify` enables the user to manipulate first class data as ordinary objects (in other words, targets of message sending).

```
(progn (setq foo (reify 1))
       (print foo)
       (print [foo <== [:fix]]))

Output:

^(Lr2c_pointer )  ; output of (print foo)
1                 ; output of (print [foo <== [:fix]])
```

The message sent to the reified first class data `foo` is actually sent to the metalevel object representing 1.

The function `reflect` is inverse function of `reify`.

3.3 Characteristics of the Metasystem

In this section, we will describe characteristics of the metalevel of plain RbCl.

3.3.1 Kernel-less System

The metasystem of RbCl is a *kernel-less system*. In this section, we describe the characteristics and benefits of kernel-less systems.

If a system written in a programming language L has a facility that enables the user to replace every part of its program code by a user-defined one, we say that the system is a *kernel-less system on language L*.

The following systems are examples of kernel-less systems: when a system's entire machine language instruction code is located in mutable store, and the system has a facility for modifying the values of contents of arbitrary addresses, the system is a kernel-less system on the machine language. The window system on a Lisp machine is a kernel-less system on Lisp because all the Lisp functions defining the behavior of the system can be redefined by the user. A more elaborate example is as follows: the behavior of a reflective language is often defined by a *reflective tower*, that is,

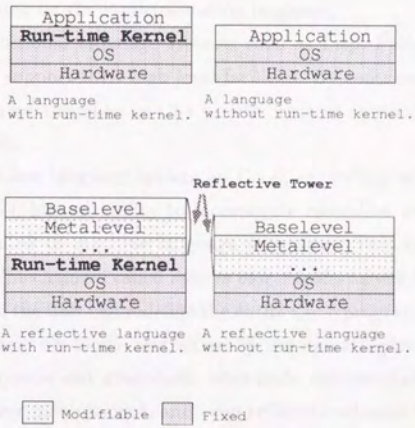


Figure 3.3: Language systems with/without run-time kernels.

an infinite tower of metacircular interpreters. More specifically (e.g., in 3-Lisp), the behavior of the language at level n is defined by the interpreter at level $n+1$. The user can replace the entire interpreter code at level n using reflective computation at level $n+1$. Therefore, any system at level n is a kernel-less system on the language defined at level $n+1$. This is one reason why the reflective system is so flexible. However, since the entire 3-Lisp system must be implemented by another language, the program code implementing the behavior of the reflective tower can never be modified by the user of the 3-Lisp system, so the 3-Lisp system is not kernel-less as it has a kernel running on the implementation language.

Most of the traditional language systems such as Lisp, Prolog, Smalltalk, have run-time kernels to support their high level facilities. By contrast, machine language programs written in an assembler or C(++), etc., require no run-time kernel support except for OS kernel.

RbCl is a kernel-less language system on C++, or, we can say "on machine language" because C++ has an ability to incorporate assembler programs, and C++ programs are compiled to machine language programs. This kernel-less system is realized by a simple mechanism called *system object tables* as will be described in Section 3.1. As a result, the user can redefine the entire C++ program code to change the behavior of the system up to the restriction by the operating system and hardware (Fig. 3.3). Thus, concurrent execution, inter-node communication, program code management, memory management, and even reflective schemes and facilities themselves can be modified/extended by the user. In language systems that have run-time kernels, the trade-offs between various characteristics such as efficiency, flexibility, programmability, safety, portability, are achieved only by the system implementer. In RbCl, the balance of the trade-offs between all such characteristics can also be changed by the language user.

A problem with kernel-less systems, such as a machine language program in mutable store (as described above), is that it is often extremely difficult and dangerous to change the behavior of the system dynamically: as a result, they cannot be used for practical purposes. In RbCl, its reflective facilities and linguistic symbiosis enable the user to easily modify/extend behavior of the system thanks to the encapsulation

provided by the object-oriented nature of RbCl.

3.3.2 Dynamic Creation of Metalevel Objects

Many types of objects are created dynamically at the metalevel. Normally, C++ uses '`new` *class*`()`' to create an object, but this is not flexible because the construction code of the specified class is directly inlined. Instead, we employ an indirect creation scheme, which enables the user to replace the construction code. We provide system objects called *generator*s corresponding to each C++ class. When a generator object receives a message `:call`, it creates an metalevel object and returns it to caller object. For example, to create a port object (a metalevel object that represents a message queue of an object), the user sends the following message to the system object named `Lport`, which is a generator of the port object:

```
[(G |Lport|) <== [: call]]
```

In plain RbCl, there are no generators that create generator objects dynamically.

Note that all metalevel objects have an opportunity to be modified/extended by the user. All metalevel objects are categorized into two types, the system objects, which exist from the initial state of system, and dynamically created objects. System objects may be replaced with user defined objects by manipulating the system object table. Replacement of the generator object changes the behavior of newly created object.

If a user decides that a dynamically created object should be replaced with another object at a certain time other than its creation time, the user could extend the generator in advance so that it returns an *invisible pointer* object that delegates all messages except the `:become` message to a newly created object Then the user can replace the object dynamically by sending the `:become` message to the invisible pointer [1].

[1]Here is another reason why the default RbCl system does not guarantee the uniqueness of object identity. If metalevel objects use such a facility to determine if two objects are same one, it is difficult to implement flexible facilities such as invisible pointers nor the linguistic symbiosis

3.3.3 Differential Programming

The facility of differential programming is necessary to enable the user to reuse the system objects provided at default system. In RbCl, all metalevel objects are defined obeying a convention that enables extending object dynamically using delegation mechanism[12].

- They must have instance variables named `self` and `super`. The initial value of `self` is the object itself, and the initial value of `super` is an object that does not do anything.
- They must have methods to refer/update the value of `self` and `super`.
- The message sending to itself must be performed as message sending to the variable `self`.

The following function extends an object.

(`extend` *mixin parent*)

mixin and *parent* are objects, not classes. They may be both an RbCl object or a C++ object as far as they obey above conventions. The return value of this function is an extended object (Fig. 3.4). The behavior of the *parent* object is not defined when it receives a message directory after this extension.

The function `extend` is defined as follows.

```
(defun extend (mixin parent)
  [mixin <== [:set-super parent]]
  (do ((super parent [super <== [:get-super]]))
      ([super <== [:null]])
    [super <== [:set-self mixin]])
  mixin)
```

This is a simple example extending cons cell generator (the system object named `Lcons`) so that it prints a message if it receives a request to generate a cons cell. The expressions in `metalevel-exec` form are evaluated in the metalevel.

within the uniformed framework. Therefore, all operations on metalevel objects must be achieved by message passing.

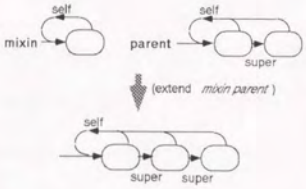


Figure 3.4: The delegation mechanism of metalevel objects.

```
(metalevel-exec
 (defclass trace-mixin
    (script
     (=> any @ reply-destination
         (print "Received !!")
         ;; Delegate any messages.
         [super <= any @ reply-destination])))
 (setf (G |Lcons|)
       (extend (new trace-mixin) (G |Lcons|))))
```

Because all metalevel objects obey the conventions that enable the differential programming, this `trace-mixin` class can be used for tracing any type of metalevel objects such as active queues, the network daemon.

Chapter 4

Metasystem

The metasystem of RbCl implements the execution of the baselevel. All the RbCl facilities are realized by metalevel objects. They are designed carefully so that the user can easily modify/extend the behavior of them. Important metalevel objects that determine the basic behavior of the baselevel are called system objects, that are recorded in the system object table of the metalevel. System objects may be replaced with user defined objects to modify/extend the behavior of the system. The metasystem is divided into some independent modules. In particular, the following mechanisms are implemented by characteristic module of RbCl.

- First class data (primitive data types).
- Scheduling mechanism that implements pseudo parallel execution within a node.
- Reflective facilities.
- Inter-node communication mechanism.
- Linguistic symbiosis.
- Garbage collection.

In this chapter, we will describe how these RbCl facilities are implemented and can be modified by the user.

4.1 First Class Data

First class data in RbCl are subject to the following treatments:

- Assigning them to variables.
- Passing them to functions as arguments of function calls.
- Sending them to other objects as arguments of message sending forms.

Currently, plain RbCl provides the following data types as first class data:

- Integers (32bit fixed)
- Cons cells
- Symbols
- Function Objects
- Object pointers

Unlike Smalltalk, first class data and objects are distinguished. *Operations* on first class data (e.g., arithmetic operations or getting car of cons cells) are not via message passings. In particular, a pointer to an object is distinguished from the object body. An object pointer is a first class data but an object body itself is not. A single object body may be referred through various pointers with which messages are transmitted in various ways (e.g., various routing algorithms). Usually, the user does not have to be conscious of this difference between pointers and object bodies.

In plain RbCl, all the first class data may be used as arguments of the `print` function. Unlike Smalltalk, printing first class data is not performed by message passing. This difference is exposed when an object pointer is printed. Printing a remote object pointer only uses the local information, and the remote access to the object body does not occur.

First class data are sometimes copied or modified by the system implicitly. For example, cons cells are copied and object pointers are modified when they are passed

to remote objects through the network. In contrast with first class data, object bodies (in plain RbCl system) never be copied or modified without explicit message passing.

At the metalevel, first class data which are simple data at the baselevel are represented as metalevel objects. All the operations on first class data are represented by message passing at the metalevel. There are no builtin operations on the metalevel objects. Even the following basic operations are represented by message passing at the metalevel.

- Identifying the type
- Equality checking
- Printing
- Encoding/decoding

For examples, print operation at the baselevel:

```
(print foo)
```

are represented by the metalevel interpreter as

```
[<foo> <== [:print]]
```

where `<foo>` is a metalevel object that represents the baselevel first class data `foo`.

Past type Message sending forms at the baselevel such as

```
[obj <= message]
```

are represented as

```
[<obj> <== [:send <message>]]
```

where `<obj>` is a metalevel object that represents the pointer `obj` and `<message>` is a metalevel object that represents the message `message` .

In this way, the implementations of first class data are completely encapsulated at the metalevel. This increases the flexibility of the metasystem dramatically. The user can define new first class data with arbitrary characteristics by defining new metalevel objects. The user can easily modify/extend the behavior of first class data using differential programming.

4.2 Scheduling

4.2.1 Scheduler and Active Queue

In RbCl, the *scheduler* and the *active queue* are realized as system objects at the metalevel so that the user can easily modify/extend the scheduling policy. Currently, a non-preemptive scheduler is provided at the metalevel. In plain RbCl, the scheduling policy satisfies the following requirements.

- All active objects should be executed within a finite time.
- All input through a network from outside should be managed within a finite time.

These are realized as follows: A system object named *active queue* is a metalevel object that queues runnable objects in the node. Each metalevel object enqueued in active queue accepts `:run` message from the *scheduler*. The metalevel object interprets the intermediate code of the corresponding object one step, then returns control to the scheduler. The default queue is a simple FIFO queue. The user can replace it with a priority queue that gives priorities to objects according to certain evaluation function.

The scheduler also performs the polling the network inputs. The scheduler sends a `:poll` message to a system object named `network-manager` at intervals.

The scheduler executes infinite loop sending messages to active objects and the network manager. The user can change the scheduler to modify the timing of polling the network.

4.2.2 Changing Scheduling Policy

The following example of reflective programming is to change the scheduling policies by replacing the active queue.

```
(metalevel-exec
 (defclass priority-queue
    (let ((queue nil))
```

```
      (script
       (=> [:empty]
           !(null queue))
       (=> [:enqueue obj]
           (setq queue (sort (cons obj queue)
                             :test #'calc-priority))
           ! nil)
       (=> [:dequeue]
           ! (pop queue)))))
(let ((new-queue (new priority-queue)))
  (do ()
      ;; Repeat until it becomes empty.
      ([(G active-queue) <== [:empty]])
    [new-queue <==
               [:enqueue
                [(G active-queue) <== [:dequeue]]]])
  (setf (G active-queue) new-queue)))
```

This priority queue has a sorted list of active objects. The priorities of objects are determined by function `calc-priority`. Replacing a global object that has internal states should be performed carefully to preserve the system consistency. In this case, all the contents of the old active queue should be enqueued in the new one. Otherwise, the execution of the baselevel will stop because initial content of the new queue is empty.

A complete example of changing the scheduling policy is presented in Appendix A.2.

4.3 Inter-node Communication

4.3.1 Three Layers

The inter-node communication mechanism is partitioned into three layers, the lowest layer depends on the hardware architecture and the highest layer depends on the characteristics of the RbCl objects. As a result the RbCl system becomes portable across various parallel machines and suitable to be a platform of implementing other

parallel languages. In the plain RbCl metasystem, there are no remote reference between metalevel objects. The inter-node communication between baselevel objects is realized at the metalevel using metalevel objects such as *network manager*. These metalevel objects communicate to other node using system calls provided by the operating system. This is the lowest layer of the inter-node communication mechanism and its implementation depends on the machine architecture. One example of modifying/extending metalevel objects in this layer is to change the buffering strategy to reduce the communication cost.

The *encoder* and the *decoder* objects are in the middle layer. When a remote message passing occurs, an encoder object on the sender node translates the structured message to a byte stream, and a decoder object on the receiver node reconstructs the message. The user can modify/extend this layer to change the protocol of encoding/decoding data.

The *pointer* and the *port* objects are in the highest layer. This layer realizes network transparent message passing using lower layers. The port object is a message queue of each baselevel object. If the pointer refers to a local object, it only contains local address of the port of that object. If the pointer refers to a remote object, it contains encoder object where the remote object resides and local address of the remote object within its node. The user can monitor message sending by extending pointer objects.

4.3.2 Network Manager

Ths system object named `network-manager` manages input/output ports from/to other RbCl nodes. The scheduler sends message `:poll` to the network manager at intervals. When the network manager receives the message `:poll` , it will check input/output ports provided by operating system. When some inputs/output are available, the network manager calls the handler object associated with the input/output port[1].

[1]In UNIX system, input/output ports are file descriptors. The network manager is implemented by `select` system call.

4.3.3 Read/Write Connections

In plain RbCl, there are three types of handlers, `Lread_connection`, `Lwrite_connection`, `Lwakeup_baselevel`. When a `Lread_connection` receives the `:handle` message from the network manager, the `Lread_connection` reads the *packet*, the message sent through the network, from the associated input port using OS primitive (e.g., `read` system call in UNIX system). When a `Lwrite_connection` receives the `:handle` message from the network manager, the `Lwrite_connection` writes a buffered packet to the associated output port.

When the meta meta system exists, the network manager at the metalevel behaves in the same way. The network manager at the metalevel is explicitly implemented by the network manager at the meta meta level. At the meta meta level, `Lwakeup_baselevel` is associated with all input/output ports used by the metalevel. When some metalevel input/output ports are available, the meta meta system wakes up the metasystem to process them.

4.3.4 Connection Request Handlers

In plain RbCl, the connections between nodes are established dynamically at execution time. From the point of view of the baselevel, the user executes the following form as described in Section 2.1.5 to make a connection.

```
(setq encoder (connect (node-address "123.45.67.89" 6789)))
```

Each RbCl metasystem has a metalevel object: the connection request handler. This object is registered in the network manager and associated with *connection request port* [2]. When an input for the connection request port is available, the network manager sends a message `:handle` to the connection request handler.

Each level has its own connection request handler. The outside system can communicate with each level on a node independently.

[2] In UNIX version of RbCl, a connection request port is a server socket.

4.3.5 Encoding/Decoding

All data should be encoded into byte streams when they are sent to remote hosts through the network. In addition, there are many types of messages that controls the meta system. The example is messages for garbage collection system (e.g., mark messages for mark&sweep collectors). These control messages are also encoded into byte streams.

In RbCl, data encoding/decoding and system control messages are processed by the same mechanism. The encoding/decoding routines are defined as system objects.

Message encoding is performed by a metalevel object called *encoder*. Each encoder is corresponding to each connection to another node. The encoder creates *packets* and sends it to the corresponding node. When a node receives a packet, the network manager on the receiver node sends a `:handle` message to the *Lread_connection* object corresponding to the sender node. System objects which have the method `:decode`is called *decoder objects*. Each packet consists of two parts, a decoder name and its argument. A decoder name is a small integer that denotes a system object name. The `Lread_connection` object sends a `:decode` message to the decoder object indicated by the name. The decoder object decodes the contents of the packet. The length of argument part is determined by each decoder object.

The argument part is passed to the specified decoder as *buffer object*. Buffer objects can receive messages `:get-int` and `:get-L`. The return values of these message sendings are an integer or a metalevel object read from the buffer, respectively.

Fig. 4.1 is an example of encoder/decoder object for cons cells [3].

4.3.6 Changing Inter-Node Communication Protocols

In RbCl, it is possible to adapt the inter-node communication protocols to specific application to reduce the overhead of the network communication.

For example, suppose all remote messages in an application are lists of two integers (e.g., `(12 34)`), length of packets can be optimized to three words, the decoder name and the two integers. This specialized encoder/decoder routine works obviously

[3]These objects do not treat recursive structure.

```
(defclass |Lcons|
 ;; cons cell
 (let (car cdr)
  (script
   ...
   (=> [:encode buf]              ; messages from an encoder
    ;; specify the decoder
    [buf <== [:put-int [(reify '|Dcons|) <== [:symbol-id]]]]
    ;; arguments for the decoder Dcons
    [buf <== [:put-L car]]
    [buf <== [:put-L cdr]]
    ! NIL)
   ...
   )))

(defclass |Dcons|
  ;; cons decoder
  (script
   (=> [:decode buf]              ; messages from an read-connection
       (let ((car [buf <== [:get-L]])
             (cdr [buf <== [:get-L]])
           ! [(G |Lcons|) <== [:call car cdr]])))))
```

Figure 4.1: The encoder and the decoder of cons cells.

more efficiently than original one. The following program is an example to do this. The function `extend-generator` extends the specified generator so that all instances created by the generator will be extended using specified mixin class. In the following example, all cons cells generated by the extended generator will have extended encoder routine defined by `special-encoder-mixin`.

```
(metalevel-exec
 (defclass special-encoder-mixin
     (script
      (=> [:encode buf]
          [buf <== [:put-int [(reify 'special-decoder)
                              <== [:symbol-id]]]]
          [buf <== [:put-int (reflect [super <== [:car]])]]
          [buf <== [:put-int (reflect [[super <== [:cdr]]
                                       <== [:car]])]]
          ! (reify nil))
      (=> any @ rep-des
          [super <= any @ rep-des])))
 (setf (G |Lcons|) (extend-generator special-encoder-mixin
                                     (G |Lcons|)))
 (defclass special-decoder
     (script
      (=> [:decode buf]
          ![(G |Lcons|)
            <== [:call
                 (reify [buf <== [:get-int]])
                 [(G |Lcons|)
                  <== [:call
                       (reify [buf <== [:get-int]])
                       (reify nil)]]]]
          )))
 (setf (G special-decoder) (new special-decoder))
)
```

4.4 Reflective Tower

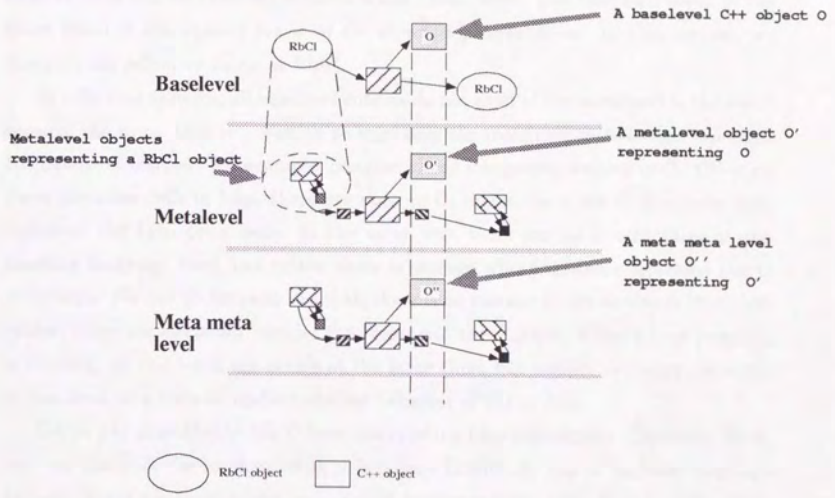


Figure 4.2: The infinite tower of the direct implementation.

4.4.1 The Infinite Tower of Direct Implementation

Without run-time kernels, the previous reflective systems would be impossible to implement with finite computing resources. A run-time kernel was necessary to make the system act as if an infinite reflective tower exists. In contrast, an RbCl system, which embodies a tower, is implemented with finite computing resources without a run-time kernel. This is achieved by employing the linguistic symbiosis with C++ objects, that can be executed without a run-time kernel. The reflective tower of the plain RbCl is *the infinite tower of the direct implementation.* In this section, we describe the reflective tower of RbCl.

In reflective systems, all baselevel entities do not exist at the metalevel in the strict sense of the word; that is to say, there exist only the metalevel entities that *represent* the baselevel entities. For example, imagine a Lisp interpreter written in C. Although there are `cons` cells in Lisp, there are none in C; rather there are C structures that represent the Lisp `cons` cells. In the same way, there are no C structures at the machine language level, but rather there is storage whose contents represent the C structures. We can go far as to say that, there is no storage at the hardware level, but rather, there are electronic entities that represent the storage. When a Lisp program is running, all the levels are active at the same time, but usually one pays attention to one level at a time to understand the behavior of the system.

Let us pay attention to the C level interpreting Lisp expressions. Generally, there are two methods for implementing a language facility on top of another language system: *direct implementation* and *explicit implementation* [18]. For example, when the '+' operation of Lisp is implemented by using only the '+' operation of the C language, we say that "the '+' operation is directly implemented." In this case, the semantics of the '+' operation of Lisp fully depends on that of C. On the other hand, if the meaning of the '+' operation is defined explicitly using the usual primitive recursion scheme, we say that "the '+' operation is explicitly implemented."

In the same way, let us pay attention to the metalevel of RbCl that implements the baselevel of RbCl. RbCl provides the linguistic symbiosis that enables C++ objects to be executed as baselevel objects. In the plain RbCl metasystem, the execution of the

baselevel C++ objects are directly implemented. This is achieved as follows(Fig. 4.2): a baselevel C++ object O is represented by a single metalevel object O' (in contrast to a baselevel RbCl object which is represented by multiple metalevel objects). O' has instance variables and methods identical to those of O. Therefore, a message sending to O at the baselevel can be simply represented by a message sending to O' at the metalevel.

Let us pay attention to the meta meta level of RbCl that implements the metalevel of RbCl. There are meta meta level C++ objects that represent metalevel C++ objects. For example, the metalevel object O' is represented as a meta meta level object O" that has instance variables and methods identical to those of O'. Because the plain RbCl metasystem consists only of C++ objects, the computation state of the meta meta level is strictly identical to that of the metalevel. In this way, the meta meta level of the plain RbCl can be regarded as actually existing and directly implementing the execution of the metalevel C++ objects. (Fig. 4.2 illustrates this whereby each C++ object is represented by a meta level C++ object that has the same hatch pattern.)

The meta meta level itself is also regarded as directly implemented by the meta meta meta level and this tower of direct implementation continues on infinitely. In this manner, the reflective tower of RbCl is realized with finite computing resources.

According to this view, arbitrary (non-reflective) systems could be regarded as being implemented by the infinite tower of direct implementation. However, such view is usually meaningless because the user of such systems cannot manipulate the lower levels. On the other hand, the reflective facilities of RbCl enable the user to manipulate the arbitrary levels if the user desires. The meta meta system that interprets RbCl objects at the metalevel is explicitly generated by a metasystem when a user creates an RbCl object at the metalevel for the first time. Therefore, the infinite tower of direct implementation of RbCl has the same power as the reflective towers of other reflective systems. How the meta meta system is generated by the metasystem is described in Section 4.4.4.

4.4.2 Level Manager

The linguistic symbiosis enables C++ objects to be executed at the baselevel. As explained in Section 3.1, each level has its own system object table that realizes a name space of system objects. Consequently, the baselevel C++ objects must be executed in the baselevel's name space. The management of the name space is conducted by system objects called *level managers*. Each level of each node has its own level manager. The level manager of the metalevel performs *level shifting*, that is, switching the current name spaces of the system objects between the baselevel and the metalevel. The level manager of the meta meta level similarly performs the level shifting between the metalevel and the meta meta level. The linguistic symbiosis and all the reflective facilities such as creating metalevel RbCl objects are implemented using the primitive level shifting capability provided by the level managers.

Note that a level manager only switches the name spaces. Wherever the name space of the metalevel is shifted, the baselevel is still executed by the metasystem and the metalevel is still executed by the meta meta system and so on.

The level shifting mechanism is implemented as follows: a C++ object refers to a C++ global variable to know the appropriate system object table that represents the current name space. A C++ object accesses a system object as follows:

```
system_object_table[symbol_id]
```

`System_object_table` is a C++ global variable which is a pointer to an array of C++ objects. This array represents the system object table. `Symbol_id` is a small integer that represents the global name of a system object. The level manager changes the value of the C++ global variable `system_object_table` to an appropriate system object table when it receives messages `:shift-to-meta` or `:shift-to-base`.

Level managers are system objects, and thus can also be replaced with user-defined objects. Therefore, programs that require changes to the reflective facilities of the RbCl itself, such as debugging of reflective programs, or performing experiments on reflective facilities, can be expressed within the RbCl language framework. In Section 4.4.5 we will explain this in detail.

4.4.3 Mechanism for Reflection

The reflective facilities of RbCl can be easily implemented because of the linguistic symbiosis. Generating a metalevel object using user specified RbCl code can be simply implemented as follows.

```
LP make_metalevel_object(LP code){
  G(S_metalevel_level_manager)->shift_to_meta();
  LP obj = make_object(code)->c2r();
  G(S_level_manager)->shift_to_base();
  return obj;
}
```

A system object named `metalevel_level_manager` generates a meta meta system if it is not exist yet. The function `make_object` generates an RbCl object using whose behavior is specified by `code`. The return value of a virtual function call `c2r()` to the generated RbCl object can be used as a C++ object.

4.4.4 Generating the Meta Meta System

All reflective systems that have reflective towers implement lazy creation of metasystems within run-time kernels. The RbCl metasystem can generate the meta meta system within the RbCl language framework — that is, it does not require a run-time kernel to do so. This is achieved as follows: first, let us define an object to be *primitive* if it is implemented only by direct implementation. For example, C++ objects are primitive objects while interpreted RbCl objects are not. As explained in Section 4.4.1, when a primitive object runs at the $(\text{meta})^n$ level, the same primitive object actually runs at the $(\text{meta})^{n+1}$ level. For example, when a metalevel primitive object A creates another metalevel primitive object B and sends a message to B, the corresponding meta meta level object A' actually creates the corresponding meta meta level object B' and sends a message to B'.

The meta meta system can be generated by the metasystem using this characteristic of primitive objects. To briefly summarize, the only thing needed to be done is to generate a new metasystem that consists of only primitive objects.

The meta meta system is generated by the metasystem in the following way:

1. Create an array of objects that represents the system object table of the meta meta system.

2. Shallow-copy all the elements from the `default-system-object-table` to the system object table of the meta meta system. `Default-system-object-table` is a system object that contains the system objects of the plain RbCl metasystem that are shared by all the levels. Most system objects such as generators have no internal states, so they can be shared.

3. Create additional objects and register them to the system object table. These objects are system objects that cannot be shared by all the levels — examples are a level manager and a scheduler.

4. Initialize the created system objects as if the execution of the metalevel were directly implemented by the meta meta system.

The generated meta meta system provides all the facilities provided by the plain RbCl metasystem. When an RbCl object is created at the meta meta level, the meta meta meta system will be generated in the same way.

4.4.5 Examples of Modifying the Reflective Scheme

Replacing level managers

The reflective scheme of RbCl itself can be modified/extended or even completely changed by the user by using the RbCl reflective capabilities. If the user wants to print messages whenever the current level shifts between the baselevel and the metalevel, the user can replace the default level manager with a user-defined level manager which prints out a message whenever the level shifts. This is achieved by the following RbCl program.

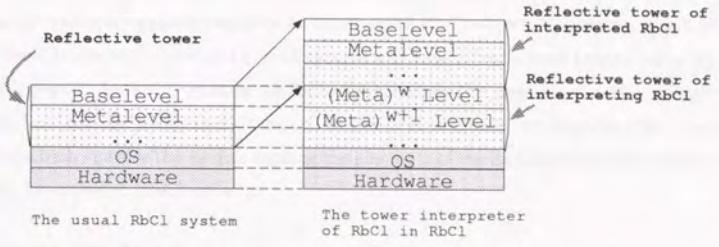


Figure 4.3: An interpreted reflective tower and $(\text{meta})^{\omega}$ level.

```
(metalevel-exec
 (setf (G level-manager)
       a-user-defined-level-manager))
```

`Metalevel-exec` is a macro form to execute expressions in the metalevel environment. `(G level-manager)` denotes a system object named `level-manager`.

The above example only changes the level manager at the metalevel. In addition to this, the user can change all the level managers of the metasystems created in a lazy manner. As described in Section 4.4.4, new metasystems are created using the elements of `default-system-object-table`. Level managers of the new metasystems are created by the system object named `level-manager-generator`. The user can replace this element of `default-system-object-table` as follows:

```
(metalevel-exec
 [[(G default-system-object-table)
    <== [:aset
         (symbol-id 'level-manager-generator)
         a-user-defined-level-manager-generator]]])
```

After executing this code, when a new metasystem (the meta meta system, the meta meta meta system, etc.) is generated, a level manager is generated using the user-defined level manager generator.

There is one caveat in this example: when the user replaces the `level-manager-generator` with an interpreted RbCl object, in practice it will never be used because the new meta meta system will have already been generated using the old value of `default-system-object-table` when the user creates the interpreted RbCl object. To avoid this situation, in the next example, we introduce the (meta)$^\omega$ level which enables the user to replace the elements of the `default-system-object-table` with interpreted RbCl objects.

The (Meta)$^\omega$ Level

The behavior of 3-Lisp programs is defined by the infinite reflective tower of meta-circular interpreters. In practice, for efficient execution, the reflective tower of 3-Lisp is realized by a run-time interpreter kernel written in another language L. We refer to such an interpreter as the *tower interpreter of 3-Lisp in L*. A tower interpreter of 3-Lisp can also be written in 3-Lisp itself (without using its reflective facilities)[18]. This system would be the tower interpreter of 3-Lisp in 3-Lisp.

In the same way, we can write a tower interpreter of RbCl in RbCl. As explained in Section 4.4.1, the reflective tower of the plain RbCl is implemented on the C++ language, so the plain RbCl metasystem is a tower interpreter of RbCl in C++. Obviously we can use the RbCl language itself to write a tower interpreter of RbCl: we call this system the tower interpreter of RbCl in RbCl (Fig. 4.3). The reader should not confuse two RbCl systems, the *interpreted RbCl* and the *interpreting RbCl*. The interpreted RbCl is implemented by the tower interpreter, and the tower interpreter is only an application program of the interpreting RbCl. The baselevel of the interpreting RbCl may be called *the (meta)$^\omega$ level* of the interpreted RbCl, because it implements the entire reflective tower of the interpreted RbCl. The programming at the (meta)$^\omega$ level allows, for example, debugging of reflective programs, experiments on reflective facilities using RbCl, rather than using other low level programming languages such as C++. New reflective systems — for example, a multi-user system supporting each user to have his/her own reflective tower — could be efficiently realized within the RbCl framework.

```
(defun Tower-in-RbCl (obj-code)
  ;; This level is the (meta)^(omega) level
  ;; interpreted by the (meta)^(omega+1) level.
  (let (level-manager
        obj)
    ;; Generate a new metasystem.
    (setq level-manager
       [(G metasystem-generator) <== [:call]])
    ;; Shift to the generated level.
    [level-manager <== [:shift-to-meta]]
    ;; Now, this level is the new metalevel
    ;;still interpreted by the (meta)^(omega+1) level.
    (setq obj [(G make-local-object)
               <== [:call obj-code]])
    [(G active-queue) <== [:enqueue obj]]
    ;; Start interpretation of the new baselevel.
    [(G scheduler) <== [:call]]
    ))
```

Figure 4.4: A tower interpreter in RbCl.

Fig. 4.4 is an example of tower interpreter in RbCl. `Obj-code` is the program code of the start-up object[4]. `(G metasystem-generator)` denotes a system object that generates a new metasystem by the scheme described in Section 4.4.4. `Metasystem-generator` returns the level manager of the generated metasystem when it receives a `[:call]` message. We can implement a complete tower interpreter in such a few program steps because `metasystem-generator` has an ability to create a complete metasystem explicitly.

It is likewise possible in other reflective language systems to write the tower in-

[4]The RbCl metasystem represents program code of RbCl objects as metalevel objects.

terpreter of itself. However, it would require hundreds of lines of program code to implement the run-time kernel explicitly. Furthermore, the execution of interpreted reflective tower would be much slower than the original one, and the user must change the program code of run-time kernel (in an ad hoc way) to experiment with new reflective facilities. In contrast, the execution speed of programs in the interpreted RbCl is as fast as the interpreting RbCl, because the entire reflective tower is directly implemented by the (meta)$^{\omega}$ level. Furthermore, the user could replace some elements of `(G default-system-object-table)` with (meta)$^{\omega}$ level RbCl objects to modify the behavior of the entire reflective tower. So, for example, the user could use RbCl to implement a debugger of reflective programs rather than using other low level programming languages such as C++.

4.5 Realization of Linguistic Symbiosis

In order to realize the linguistic symbiosis, the implementation gap between RbCl and C++ is absorbed by metalevel objects called *intermediate pointers* (Fig. 4.5). Each reference between baselevel RbCl objects and baselevel C++ objects is represented by an intermediate pointer object at the metalevel. The user can modify/extend the behavior of intermediate pointers as well as other metalevel objects. Intermediate pointers perform the following things:

1. Conversion of message passing protocols between the two languages.
2. Conversion of data representation between the two languages.
3. Level shifting (switching system object tables) by sending messages to the level manager of the metalevel.
4. Coroutine management using the thread library of the C language.

Currently, C++ objects must meet the following requirements:

- A C++ class must be defined to be a subclass of the class `L`. To access a C++ object, the RbCl system needs to invoke the methods provided by the class `L`.

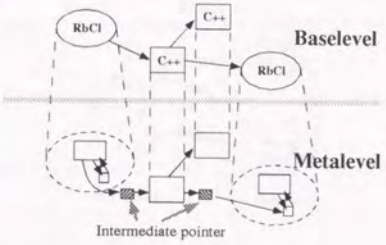


Figure 4.5: Implementation of the linguistic symbiosis.

- All the methods which the RbCl may invoke must be defined as virtual functions.
- All the arguments of the methods which the RbCl may invoke must have the type either basic type (e.g., integer, Boolean, etc.) or the pointer type referencing an instance of the class L.

The intermediate pointers between two object types are generated in object-oriented manner. We can regard the baselevel world as separated into two language worlds, the RbCl world and the C++ world. When a reference goes across the *language boundary*, there must be an intermediate pointer. (Fig. 4.6) When a reference of an RbCl object R goes across the language boundary from RbCl to C++, the intermediate pointer at that boundary traps that message sending and sends the message `:c2r` to R. The return value of this message sending form is new intermediate pointer P which will convert a C++ virtual function call to an RbCl message sending. P can be regarded as R itself from the point of view of the C++ world.

The intermediate pointers convert message passing protocols as follows. The method `:foo` with one method defined for the intermediate pointer class looks like:

```
LP Lc2r_pointer::foo(LP arg1){
  level_manager->shift_to_meta();
  LP r_ret = c2r_nowsend(list(builtin_symbol(S_foo), arg1->r2c()), r_obj);
```

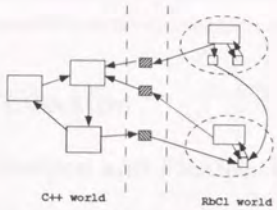


Figure 4.6: The language boundary and intermediate pointers.

```
  LP ret = r_ret->c2r();
  level_manager->shift_to_base();
  return ret;
}
```

`Lc2r_pointer` is the class of the intermediate pointer from the C++ world to the RbCl world. `LP` is a pointer type to metalevel objects. This method `:foo` translates the C++ representation of a message and its arguments into the RbCl representation. Because instances of different C++ classes receive different set of messages, every class needs the class-specific definition of the intermediate pointer class. This burdensome task of producing an intermediate pointer class may be automatically done[5]. Message passing from an RbCl object to a C++ object is realized in a similar way.

The function `c2r_nowsend` appeared at above program list performs an RbCl message sending. It also manages the coroutines using thread library. All baselevel RbCl objects are interpreted at the metalevel on one thread, and baselevel C++ objects are directly implemented by metalevel C++ objects that run at the metalevel on the other threads. When a baselevel RbCl object sends a message to a baselevel C++ object, the intermediate pointer at the metalevel switches the active thread to a new thread where the corresponding metalevel C++ object runs. When control is passed back to a baselevel RbCl object, the interpreter thread at the metalevel is

[5]This mechanism is not implemented yet. Currently, only instances of class `L` can used as RbCl objects.

reactivated. Message passing between C++ objects is performed efficiently with an ordinary virtual function invocation without using intermediate pointers.

4.6 Garbage Collection

4.6.1 Garbage Collection and Finalization

Garbage collection of RbCl objects is implemented using the finalization facility. Each object may have a *finalization form* [6], which will be executed before the object is garbage collected. (There is an example program using a finalization form in Appendix A.1.) In the current prototype system, C++ objects are garbage collected by reference counting scheme [7].

The garbage collection of RbCl objects is implemented as follows. When a reference to an object is discarded at a baselevel, the pointer object which represents that reference become garbage because no metalevel objects refer it yet. The finalization form of the pointer object decrement the reference count of the object it had been referred. If the reference count becomes zero, the corresponding baselevel object is regarded as garbage.

4.6.2 Changing Garbage Collection Scheme

Plain RbCl does not support distributed garbage collection. The user can implement various distributed garbage collection schemes by extending pointer and port objects. For example, group reference counting scheme[7], that can collect some distributed cyclic garbage, can be implemented. The following program is an example to implement a simple distributed garbage collection using reference counting.

In plain RbCl, a remote pointer is represented as a metalevel object called `Lglobal_pointer`, that contains an encoder of the node where the target object resides and local id of the object.

[6] In C++ terminology, it is a *destructor*[21].

[7] It is implemented by using *smart pointer* and destructor of it .

When a Lglobal_pointer is regarded as a garbage at the metalevel, the finalization form of the Lglobal_pointer is executed. In the following program, the finalization form sends a packet whose tag is Ddiscard. The receiver of the packet send a :decode message to the system object named Ddiscard, that decrement the reference counter of the target object.

```
(metalevel-exec
 (defclass pointer-mixin
    (unwind-protect
        (script
         (=> [:set-super x]
             (setq self nil)
             !(setq super x))
         (=> any @ rep-des
             [super <= any @ rep-des]))
      ;; The following form is executed
      ;; when this object is regarded as a garbage.
      (let ((encoder [super <== [:get-encoder]])
            (localid [super <== [:get-localid]])
            (buf [(G |Lio_buffer|) <== [:call]]))
        [buf <== [:put-int [(reify 'Ddiscard)
                            <== [:symbol-id]]]]
        [buf <== [:put-int localid]]
        [encoder <== [:nacl-send buf]]
        )))
(setf (G |Lglobal_pointer|)
      (extend-generator pointer-mixin
                        (G |Lglobal_pointer|)))
(defclass Ddiscard
    (script
     (=> [:decode buf]
         (let ((obj (localid-to-obj [buf <== [:get-int]])))
           ;; Decrement the reference count.
           [obj <== [:user-discard]]
           !(reify nil)
           ))))
(setf (G Ddiscard) (new Ddiscard)))
```

Chapter 5

Performance Evaluation

5.1 Current Status of Implementation

Currently, the prototype system of RbCl has been implemented on SparcStations connected with an Ethernet. The lwp (light weight process) library of SunOS is used to implement the linguistic symbiosis. The following RbCl features are implemented.

- Concurrent execution of objects.
- The linguistic symbiosis.
- Inter-node communication.
- Reflective facilities.

5.2 Overhead of Non-Reflective Execution

During executing non-reflective programs, indirect accesses to metalevel objects such as virtual function call become execution overhead. This is a result of tradeoff between flexibility and efficiency. As demonstrated in the previous Sections, RbCl provides highly flexibility that is worth this overhead. The user who thinks that the performance is more important can decrease this overhead as follows. For example, suppose a user wants to improve performance of inter-node communications. The inter-node

communication module consists of several system objects, as described at Section 4.3. The user may replace system objects in inter-node communication module so that they communicate directly (communicate with ordinary function calls rather than virtual function calls) each other. In this case, flexibility of the inter-node communication module is lost instead of efficiency.

In this way, RbCl perfectly leaves the chance of tradeoff between flexibility and efficiency for the user because there are no run-time kernels that restrict the range of customization.

It is also possible to decrease overhead of indirection without decrease flexibility by employing compiler optimization techniques for sequential object-oriented languages[2]. It decreases overhead of indirect message sending.

5.3 Overhead of Reflective Execution

When the user replaces a metalevel system object with a C++ object (or a compiled RbCl object), there is absolutely no overhead except for *default* overhead described above.

When the user replaces a metalevel system object with an RbCl object, the new system object will be interpreted by the meta meta system, so the performance of the system will be degraded. The overhead of message sending between RbCl objects and C++ objects is also not so small because it causes data conversion of arguments, thread switching and level shifting.

5.4 Performance Improvement Using Reflective Programming

We measured the cost of the asynchronous remote message sending in three cases. The content of the message is a list of two integers. The first case is measured in the default metasystem of plain RbCl. In the second case, inter-node communication routines are customized as described at Section 4.3.6. Two system objects are replaced

with the user defined RbCl objects that are specialized to the message type, lists of two integers. In the third case, two system objects are replaced with C++ objects whose behavior is almost same to the RbCl objects of the second case.

- The plain RbCl metasystem : 9.8 ms
- Replaced with RbCl objects : 600 ms
- Replaced with C++ objects : 7.4 ms

In the second case, the speed is slower than the default system, but this customization scheme (replacing with RbCl objects) can be used for prototyping the metalevel. If the user defined RbCl objects are compiled to the native code, the cost will become near to that of the third case.

Chapter 6

Conclusions

RbCl realizes the reflective tower efficiently with finite computing resources without a run-time kernel. This is achieved by employing a simple mechanism called *system object tables* and a novel facility called *linguistic symbiosis* with C++ objects.

Because RbCl is a kernel-less system, the user can change the behavior of the system up to the restriction imposed by the operating system and hardware within the RbCl framework. This implies that every possible run-time facility can be provided as libraries or applications written in RbCl.

The system object table of the metalevel on a node is the CCSR of the baselevel on the node. The system objects that realize the basic behavior of the baselevel are the elements in the system object table of the metalevel. The benefits of the system object tables are as follows:

1. The system object table of the metalevel is the mechanism that makes the RbCl metasystem kernel-less. All the system objects are indirectly accessed through the system object table. Therefore, the user can change the behavior of the metasystem by replacing elements of the system object table with user-defined objects. The overhead of this indirection is negligible because access to a system object can be achieved in only a few instruction steps.

2. The system object tables represent the name spaces of system objects that are equivalently accessed by two languages, C++ and RbCl.

3. Level shifting, that is, switching the current name space of the system objects, can be efficiently implemented.

The benefits of the linguistic symbiosis with the C++ objects are as follows:

1. An efficient reflective system can be easily implemented. The metasystem can be constructed only by the C++ objects that are efficiently executed, and the user can manipulate the metalevel C++ objects just as RbCl objects.

2. The linguistic symbiosis also serves as a foreign language interface to the C(++) language that enables the user to directly use the system calls provided by the operating system.

It should be noted that the implementation techniques used by RbCl are easily applicable to other systems. The reflective facilities of RbCl are implemented using only simple mechanisms of the implementation language: the system object table is only a simple array of objects, and the linguistic symbiosis requires only the coroutine facility for its implementation.

Design and implementation of the following mechanisms are the future work.

- Exception handling and real time features.
  Practical programming system should provide these features. [5] proposes such features for object-oriented concurrent languages. Interrupt manager should be introduced into the metasystem to implement these features.

- Compiler.
  Currently, the user defined object is executed by the interpreter and the differential programming is supported naively. In addition, all objects, including very small one such as cons cells, are operated through virtual function invocations. The implementation techniques to make object-oriented languages efficient[2] are very effective to reduce this overhead. Compiling concurrent objects to efficient code is difficult and very important research issue. Development of an open-ended compiler that can be used as a platform of experiments about compiling techniques would contribute to such research.

Bibliography

[1] Burdorf, C. & J. Marti, "Non-Preemptive Time Warp Scheduling Algorithms", Operating Systems Review, vol.24,pp7-18,Apr.1990.

[2] Chambers, C. & Unger, D., "Making Pure Object-Oriented Languages Practical," In Proc. of OOPSLA '91, ACM, October, pp.1-15, 1991.

[3] Etsuya Shibayama, Yonezawa, A., & Ichisugi, Y. The ABCL/1 user's guide. In [27].

[4] Ferber, J., "Computational Reflection in Class based Object Oriented Languages," In Proc. of OOPSLA '89, ACM, pp.317–326, October, 1989.

[5] Ichisugi, Y. & Yonezawa, A. Exception handling and real time features in an object-oriented concurrent language. In *Concurrency: Theory, Languages and Architecture*, volume 491 of *Lecture Notes in Computer Science*, pages 92–109. Springer-Verlag, 1990.

[6] Ichisugi, Y., Matsuoka, S., Watanabe, T. & Yonezawa, A. An object-oriented concurrent reflective architecture for distributed computing environments (extended abstract). In *Proceedings of 29th Annual Allerton Conference on Communication, Control and Computing, Allerton Illinois, 1991*, 1991.

[7] Ichisugi, Y. & Yonezawa, A. Distributed garbage collection using group reference counting. In Ikuo Nakata and Masami Hagiya, editors, *Software Science and Engineerings*, pages 212–226. World Scientific, 1992.

[8] Ichisugi, Y., Matsuoka, S. & Yonezawa, A. RbCl: A Reflective Object-Oriented Concurrent Language without a Run-time Kernel. In Proc. of IMSA'92 International Workshop on Reflection and Meta-level Architecture, Tokyo, Nov.4–7, 1992.

[9] Ishikawa, Y., "Reflection Facilities and Realistic Programming," SIGPLAN Notices, Vol.26, No.8, Aug.1991, pp.101-110.

[10] Jefferson, D.R., "Virtual Time", ACM Trans. Programming Languages and Systems, vol.7,no.3,pp.404-425, 1985.

[11] Kiczales, G., des Rivières, J. & Bobrow, D. G., The Art of Metaobject Protocol, MIT Press, 1991.

[12] Lieberman., H., "Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems," In Proc. of OOPSLA '86, ACM, September-October, pp.214–223 1986.

[13] Maes, P., "Concepts and Experiments in Computational Reflection", In Proc. OOPSLA '87, ACM, pp. 147–155, 1987.

[14] Masuhara, H., Matsuoka, S., Watanabe, T. & Yonezawa, A. "Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently", In Proc. of OOPSLA'92, ACM, pp.127-144, 1992.

[15] Matsuoka, S., Watanabe, T. & Yonezawa, A., "Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming", In Proc. of ECOOP '91, pp. 231–250, Lecture Notes in Computer Science, 512, Springer, 1991.

[16] Matsuoka, S., Watanabe, T., Ichisugi, Y.& Yonezawa, A., "Object-Oriented Concurrent Reflective Architectures," In Proc. of ECOOP Workshop on Object-Based Concurrent Programming, Geneve, Switzerland, July, 1991, also in a LNCS 612, Springer, 1992.

[17] Okamura, H., Ishikawa, Y. & Tokoro, M., "AL-1/D: A Distributed Programming System with Multi-Model Reflection Framework," In Proc. of IMSA'92 International Workshop on Reflection and Meta-level Architecture, Tokyo, Nov.4–7, 1992.

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

[19] Rose, J., "A Minimal Metaobject Protocol for Dynamic Dispatch", In Proc. of the OOPSLA'91 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming, October, 1991.

[20] Smith, B. C., "Reflection and Semantics in Lisp", In Conference Record of ACM POPL '84, pp. 23-35, 1984.

[21] Stroustrup,B., "THE C++ PROGRAMMING LANGUAGE SECOND EDITION", Addison Wesley, 1991.

[22] Wand,M. & Friedman,D., "The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower," Meta-Level Architectures and Reflection, pp.111–134, Elsevier Science, North-Holland, 1988.

[23] Watanabe, T. & Yonezawa, A., "Reflection in an Object-Oriented Concurrent Language", In Proc. ACM OOPSLA '88, pp. 306-315, 1988, (revised version in [27]).

[24] Watanabe, T. & Yonezawa, A., "An Actor-Based Metalevel Architecture for Group-Wide Reflection", In Proc. REX School/Workshop on Foundations of Object-Oriented Languages (REX/FOOL), Lecture Notes in Computer Science, 489, Springer, 1991.

[25] Yokote,Y., Teraoka,F. & Tokoro,M., "A Reflective Architecture for an Object-Oriented Distributed Operating System," Proceedings of European Conference on Object-Oriented Programming, July, 1989.

[26] Yokote,Y., "The Apertos Reflective Operating System: The Concept and Its Implementation", In Proc. of OOPSLA'92, ACM, October, 1992.

[27] Yonezawa, A.(Ed.), ABCL: An Object-Oriented Concurrent System, MIT Press, 1990.

[28] Yonezawa, A. & Watanabe, T., "An Introduction to Object-Based Reflective Concurrent Computations", In Proc. of 1989 Workshop on Object-Based Concurrent Programming, ACM SIGPLAN Notices, 24(4), pp. 50-54, 1989.

Appendix A

Examples Programs in RbCl

In this appendix, we will present example programs of RbCl.

A.1 Non-reflective Programs

In this section, we explain some non-reflective language features.

A.1.1 Prime numbers

This is a simple example of RbCl program, that print the prime numbers.

```
(progn
  (defclass filter-class
      (let (next-filter)
        (script
         (=> [:check n]
             (print n)
             (setq next-filter (new filter-class))
             (script
              (=> [:check m]
                  (unless (= (- m (* (/ m n) n)) 0)
                    [next-filter <= [:check m]])))))))
  (let ((filter (new filter-class))
        (x 1))
```

```
    (loop [filter <= [:check (incf x)]])))
```

```
Output:

2
3
5
7
11
13
...
```

A.1.2 Linguistic Symbiosis

The system object table contains parts of the metasystem. A system object named `Lcons` generates objects which represent cons cells.

```
(progn (setq foo (G |Lcons|))
       (print foo)      ; This is a C++ object at baselevel
       (print (setq bar [foo <== [:call foo foo]]))
       (print [bar <== [:car]]))
```

```
Output:

^(Lr2c_pointer )
^(Lr2c_pointer )
^(Lr2c_pointer )
```

When a C++ object is passed as an argument of the `print` function, "`^(Lr2c_pointer )`" is printed. This is actually an intermediate pointer (described at Section 4.5) from a RbCl object to a C++ object.

A.1.3 Coroutine Libraries

In RbCl, coroutine library provided for C(++) is encapsulated in system objects. Although coroutine libraries are usually used for metasystem, it can be used at base-

level. This is an example that shows that the semantics of the baselevel and the metalevel in plain RbCl are completely equivalent.

```
(progn
  (defclass foo
      (script
       (=> [:call]
           (dotimes (n 3)
             (dotimes (m 2)
               (print (list :foo n m)))
             [bar-thread <== [:yield]])
           )))
  (defclass bar
      (script
       (=> [:call]
           (dotimes (n 3)
             (dotimes (m 3)
               (print (list :bar n m)))
             [foo-thread <== [:yield]])
           )))
  (setq foo-thread [(G |LCthread|) <== [:call (new foo)]])
  (setq bar-thread [(G |LCthread|) <== [:call (new bar)]])
  [foo-thread <== [:yield]])
```

Output:

```
(:FOO 0 0)
(:FOO 0 1)
(:BAR 0 0)
(:BAR 0 1)
(:BAR 0 2)
(:FOO 1 0)
(:FOO 1 1)
(:BAR 1 0)
(:BAR 1 1)
(:BAR 1 2)
(:FOO 2 0)
```

```
(:FOO 2 1)
(:BAR 2 0)
(:BAR 2 1)
(:BAR 2 2)
```

A.1.4 Finalization

RbCl provides a finalization facility as mentioned in Section 4.6. The semantics of unwind-protect (special form of Common Lisp) is extended to support the finalization. When an object is garbage collected, all unwind-protected form will be executed.

```
(progn
  (defclass foo
      (let ((obj nil))
        (unwind-protect
             (script
              (=> [:set-obj x]
                  (setq obj x)))
          (print (list 'obj= obj)))))
  (setq x (new foo))
  (setq y (new foo))
  [x <= [:set-obj y]]
  (print (list x y))
  (setq y nil)
  (print (list x y))
  (setq x nil)
  (print (list x y)))


Output:

(^(Llocal_pointer ) ^(Llocal_pointer ))
(^(Llocal_pointer ) NIL)
(NIL NIL)
(OBJ= ^(Llocal_pointer ))
```

```
(OBJ= NIL)
```

A.1.5 Remote Message Sending

This is an example of distributed feature explained in Section 4.3.

```
(if (equal (hostname) "server-host")
    (progn
      (print (hostname))
      (funcall
       (meta-lambda ()
                    [(G |Ltcpip_connection_request_handler|)
                     <== [:call (reify 6667)]]))
      (defclass foo
          (script
           (=> [:foo n]
               (print n)
               !(1+ n))))
      (setf (G foo) (new foo))
      (print [(G foo) <== [:foo 12]])
      )
    (progn
      (print (hostname))
      (setq node (connect (node-address "server-host" 6667)))
      (print (setq remote-foo (remote-system-object 'foo node)))
      (print [remote-foo <== [:foo 34]])))
```

Output of server-host:

```
"server-host"
12
13
34
```

Output of client-host:

```
"client-host"
35
```

A.1.6 Startup Scheduling

This example explains how to start up scheduling. This scheme is used in some examples appears below.

```
(progn
  (defclass base-base
      (dotimes (n 10) (print n)))
  (print [(G active-queue) <== [:empty]])
  [(G make-local-object) <== [:call (reify base-base)]]
  (print [(G active-queue) <== [:empty]])
  [(G scheduler) <== [:call]]
  ;; never return
  (print :end))
```

Output:

```
T
NIL
0
1
2
3
4
5
6
7
8
9
```

A.2 Reflective Programs

A.2.1 Meta-lambda

This is a simple example of using `meta-lambda`. The message sending `:fix` to a metalevel object which represents an integer returns its representing integer value.

```
(progn (print (funcall
               (meta-lambda (x)
                            (print x)
                            (print [x <== [:fix]])
                            x)
               345)))
```

```
Output:

^(Lr2c_pointer )      ; output of (print x)
345                   ; output of (print [x <== [:fix]])
345                   ; output of (print (funcall ...))
```

A.2.2 System Object Tables

This example shows each level has its own system object table.

```
(progn
  (print (reflect (G |Lcons|)))
  (funcall (meta-lambda
             ()
             [[(G baselevel-system-object-table)
               <== [:aset [(reify '|Lcons|) <== [:symbol-id]]
                          (reify 123)]]]
             (reify nil)
             ))
  (print (reflect (G |Lcons|))))
```

```
Output:

#<builtin generator>
```

123

A.2.3 Extending Objects

This is an example of differential programming described at Section 3.3.3. A system object Lcons at the baselevel is extended.

```
(progn
  (defclass trace-mixin
      (script
       (=> [:set-super x]
           !(setq super x))
       (=> any @ rep-des
           (print any)
           [super <= any @ rep-des])))
  (defun extend (mixin parent)
    [mixin <== [:set-super parent]]
    (do ((super parent [super <== [:get-super]]))
        ([super <== [:null]])
      [super <== [:set-self mixin]])
    mixin)
  (setf (G |Lcons|) (extend (new trace-mixin) (G |Lcons|)))
  [(G |Lcons|) <== [:print]])

Output:

(:PRINT)
(:PRIN1 ^(Lr2c_pointer ))
#<builtin generator>
```

A.2.4 Extending Cons Generators

This is an example of extending a generator. When cons generator received some message, such as `:call`, the message tag will be printed. Note that this metalevel program also creates cons cells, but it will not print trace message because the cons generator at the meta meta level is still default one.

```
(progn
  (funcall
   (meta-lambda
    ()
    (defclass trace-mixin
        (script
         (=> [:set-super x]
             !(setq super x))
         (=> any @ rep-des
             (print (car any))
             [super <= any @ rep-des])))
    (defun extend (mixin parent)
      [mixin <== [:set-super parent]]
      (do ((super parent [super <== [:get-super]]))
          ([super <== [:null]])
        [super <== [:set-self mixin]])
      mixin)
    (defclass generator-mixin
        (let (mixin-class)
          (script
           (=> [:set-super x]
               !(setq super x))
           (=> [:init-generator-mixin class]
               !(setq mixin-class class))
           (=> any
               !(extend (new mixin-class) [super <== any])))))
    (defun extend-generator (mixin-class generator)
      (let ((new-generator (extend (new generator-mixin) generator)))
        [new-generator <== [:init-generator-mixin mixin-class]]
        new-generator))
    (setf (G |Lcons|)
          (extend-generator
           trace-mixin (G |Lcons|)))
    (reify nil)))
  (print (list 1 2 3)))
```

Output:

```
:PRIN1
(:CDR
:CONSP
:CAR
1 :CDR
:CDR
:CONSP
:CAR
2 :CDR
:CDR
:CAR
3:CDR
)
```

A.2.5 Extending Port Generator

This example is similar to above one. When a port generator receives messages, the name of the message will be printed. Note that definition of trace-mixin is equivalent to the above example.

```
(progn
  (funcall
   (meta-lambda
    ()
    (defclass trace-mixin
        (script
         (=> [:set-super x]
             !(setq super x))
         (=> any @ rep-des
             (print (car any))
             [super <= any @ rep-des])))
    (defun extend (mixin parent)
      [mixin <== [:set-super parent]]
      (do ((super parent [super <== [:get-super]]))
          ([super <== [:null]])
        [super <== [:set-self mixin]])
```

```
    mixin)
  (defclass generator-mixin
     (let (mixin-class)
       (script
        (=> [:set-super x]
            !(setq super x))
        (=> [:init-generator-mixin class]
            !(setq mixin-class class))
        (=> any
            !(extend (new mixin-class) [super <== any])))))
  (defun extend-generator (mixin-class generator)
    (let ((new-generator (extend (new generator-mixin) generator)))
      [new-generator <== [:init-generator-mixin mixin-class]]
      new-generator))
  (setf (G |Lport|)
        (extend-generator
         trace-mixin (G |Lport|)))
  (reify nil)))
(defclass foo
   (script
    (=> [:foo]
        (print :foo))))
(setq x (new foo))
[x <= [:foo]]
(setq x nil))
```

Output:

```
:SET-BODY
:USER-REFER
:WAIT-FOR-MESSAGE
:SEND
:NEXT-VALUE
:FOO
:WAIT-FOR-MESSAGE
```

A.2.6 Changing Scheduling Policy

This is a more complex example. Objects and the active-queue are extended to implement scheduling with priority. (Cf. Section 4.2) After extending behavior of objects, a function `set-priority` becomes available. The function `set-priority` just send a priority value specified as an argument to the currently running object. This example creates two objects, `p1` and `p2` with different priority.

```
(progn
  (defclass p1
      (set-priority 6)
    (dotimes (n 3) (print :p1)))
  (defclass p2
      (set-priority 7)
    (dotimes (n 3) (print :p2)))
  (defclass boot
      (new p2)
    (new p1))


  (funcall
   (meta-lambda
    ()
    (defclass priority-mixin
        (let ((priority 5)) ; default priority
          (script
           (=> [:set-super x]
               !(setq super x))
           (=> [:set-priority n]
               !(setq priority n))
           (=> [:get-priority]
               ! priority)
           (=> any @ rep-des
               [super <= any @ rep-des])))))
    (defun set-priority (n)
      (funcall (meta-lambda (n-obj)
                            [(G current-object)
                             <== [:set-priority (reflect n-obj)]]
```

```
                    (reify nil))
            n)
  )
(defun insert-obj (obj queue)
  (cond ((null queue)
         (list obj))
        ((< [obj <== [:get-priority]]
            [(car queue) <== [:get-priority]])
         (cons obj queue))
        (t
         (cons (car queue)
               (insert-obj obj (cdr queue))))))
(defclass priority-queue
    (let ((queue nil))
      (script
       (=> [:empty]
           !(null queue))
       (=> [:enqueue obj]
           (setq queue
                 (insert-obj obj queue))
           ! nil)
       (=> [:dequeue]
           ! (pop queue)))))
;;
(defun extend (mixin parent)
  [mixin <== [:set-super parent]]
  (do ((super parent [super <== [:get-super]]))
      ([super <== [:null]])
    [super <== [:set-self mixin]])
  mixin)
(defclass generator-mixin
    (let (mixin-class)
      (script
       (=> [:set-super x]
           !(setq super x))
       (=> [:init-generator-mixin class]
           !(setq mixin-class class))
```

```
        (=> any
            !(extend (new mixin-class) [super <== any])))))
    (defun extend-generator (mixin-class generator)
      (let ((new-generator (extend (new generator-mixin) generator)))
        [new-generator <== [:init-generator-mixin mixin-class]]
        new-generator))

    (let ((old-queue (G active-queue)))
      (setf (G active-queue)
            (new priority-queue))
      (setf (G |Lobject|)
            (extend-generator priority-mixin
                              (G |Lobject|)))
      [(G make-local-object)
       <== [:call (reify boot)]]
      (print :start)
      [(G scheduler) <== [:call]]
      ;; never return
      ))))
```

Output:

```
:START
:P1
:P1
:P1
:P2
:P2
:P2
```

A.3 Extending Reflective Facilities

One of the most notable characteristics of RbCl is that the user can modify/extend even reflective facilities themselves. The following programs are examples of modify/extend the reflective facilities.

A.3.1 Extending Intermediate Pointers

Intermediate pointers, that realizes the linguistic symbiosis are also customizable. This is an example to trace intermediate pointers at metalevel.

```
(progn
  (funcall
   (meta-lambda
    ()
    (setq *id* 0)
    (defclass trace-mixin
        (let ((id (incf *id*)))
          (print (list id :created))
          (script
           (=> [:set-super x]
               !(setq super x))
           (=> any @ rep-des
               (print (list id :delegate (car any)))
               [super <= any @ rep-des])))))
   (defun extend (mixin parent)
     [mixin <== [:set-super parent]]
     (do ((super parent [super <== [:get-super]]))
         ([super <== [:null]])
       [super <== [:set-self mixin]])
     mixin)
   (defclass generator-mixin
       (let (mixin-class)
         (script
          (=> [:set-super x]
              !(setq super x))
          (=> [:init class]
```

```
            !(setq mixin-class class))
        (=> any @ rep-des
            (let ((a-mixin (new mixin-class))
                  (a-cont (new mixin-cont)))
              [a-cont <== [:init rep-des a-mixin]]
              [super <= any @ a-cont])))))
  (defclass mixin-cont
      (script
       (=> [:init rep-des a-mixin]
           !nil
           (send rep-des (extend a-mixin (next-value me))))))
  (defun extend-generator (mixin-class generator)
    (let ((new-generator (extend (new generator-mixin) generator)))
      [new-generator <== [:init mixin-class]]
      new-generator))
  (setf (G |Lr2c_pointer|)
        (extend-generator trace-mixin
                          (G |Lr2c_pointer|)))
  (setf (G |Lc2r_pointer|)
        (extend-generator trace-mixin
                          (G |Lc2r_pointer|)))
  (reify nil)
  ))

(defclass foo
    (script
     (=> any @ rep-des
         [(G |Lcons|) <= any @ rep-des])))
(print 1)
(print (G |Lcons|))
(print 2)
(setq x [(G |Lcons|) <== [:call (G |Lcons|) (new foo)]])
(print x)
(print 3)
[x <== [:print]]
)
```

```
Output:

1
(1 :CREATED)
(1 :DELEGATE :PRIN1)
^(Lr2c_pointer )
2
(2 :CREATED)
(3 :CREATED)
(3 :DELEGATE :EXPORT-COPY)
(2 :DELEGATE :SEND)
(3 :DELEGATE :C2R)
(4 :CREATED)
(4 :DELEGATE :PRIN1)
^(Lr2c_pointer )
3
(4 :DELEGATE :SEND)
((5 :CREATED)
(5 :DELEGATE :SEND)
#<builtin generator>(6 :CREATED)
(6 :DELEGATE :SEND)
 . (7 :CREATED)
(8 :CREATED)
(7 :DELEGATE :EXPORT-COPY)
(8 :DELEGATE :SEND)
(7 :DELEGATE :C2R)
#<builtin generator>)
```

A.3.2 Extending a Level Manager

In this example, a level manager at the metalevel is extended to monitor the level shifting between the baselevel and the metalevel.

```
(progn
  (funcall (meta-lambda
            ()
            (setq *id* 0)
```

```
          (defclass trace-mixin
              (let ((id (incf *id*)))
                (print (list id :created))
                (script
                 (=> [:set-super x]
                     !(setq super x))
                 (=> any @ rep-des
                     (print (list id :delegate (car any)))
                     [super <= any @ rep-des]))))
          (defun extend (mixin parent)
            [mixin <== [:set-super parent]]
            (do ((super parent [super <== [:get-super]]))
                ([super <== [:null]])
              [super <== [:set-self mixin]])
            mixin)
          (setf (G level-manager)
                (extend (new trace-mixin)
                        (G level-manager)))
          (G |Lcons|) ;dummy
          ))
  (defclass foo
      (script
       (=> any
           (print :foo)
           !(G |Lcons|))))
  (print 1)
  (print (G |Lcons|))
  (print 2)
  (setq x [(G |Lcons|) <== [:call (G |Lcons|) (new foo)]])
  (print x)
  (print 3)
  [x <== [:print]]
  )
```

Output:

```
(1 :CREATED)
```

```
1
^(Lr2c_pointer )
2
(1 :DELEGATE :SHIFT-TO-BASE)
(1 :DELEGATE :SHIFT-TO-META)
^(Lr2c_pointer )
3
(1 :DELEGATE :SHIFT-TO-BASE)
((1 :DELEGATE :SHIFT-TO-META)
:FOO
(1 :DELEGATE :SHIFT-TO-BASE)
#<builtin generator> (1 :DELEGATE :SHIFT-TO-META)
:FOO
(1 :DELEGATE :SHIFT-TO-BASE)
(1 :DELEGATE :SHIFT-TO-META)
:FOO
(1 :DELEGATE :SHIFT-TO-BASE)
#<builtin generator>(1 :DELEGATE :SHIFT-TO-META)
:FOO
(1 :DELEGATE :SHIFT-TO-BASE)
 . (1 :DELEGATE :SHIFT-TO-META)
:FOO
(1 :DELEGATE :SHIFT-TO-BASE)
#<builtin generator>)
(1 :DELEGATE :SHIFT-TO-META)
```

A.3.3 Extending all Level Managers

This is an example extending the all level managers in the reflective tower using meta omega level described in Section 4.4.5.

```
(progn
  (setq *id* 0)
  (defclass trace-mixin
      (let ((id (incf *id*)))
        (print (list id :created))
        (script
```

```
      (=> [:set-super x]
          !(setq super x))
      (=> any @ rep-des
          (print (list id :delegate (car any)))
          [super <= any @ rep-des])))))
(defun extend (mixin parent)
  [mixin <== [:set-super parent]]
  (do ((super parent [super <== [:get-super]]))
      ([super <== [:null]])
    [super <== [:set-self mixin]])
  mixin)
(defclass generator-mixin
    (let (mixin-class)
      (script
       (=> [:set-super x]
           !(setq super x))
       (=> [:init class]
           !(setq mixin-class class))
       (=> any @ rep-des
           (let ((a-mixin (new mixin-class))
                 (a-cont (new mixin-cont)))
             [a-cont <== [:init rep-des a-mixin]]
             [super <= any @ a-cont])))))
(defclass mixin-cont
    (script
     (=> [:init rep-des a-mixin]
         !nil
         (send rep-des (extend a-mixin (next-value me))))))
(defun extend-generator (mixin-class generator)
  (let ((new-generator (extend (new generator-mixin) generator)))
    [new-generator <== [:init mixin-class]]
    new-generator))
[[(G default-system-object-table)
  <== [:aset [(reify '|Llevel_manager|)
              <== [:symbol-id]]
             (extend-generator
              trace-mixin
```

```
                (G |Llevel_manager|))]]]
  ;; create new metasystem
  [[(G metasystem-generator) <== [:call]] <== [:shift-to-meta]]
  ;; now, this level is new metalevel
  (defclass foo
     (script
      (=> any @ rep-des
          [(G |Lcons|) <= any @ rep-des])))
  (defclass startup
     (print 1)
    (print (G |Lcons|))
    (print 2)
    (setq x [(G |Lcons|) <== [:call (G |Lcons|) (new foo)]])
    (print x)
    (print 3)
    [x <== [:print]]
    (funcall (meta-lambda ()
                          (print 4)
                          (funcall (meta-lambda ()
                                                (print 5)
                                                (G |Lcons|) ;dummy
                                                ))
                          (reify nil)
                          )))
  [(G make-local-object) <== [:call (reify startup)]]
  [(G scheduler) <== [:call]]
  ;; never returns
  )
```

Output:

```
(1 :CREATED)
(1 :DELEGATE :SHIFT-TO-BASE)
(1 :DELEGATE :SHIFT-TO-META)
1
#(Lr2c_pointer )
2
```

```
(1 :DELEGATE :SHIFT-TO-BASE)
(1 :DELEGATE :SHIFT-TO-META)
^(Lr2c_pointer )
3
(1 :DELEGATE :SHIFT-TO-BASE)
((1 :DELEGATE :SHIFT-TO-META)
(1 :DELEGATE :SHIFT-TO-BASE)
(1 :DELEGATE :SHIFT-TO-META)
(1 :DELEGATE :SHIFT-TO-BASE)
#<builtin generator>(1 :DELEGATE :SHIFT-TO-META)
(1 :DELEGATE :SHIFT-TO-BASE)
(1 :DELEGATE :SHIFT-TO-META)
(1 :DELEGATE :SHIFT-TO-BASE)
 . (1 :DELEGATE :SHIFT-TO-META)
(1 :DELEGATE :SHIFT-TO-BASE)
#<builtin generator>(1 :DELEGATE :SHIFT-TO-META)
(1 :DELEGATE :SHIFT-TO-BASE)
)
(1 :DELEGATE :SHIFT-TO-META)
(2 :CREATED)
(2 :DELEGATE :SHIFT-TO-BASE)
(2 :DELEGATE :SHIFT-TO-META)
(2 :DELEGATE :SHIFT-TO-BASE)
(2 :DELEGATE :SHIFT-TO-META)
4
(3 :CREATED)
(3 :DELEGATE :SHIFT-TO-BASE)
(3 :DELEGATE :SHIFT-TO-META)
(3 :DELEGATE :SHIFT-TO-BASE)
(3 :DELEGATE :SHIFT-TO-META)
5
(3 :DELEGATE :SHIFT-TO-BASE)
(3 :DELEGATE :NULL)
(3 :DELEGATE :NULL)
(3 :DELEGATE :SHIFT-TO-META)
(3 :DELEGATE :SHIFT-TO-BASE)
(2 :DELEGATE :SHIFT-TO-BASE)
```

```
(2 :DELEGATE :NULL)
(2 :DELEGATE :NULL)
(2 :DELEGATE :SHIFT-TO-META)
(3 :DELEGATE :NULL)
(3 :DELEGATE :NULL)
(3 :DELEGATE :SHIFT-TO-META)
(3 :DELEGATE :SHIFT-TO-BASE)
(3 :DELEGATE :NULL)
(3 :DELEGATE :NULL)
(3 :DELEGATE :SHIFT-TO-META)
(3 :DELEGATE :SHIFT-TO-BASE)
(2 :DELEGATE :SHIFT-TO-BASE)
(2 :DELEGATE :NULL)
(2 :DELEGATE :NULL)
(2 :DELEGATE :SHIFT-TO-META)
(3 :DELEGATE :NULL)
(3 :DELEGATE :NULL)
(3 :DELEGATE :SHIFT-TO-META)
(3 :DELEGATE :SHIFT-TO-BASE)
(2 :DELEGATE :SHIFT-TO-BASE)
(2 :DELEGATE :NULL)
(2 :DELEGATE :NULL)
(2 :DELEGATE :SHIFT-TO-META)
(3 :DELEGATE :NULL)
(3 :DELEGATE :NULL)
(3 :DELEGATE :SHIFT-TO-META)
(3 :DELEGATE :SHIFT-TO-BASE)
(3 :DELEGATE :NULL)
(3 :DELEGATE :NULL)
(3 :DELEGATE :SHIFT-TO-META)
```

Appendix B

Rscheme

We present a full program list of *Rscheme*, which is a kernel-less language on Scheme based on a reflective architecture modeling that of RbCl.

The program list consists of two parts, one representing system object tables and the other run-time routines. Run-time routines mainly consists of three parts which are respectively implementing the interpreter, linguistic symbiosis and the reflective tower. All the run-time routines are elements of the system object table.

Rscheme has a primitive language construct for reflection, named `exec-at-metalevel`. The user can modify/extend elements of the system object table at the metalevel using this primitive. For example, the following code replaces the system function called `eval` with a user defined function. After executing this code, each expression will be printed out when it is evaluated.

```
;;; trace
(exec-at-metalevel
 ((lambda (old-eval)
    (setG 'eval
          (lambda (exp cont)
            (write exp)
            (newline)
            (old-eval exp cont))))
  (G 'eval)))
```

The value of `old-eval` is actually a Scheme procedure, but the linguistic symbiosis enables the user to manipulate the Scheme procedure just as an Rscheme function.

```
;;;; Rscheme
;;;-------------------------------------------------
;;;-------------------------------------------------
;;; The representation of system object tables.
;;; In the actual RbCl implementation, SOT is represented as
;;; an array and access functions are defined as macros.

(define SOT '())
(define (get-current-SOT) SOT)
(define (set-current-SOT table)
  (set! SOT table))
(define (G name)
  (let ((pair (assoc name SOT)))
    (if pair
        (cdr pair)
        ;; NOTE: The function "error" is not defined
        ;; at standard scheme.
        (error name "Undefined system object."))))
(define (setG name value)
  (set! SOT (cons (cons name value) SOT))
  name)
(define (copy-table table)
  (if (null? table)
      '()
      (cons (cons (car (car table)) (cdr (car table)))
            (copy-table (cdr table)))))
;;;-------------------------------------------------
;;;-------------------------------------------------
(define (boot)
  (set-current-SOT '())
  ;; The system object called default-SOT is the template of
  ;; metasystem represented as an alist.
  ;; All parts of the plain Rscheme interpreter are registered
  ;; as elements of default-SOT.
  (setG
   'default-SOT
   (list
```

```
    (cons 'default-SOT 'dummy)
;;;-------------------------------------------------
;;; evaluator
    (cons 'eval
          (lambda (exp cont)
            (cond ((symbol? exp)
                   ((G 'eval-var) exp cont))
                  ((pair? exp)
                   (case (car exp)
                     ((quote) ((G 'eval-quote) exp cont))
                     ((if) ((G 'eval-if) exp cont))
                     ((set!) ((G 'eval-set!) exp cont))
                     ((lambda) ((G 'eval-lambda) exp cont))
                     ((exec-at-metalevel)
                      ((G 'exec-at-metalevel) exp cont))
                     (else
                      ((G 'eval-list)
                       exp
                       (lambda (l)
                         ((G 'apply)
                          (car l) (cdr l) cont))))))
                  (else (cont exp)))))
    (cons 'eval-quote
          (lambda (exp cont)
            (cont ((G 'S->R) (car (cdr exp))))))
    (cons 'eval-if
          (lambda (exp cont)
            (let ((cond (car (cdr exp)))
                  (then (car (cdr (cdr exp))))
                  (else (car (cdr (cdr (cdr exp))))))
              ((G 'eval) cond
                         (lambda (val)
                           (if val
                               ((G 'eval) then cont)
                               ((G 'eval) else cont)))))))
    (cons 'eval-var
          (lambda (var cont)
```

```
            (let ((pair (assoc var (G 'env))))
              (if pair
                  (cont (cdr pair))
                  (error var "Unbound variable."))))))
    (cons 'eval-set!
          (lambda (exp cont)
            (let ((var (car (cdr exp)))
                  (val (car (cdr (cdr exp)))))
              ((G 'eval)
               val
               (lambda (val)
                 (set-cdr! (assoc var (G 'env)) val)
                 (cont var))))))
;;; apply
    (cons 'apply
          (lambda (fun args cont)
            (cond (((G 'R-procedure?) fun)
                   ((G 'apply-R-procedure) fun args cont))
                  (((G 'R-function?) fun)
                   ((G 'apply-R-function) fun args cont))
                  (else
                   (error fun "It is not a function")))))
    (cons 'eval-list
          (lambda (exp cont)
            (if (null? exp)
                (cont '())
                ((G 'eval) (car exp)
                           (lambda (car-val)
                             ((G 'eval-list)
                              (cdr exp)
                              (lambda (cdr-val)
                                (cont
                                 (cons car-val
                                       cdr-val)))))))))
;;; functions
    (cons 'eval-lambda
          (lambda (exp cont)
```

```
            (cont
             (let ((args (car (cdr exp)))
                   (body (cdr (cdr exp))))
               (list 'R-function args body (G 'env))))))
    (cons 'R-function?
          (lambda (x)
            (and (pair? x) (eq? (car x) 'R-function))))
    (cons 'apply-R-function
          (lambda (fun args cont)
            (let ((vars (car (cdr fun)))
                  (body (car (cdr (cdr fun))))
                  (env (car (cdr (cdr (cdr fun)))))
                  (old-env (G 'env)))
              (setG 'env
                    (append (map cons vars args) env))
              ((G 'eval-list)
               body
               (lambda (val)
                 (setG 'env old-env)
                 (cont
                  ((G 'last-element) val)))))))
    (cons 'last-element
          (lambda (l)
            (cond ((null? l) '())
                  ((null? (cdr l)) (car l))
                  (else ((G 'last-element) (cdr l))))))

;;;-------------------------------------------------
;;;-------------------------------------------------
;;; The following system functions play the same roles
;;; as the intermediate pointers
;;; and realize linguistic symbiosis.
;;; In this implementation, we use
;;; call-with-current-continuation to implement
;;; coroutine facility.

;;; Function calls from baselevel Rscheme functions
```

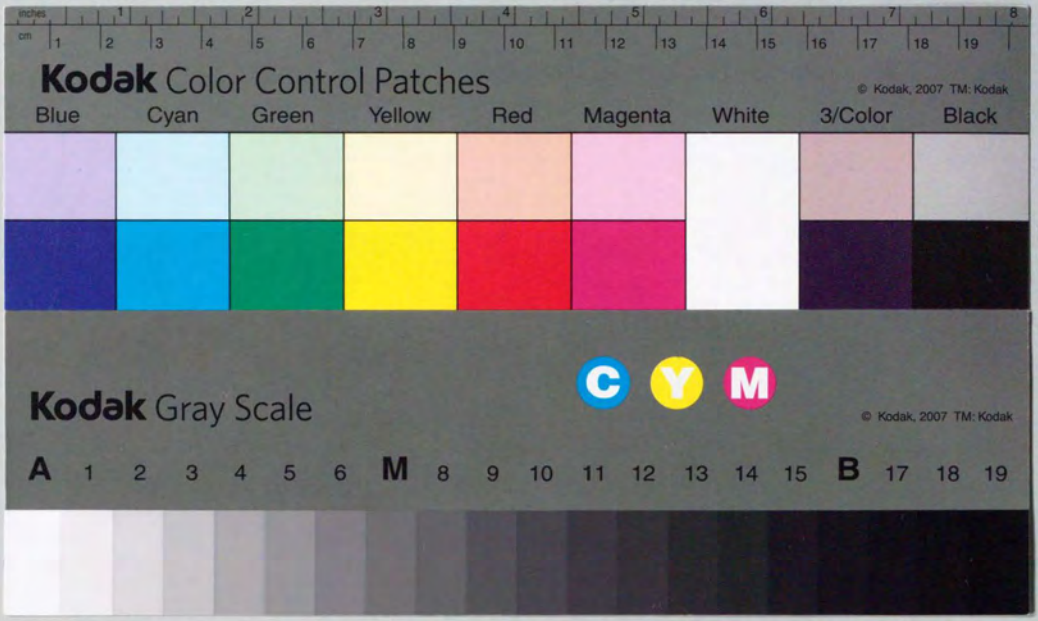
```
;;; to baselevel Scheme procedures.
    (cons 'apply-R-procedure
          (lambda (R-proc R-args cont)
            (let ((proc ((G 'R-procedure->procedure)
                         R-proc))
                  (S-args (map (G 'R->S) R-args))
                  (shift-to-meta (G 'shift-to-meta)))
              ((G 'shift-to-base))
              (let ((val (apply proc S-args)))
                (shift-to-meta)
                (cont ((G 'S->R) val))))))
;;; Convert an Rscheme function to a Scheme procedure.
    (cons 'R-function->procedure
          (lambda (x)
            (let ((shift-to-meta (G 'shift-to-meta)))
              (lambda args
                (call-with-current-continuation
                 (lambda (cont)
                   (shift-to-meta)
                   ((G 'apply-R-function)
                    x
                    (map (G 'S->R) args)
                    (lambda (R-val)
                      (let ((S-val ((G 'R->S) R-val)))
                        ((G 'shift-to-base))
                        (cont S-val)))))))))))
;;;------------------------------------------------
;;; Data representation conversions between Scheme and Rscheme.
    (cons 'S->R
          (lambda (x)
            (cond ((pair? x)
                   ((G 'list->R-list) x))
                  ((procedure? x)
                   ((G 'procedure->R-procedure) x))
                  (else x))))
    (cons 'R->S
          (lambda (x)
```

```
            (cond (((G 'R-list?) x)
                   ((G 'R-list->list) x))
                  (((G 'R-procedure?) x)
                   ((G 'R-procedure->procedure) x))
                  (((G 'R-function?) x)
                   ((G 'R-function->procedure) x))
                  ((pair? x)
                   (error x "Cannot convert R->S"))
                  (else x))))
;;; Scheme procedure
    (cons 'procedure->R-procedure
          (lambda (proc)
            (list 'R-procedure proc)))
    (cons 'R-procedure?
          (lambda (x)
            (and (pair? x) (eq? (car x) 'R-procedure))))
    (cons 'R-procedure->procedure
          (lambda (x)
            (car (cdr x))))
;;;  list
    (cons 'list->R-list
          (lambda (l)
            (cons 'R-list l)))
    (cons 'R-list?
          (lambda (x)
            (and (pair? x) (eq? (car x) 'R-list))))
    (cons 'R-list->list
          (lambda (x)
            (cdr x)))
;;;-----------------------------------------------
;;;-----------------------------------------------
;;; The following system function returns two function closures
;;; which play the same role as the level managers.
    (cons 'generate-level-manager
          (lambda (metalevel-SOT)
            (cons
             ;; shift-to-base
```

```
            (lambda ()
              (set! metalevel-SOT (get-current-SOT))
              (set-current-SOT
               (G 'baselevel-SOT)))
            ;; shift-to-meta
            (lambda ()
              (let ((baselevel-SOT (get-current-SOT)))
                (set-current-SOT metalevel-SOT)
                (setG 'baselevel-SOT
                      baselevel-SOT))))))
;;;-------------------------------------------------
;;; The following system functions generates a metasystem
;;; in the lazy manner.
    (cons 'generate-metasystem
          ;; returns shift-to-meta
          (lambda ()
            (let* ((new-SOT (copy-table
                             (G 'default-SOT)))
                   (pair ((G 'generate-level-manager)
                          new-SOT))
                   (shift-to-base (car pair))
                   (shift-to-meta (cdr pair)))
              (shift-to-meta)
              ;; initialize meta meta level's SOT
              (setG 'shift-to-base shift-to-base)
              (setG 'shift-to-meta shift-to-meta)
              (setG 'env
                    (map (lambda (pair)
                           (cons (car pair)
                                 ((G 'S->R) (cdr pair))))
                         (G 'default-env)))
              (shift-to-base)
              shift-to-meta)))
    (cons 'shift-to-metametalevel
          (lambda ()
            (setG 'shift-to-metametalevel
                  ((G 'generate-metasystem)))
```

```
            ((G 'shift-to-metametalevel))))
;;;-------------------------------------------------
    (cons 'exec-at-metalevel
          (lambda (exp cont)
            (setG 'exp exp)
            (setG 'cont cont)
            ((G 'shift-to-metametalevel))
            ((G 'eval) (car (cdr exp))
                       (lambda (R-val)
                         (let ((S-val ((G 'R->S) R-val)))
                           ((G 'shift-to-base))
                           ((G 'cont) S-val))))))
;;;-------------------------------------------------
    (cons 'read-eval-print-loop
          (lambda ()
            (newline)
            (write '==>)
            ((G 'eval)
             (read)
             (lambda (val)
               (write ((G 'R->S) val))
               ((G 'read-eval-print-loop))))))
;;;-------------------------------------------------
    ;; This is the default environment of plain Rscheme.
    ;; ALL the Scheme functions (including higher order
    ;; functions such as apply or map) can be used
    ;; just as Rscheme functions.
    (cons 'default-env
          (list
           (cons 'cons cons)
           (cons 'car car)
           (cons 'cdr cdr)
           (cons 'list list)
           (cons 'null? null?)
           (cons 'eq? eq?)
           (cons '+ +)
           (cons '- -)
```

```
        (cons '* *)
        (cons '/ /)
        (cons '= =)
        (cons 'map map)
        (cons 'apply apply)
        (cons 'write write)
        (cons 'newline newline)
        (cons 'G G)
        (cons 'setG setG)
        ;; NOTE: The function "eval" is not defined
        ;; at standard scheme.
        (cons 'scheme-eval eval)
        ))
  ))                          ; end of default-SOT
(set-cdr! (car (G 'default-SOT)) (G 'default-SOT))
(setG 'baselevel-SOT '())
(setG 'generate-level-manager
      (cdr (assoc 'generate-level-manager
                  (G 'default-SOT))))
(setG 'generate-metasystem
      (cdr (assoc 'generate-metasystem
                  (G 'default-SOT))))
(setG 'shift-to-metametalevel
      ((G 'generate-metasystem)))
((G 'shift-to-metametalevel))
((G 'read-eval-print-loop)))
```


inches
cm
Kodak Color Control Patches
© Kodak, 2007 TM: Kodak
Blue
Cyan
Green
Yellow
Red
Magenta
White
3/Color
Black
Kodak Gray Scale
C
Y
M
© Kodak, 2007 TM: Kodak
A 1 2 3 4 5 6 M 8 9 10 11 12 13 14 15 B 17 18 19