Architecture Design and Compilation Techniques Using Partial Evaluation in Reflective Concurrent Object-Oriented Languages

by

Hidehiko Masuhara

A Dissertation Submitted to
Department of Information Science
Graduate School of Science
University of Tokyo

In Partial Fulfillment of the Requirements
For the Degree of Doctor of Science

January 1999

Abstract

Parallel and distributed programs often have hardware/problem specific optimizations for improving quality of the program such as efficiency and robustness. Those optimizations, unfortunately, degrade portability and re-usability as they are intertwined with the original algorithm description. *Reflective languages,* which provide the application programmer extensible and abstract implementation of the language, can describe such optimizations as extensions to the language. The separation of optimization descriptions gains portability and re-usability of both application programs and optimizations. However, the interpretive execution model of reflective languages imposes a large amount of performance overhead, which sometimes outweighs benefits of optimizations. Previous reflective languages prohibit some of operations being modified via reflection, so as to reduce the amount of interpretation overhead. The imperfection of this approach is that it still leaves a considerable amount of overhead, and it yields less flexible, unclear reflective architecture.

This dissertation investigates design and compilation framework of *meta-interpreters* and *meta-objects* in an object-oriented concurrent language ABCL/R3. By using partial evaluation to compile reflective programs, ABCL/R3 achieves flexible and lucid reflective architecture and efficient execution at the same time. We design full-fledged meta-interpreters by examining several concurrent programming examples. A newly proposed delegation mechanism enables to define modular and scope controlled extensions to meta-interpreters. We design meta-objects by exploiting the notion of reader/writer methods in a concurrent object-oriented language Schematic, so that they can be effectively partially evaluated. The compilation frameworks of meta-interpreters and meta-objects basically translate concurrent object definitions into a sequential program, then apply partial evaluator for a sequential language, and generates a program in a (non-reflective) concurrent object-oriented language, in which base-level and meta-level objects are collapsed to single level objects. The efficiency of generated programs is demonstrated by several benchmark programs, in which our compiler exhibits performance close to non-reflective languages.

Acknowledgments

First and foremost, my heartfelt thanks to my advisor Akinori Yonezawa for giving me opportunities to study on this field and for suggesting right directions when things weren't going very well. It was also a privilege to study under his supervision.

I am particularly pleased to express my appreciation to Satoshi Matsuoka, who has been the best advisor since I began my academic career. His enthusiasm and support helped me to deepen, articulate, and extend premature ideas.

The Professor Yonezawa's research group provided a supportive, educational, and stimulating environment. Especially, I would like to thank my friend Kenjiro Taura for many pleasant and fruitful discussions, and for his invented languages systems ABCL/f and Schematic, which made it possible to use compilers of concurrent object-oriented languages as the back-ends of our system. I would also like to thank Kenichi Asai for many discussions on partial evaluation and nature of reflection. Takuo Watanabe and Yuuji Ichisugi gave me many discussions and comments on reflective and concurrent object-oriented languages.

I would like to express my gratitude to the members of my thesis committee: Masami Hagiya, Yoshihiko Futamura, Tetsuo Tamai, Kei Hiraki and Kentaro Shimizu for their comments and suggestions.

I would like to thank the users of ABCL/R2 and ABCL/R3 systems including Yoshinori Kishimoto, Yasuaki Honda, Masakazu Hori, Atsushi Igarashi, Adriana Diaz, and Naoyasu Ubayashi. They patiently gave numerous suggestions to improve the systems.

Finally, special thanks to my parents Yoshihiko and Hiroko Masuhara, who encouraged, sustained, and tolerate me through the many years.

Contents

1 Introduction **1**

1.1 Background 2

1.1.1 Requirements to Parallel and Distributed Programming Languages 2

1.1.2 Reflective Concurrent Object-oriented Languages 3

1.1.3 Schematic: a Concurrent Object-Oriented Language 7

1.2 Contributions 11

1.3 Organization of the Dissertation 13

2 Reflective Languages and Implementation Techniques **14**

2.1 Computational Reflection 14

2.2 Issues of Reflective Languages 15

2.3 Implementation Techniques of Reflective Languages 18

2.3.1 Naïve Implementations 18

2.3.2 Narrowing Target of Interpretation 18

2.3.3 Compile-Time Metaobject Protocol 21

2.3.4 The First Futamura Projection 22

2.4 Partial Evaluation, the First Futamura Projection, and Reflective Architectures 22

2.4.1 Brief Introduction to Partial Evaluation 23

2.4.2 The First Futamura Projection 24

2.4.3 Impact of the first Futamura projection on Reflective Architectures 26

2.5 Summary 27

3 Design and Implementation of Meta-interpreters **28**
3.1 Design Issues 28
3.1.1 Examples of Extended Concurrent Programming Constructs . 29
3.1.2 The Meta-Interpreter Design 41
3.1.3 Implementation of Customized Language Constructs 49
3.2 Implementation Issues 58
3.2.1 Overview and Problems 61
3.2.2 Compilation Scheme 66
3.2.3 Performance Evaluation 80
3.3 Summary 84

4 Design and Implementation of Meta-objects **86**
4.1 Design Issues 87
4.1.1 Problems of Existing Meta-object Design 87
4.1.2 A New Meta-object Design 91
4.2 Implementation Issues 97
4.2.1 Optimization Using Partial Evaluation 97
4.2.2 Performance Evaluation 105
4.3 Related Work 110
4.4 Summary 111

5 Discussions **112**
5.1 Dynamic Modification 112
5.1.1 Code Versioning 113
5.1.2 Dynamic Code Generation 114
5.2 Infinite Tower 115
5.3 Coexistence with Compile-time MOP 117
5.4 Preactions 118

6 Conclusion **119**
6.1 Summary of the Dissertation 119
6.2 Future Direction 120

Bibliography **122**

A Example Compilation of Reflective Programs in ABCL/R3 **138**

A.1 Base-level Program 138

A.2 Meta-interpreter for Locality Control 139

A.3 Meta-interpreter for Weighted Termination Detection 140

A.4 Specialized Program 141

B Programs Using Guarded Methods **144**

B.1 Base-level Program 144

B.2 Meta-level Program 144

B.3 Optimized Program 145

B.4 Nonreflective Program 147

List of Figures

1.1 Objects in ABCM/1. 5
1.2 ABCL/R objects. 6
1.3 Binary tree in Schematic. 10
1.4 Binary tree in Schematic (continued). 11

2.1 A Simple interpreter definition. 26

3.1 A naïvely written dot-product method. 31
3.2 "Manual" latency hiding versions of dot-product (A). 33
3.3 "Manual" latency hiding versions of dot-product (B). 34
3.4 Communication diagrams of dot-product. 35
3.5 Latency hiding versions of the dot-product function using annotations. 37
3.6 Latency hiding versions of the dot-product function using annotations. (continued) 38
3.7 Constructs `fork/wait` and `fork` for termination detection. 39
3.8 Description of n-queens problem using `fork` and `fork/wait`. 40
3.9 Explicitly implemented search algorithm with a user-level scheduler. . 42
3.10 Explicityl implemented search algorithm with a user-level scheduler. (continued) 43
3.11 Explicit user-level scheduling system. 43
3.12 Example of meta-interpreter customization using delegation. 48
3.13 Meta-level objects for object replication. 51
3.14 Dot-product function using a replica. 52
3.15 Interpretation of replica-creating annotations. 54
3.16 Behavior of customized interpreters for latency hiding. 55

3.17 Interpreter definition for latency hiding. 56
3.18 Interpreter definition for latency hiding. (continued) 57
3.19 Delegation paths of evaluators for termination detection. 57
3.20 Meta-interpreter implementing termination-detection algorithm using acknowledgment messages. 59
3.21 Meta-interpreter implementing customized scheduling. 60
3.22 Example of customized meta-interpreters in ABCL/R3. 62
3.23 Customized meta-interpreters for monitoring variable references. . . . 62
3.24 A base-level program monitored by the customized interpreter. 63
3.25 A base-level program with manually inserted notifications. 64
3.26 Compilation phases of ABCL/R3. 66
3.27 Translated meta-interpreter functions. 69
3.28 Domains for partial evaluation 69
3.29 Simple partial evaluation rules. 70
3.30 Examples that I/O side-effects are not properly preserved. 73
3.31 Extended partial evaluation rules with preactions. 74
3.32 Expression to be passed onto the partial evaluator. 78
3.33 Residual code yielded by the partial evaluator. 79
3.34 Comparison of overhead of meta-level programming. 83
3.35 Elapsed time for 11-queens problem in C, ABCL/f, and ABCL/R3. . 84

4.1 Definition of an ABCL/R meta-object. 88
4.2 Definition of an ABCL/R meta-object. (contiuned) 89
4.3 State transition diagram of an ABCL/R meta-object. 89
4.4 Our new meta-object design. 92
4.5 Our new meta-object design. (continued) 93
4.6 Overview of our optimization framework. 98
4.7 Example base-level program. 99
4.8 Specialization point function for method `move!` of class `point`. 101
4.9 Result of optimization (the underlined expressions come from the base-level method). 104
4.10 Result of optimization (continued) 105
4.11 "Fake" evaluator for `point`. 107

5.1 Compilation of a reflective tower. 116

List of Tables

3.1 Performance comparison between compiled and interpreted executions. 81

4.1 Performance improvement and residual overheads. 109
4.2 Performance of bounded buffer with guarded methods. 110

Chapter 1

Introduction

The thesis of this dissertation is that new implementation techniques using partial evaluation are beneficial to efficiency and extensibility of concurrent object-oriented reflective languages.

Over the past decade, *reflection* has been recognized as useful for parallel and distributed programming[17, 46, 67, 87, 93, 103, 125, 129] because the application programmer can investigate and modify a language by means of an abstract model of the language. One of the most serious problems of reflection is that the model of a language requires interpretive execution. Some naïve implementations, which run full-fledged interpreters, degrade run-time performance by a factor of 10 to 1000 over the non-reflective languages with optimizing compilers.

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

Partial evaluation technique can be a tool that radically remove the run-time overhead from reflective programs, and, at the same time, it is capable of less restrictive reflective model. This is because partial evaluation can compile programs that is executed under a user-defined interpreter, and generate a specialized and efficient program before execution. However, it is not trivial to compile reflective programs by

using partial evaluation. The meta-level architecture should be reconsidered so that it suits partial evaluation. Partial evaluation technique itself should also be improved so that it can properly handle our target languages—concurrent object-oriented languages. Also, language mechanisms that foster meta-level programmability (e.g., inheritance, delegation, etc.) could be incorporated without degrading the run-time performance, if those mechanisms can be optimized by partial evaluation.

Before presenting our solutions to those issues, we first introduce the background: an overview of existing reflective concurrent object-oriented languages, how useful they are for parallel and distributed applications, and an overview of a (non-reflective) concurrent object-oriented language Schematic as a description language.

1.1 Background

1.1.1 Requirements to Parallel and Distributed Programming Languages

Improvements in computer networks and high-performance parallel computers give more and more opportunities to develop parallel and distributed programs. The development of those programs will face various issues that are not apparent in sequential programming such as efficiency (e.g., load-balancing and prioritized scheduling), robustness (e.g., fault-tolerance and security), and portability over different parallel/distributed execution environments.

Without special support from programming languages, it is difficult to cope with all the issues at once. If we had to improve efficiency or robustness for a specific execution environment, an application program would have intertwined description of the original algorithm and the machine specific improvements.

A number of language mechanisms have been studied to cope with those issues. For example, the distributed language Emerald[52] has an *object-migration* mechanism; one of two tightly communicating objects can be moved to the host where the other object resides, to reduce the overheads of remote communication. Another distributed language Argus[63–65] has a *transaction* mechanism, which easily handles and recovers faulty processes in distributed systems.

If we had the desired mechanisms built-into a language, it would be easier to

develop efficient, portable, modular and robust application programs. However, few of them are available to the actual language implementations. This is because:

- most mechanisms are not generic to all circumstances. A mechanism, which is useful in a certain situation, may not be useful, or even harmful in another situation. For example, the transaction mechanism, which is especially useful to distributed database systems, may not be useful to parallel scientific applications running on parallel computers. Moreover, the run-time system of the language may perform expensive lock operations to support transactions, even for programs that do not need transactions.

- implementation of the mechanisms is burdensome. Since the inside of a language system is not uniform over different implementations, a new mechanism may not be portable among different implementations of the same language. In addition, the internal structure of a language implementation is too monolithic to be extended. For example, the implementation of the object migration mechanisms would be very different depending on whether the underlying system supports global address space.

1.1.2 Reflective Concurrent Object-oriented Languages

1.1.2.1 Advantages of Extensible Languages

A promising approach is to provide a relatively small, extensible language, and then implement useful mechanisms as extensions. Advantages of this approach are as follows. (1) Since a language mechanism that is only useful in a certain situation is not provided by default, there is no performance degradation of programs that do not need such a mechanism. Of course, an application that will be executed in the supposed situation can enjoy the mechanism as if it is built-into the language. (2) Optimizations that are specific to a certain situation can be split from the application program descriptions, by defining those optimizations as extensions to the language. Without the language support, application programs tend to have situation-specific optimizations that are intertwined with the algorithm description of the problem. (3) The implementation of the language requires smaller amount of burden to implementors. Similar advantages can be observed in implementation of sequential languages.

An implementation of Lisp, for example, consists of a small language kernel and a number of macros and functions; i.e., a large part of Lisp's language specification is implemented as extensions to the language kernel.

1.1.2.2 Basic Concepts of Reflective Concurrent Object-oriented Languages

Based on this observation, several studies, including this study, focus on *computation reflection*[71, 72, 112] as a theory of extensible languages. In reflective languages, customizable *meta-objects,* which are the abstract implementation of a language, gives extensibility in a separated and portable manner. The rest of the section gives basic concepts in reflective concurrent object-oriented languages, by taking ABCL/R[125] and ABCL/R2[75, 76, 84][1], which are the predecessors of our proposed language ABCL/R3, as examples.

Non-reflective part of ABCL/R is based on ABCM/1[130, chapter 2], a model of concurrent object-oriented computing. In ABCM/1, an object has own state (*instance variables*), behavior (*methods*), and activity (*thread of control*). In ABCM/1, computation is performed by exchanging messages among objects. When an object receives a message, it (1) changes own state, (2) sends messages to other objects, and (3) creates new objects, by following a method that is specified by the received message. (Figure 1.1)

In ABCL/R, an object in an application program (*a base-level object*) has a *meta-object,* which has all information about the base-level object, and specifies behavior of that object. (Figure 1.2) The instance variables of the meta-object contain following information: a queue of pending messages, names and values of instance variables, a list of methods, and current state of the object (e.g., waiting for a message and executing a method). The methods of the meta-object define how a method invocation request is processed—i.e., policy for mutual exclusion and method selection. Also, an auxiliary object (what we call an *evaluator* object or a *meta-interpreter*) of the meta-object determines how a base-level method is interpreted.

[1] ABCL/R2, which is a successor of ABCL/R, has the similar reflective architecture to ABCL/R, except for the notion of 'object groups.' Since the object groups are beyond the focus of the dissertation, we regard ABCL/R as a representative of those two.

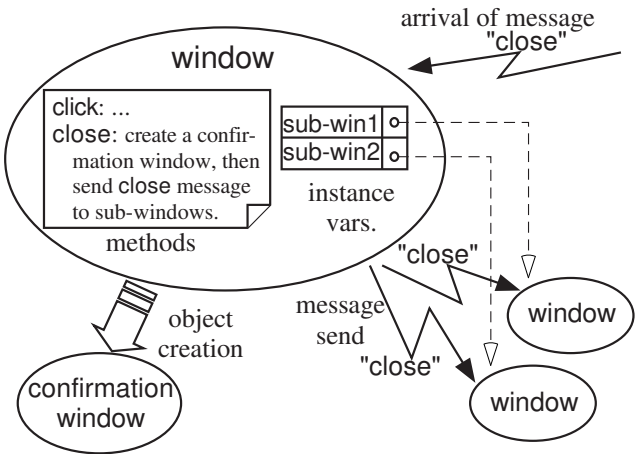


Figure 1.1: Objects in ABCM/1.

A meta-level object is also an object in ABCL/R. The user can customize the language by defining a new meta-object with additional instance variables and customized methods for message acceptance. A meta-object can also be accessed from application-level objects and other meta-objects in order to inspect the state of the base-level objects, and installation/removal of base-level methods.
N.B. In the dissertation, we call any object at the meta-level a *meta-level object.* Especially, a meta-level object that represents a base-level object is called a *meta-object* of the base-level object. A *meta-interpreter* is an object that interprets expressions in base-level programs. In the dissertation, we often distinguish meta-interpreters from meta-objects, unlike other reflective languages.

1.1.2.3 Use of Reflection for Concurrent Applications

The advantages of extensible languages discussed in Section 1.1.2.1 can be realized by customizing meta-level objects in ABCL/R. The examples are as follows:

- In ABCL/R, the messages sent to an object are processed in the arrival order by default. A customized message queue in a meta-object can provide a priority scheduling policy. The A*-algorithm for search problems, for example, can be implemented in this approach.

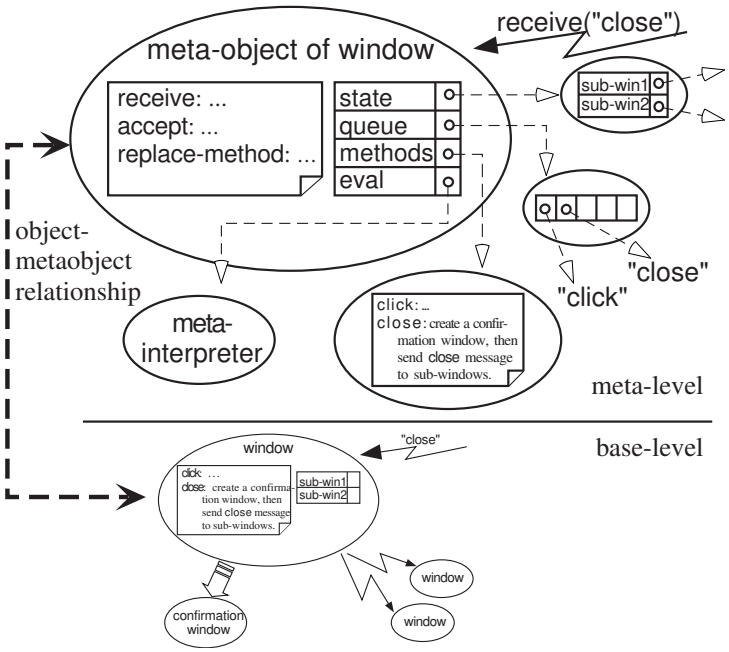


Figure 1.2: ABCL/R objects.

- Fault-tolerant communication can be achieved by customizing meta-objects and meta-interpreters. For method invocation between objects, a customized meta-object at the receiver's side returns an acknowledgment when it receives a method invocation request. A customized meta-interpreter at the caller's side waits for an acknowledgment of the invocation, and retries when no acknowledgment is returned for a certain period. An advantage of this approach is transparency; application-level objects become fault-tolerant by only specifying that those objects use the customized meta-object.

- The simplest way to "migrate" an object to a remote processor is creation of a copy of the object at the remote processor, and forwarding mechanism of method invocation requests from the original object to the copy object. This can be easily implemented by means of ABCL/R's meta-object, since it has internal information about an object, and it implements the behavior upon method invocation.

  The migration mechanism, implemented by the customized meta-object, can be used as if it is built-into the language; i.e., no additional descriptions such as marshaling and unmarshaling code are needed in base-level applications.

Other examples of customized meta-level objects in ABCL/R and ABCL/R2 are used to implement a mechanism to support discrete event simulation[125], a language for soft real-time systems[37], an implementation of synchronization constraints among multiple objects[43], a system for shared resource management[38], a distributed simulation environment[26], and so forth.

1.1.3 Schematic: a Concurrent Object-Oriented Language

In this dissertation, a reflective language ABCL/R3 is proposed. The non-reflective features of the language are based on Schematic[2] [98, 122], while the reflective features are inherited from ABCL/R and ABCL/R2. This section presents basic concepts and syntax of Schematic.

[2] The early version of ABCL/R3 was based on ABCL/f[119–121], which is a yet another concurrent object-oriented language.

Schematic is a parallel and object-oriented extended Scheme. Its key features for concurrent programming are as follows:

- It has parallel constructs *future,*[3] which creates a new thread, and *touch,* which combines synchronization and communication between threads.
- It has a class-based objects. Like many concurrent object-oriented languages, Schematic integrates method invocation and mutual exclusion to easily preserve consistency of mutable data. The mutual exclusion mechanism is similar to the "multiple reader and single writer" model.

Figure 1.3 and 1.4 shows an example program in Schematic.

class and object creation. A class declaration form takes a class name (`btree`), a super-class ("`()`" for no super-class), and instance variables. A function that has the same name to a class (e.g., `btree` in the function `make-empty-node`) creates an object in the class. The arguments to the function specify the initial values of instance variables in the created object. The order of arguments matches to the order of instance variables in the class declaration.

reader method. The third form (`(define-method ...)`) defines a *reader method.* It takes a class name (`btree`), a parameter list consisting of a method name (`lookup`) and formal variables. The first formal variable (`self`) will be bound to the receiver object. The form `(lookup left v)` in a clause of the `cond` form in method `lookup` is a method invocation form, whose method name, receiver object, and argument are `lookup`, `left`, and `v`, respectively. In a reader method, the value of an instance variables of the receiver object can be referenced via the name of the instance variable (e.g., `has-value?`), but not be modified. The referenced values in a reader method are the ones when the method is invoked; even if some instance variables are modified by the other thread during the execution of a reader method, the modification will not be observed within the reader method.

[3]The future invocation is originally proposed in Multilisp[33]. In the context of object-oriented languages, ABCL/*f*is the first language that has the future mechanism.

writer method. The fourth form (`(define-method! ...)`) defines a *writer method,* which uses the same syntax to the reader methods. A writer method can modify the values of instance variables in the receiver object by evaluating the `become` form. For example, the form

```
(become #t :has-value? #t :value v
           :left (make-empty-node)
           :right (make-empty-node))
```

updates instance variables `has-value?` to true, `value` to `v`, and `left` and `right` to newly created empty nodes; and then the expression `#t`, which we call the *result expression,* is evaluated. The invocation of a writer method (except for the result expressions in `become` forms) is mutually excluded from the other writher method invocations on the same object. In this example, two threads cannot execute `insert!` on a node object at the same time. However, when a thread that is executing `insert!` on a node reaches to the result expression `(insert! left v)` in the `cond` form, another thread can start `insert!` on the same node.

future and touch. The fifth form is an example of `future` and `touch`. For a non-empty node, this method creates two threads that compute the sums of values in sub-trees, by evaluating `(future (sum left))` and `(future (sum right))`. The evaluation of the `future` form immediately returns a *reply channel,* instead of waiting for the end of the invocation. Therefore, summation processes of two sub-trees are performed concurrently. A reply channel is a storage where the return value of the `future` invocation will be placed. When the `touch` operation is performed on a reply channel, the stored value is returned whenever the invocation performed by the `future` returned any value; otherwise, it will wait for return value from the invocation. The reply channels can be implicit at the callee's side. If a method is invoked with `future`, a return value of the method is automatically placed in the reply channel by default. In this example, the value of `(+ value (touch left-channel) (touch right-channel))` will be placed an appropriate channel at the caller's side, if the invocation was performed with `future`.

```
;; A class of a node in a binary tree.
(define-class btree ()
  has-value? value left right)

;; A function that creates an empty node.  All instance variables are
;; initialized to false.
(define (make-empty-node)
  (btree #f #f #f #f))

;; A method that tests whether a value is stored in a tree.
(define-method btree (lookup self v)
  (if has-value?                              ; test if it is an empty node.
      (cond ((< v value) (lookup left v))     ; dispatch to its sub-node
            ((> v value) (lookup right v))    ; ibid.
            (else #t)) ; i.e., (= v value)
      #f))                                    ; return false for an empty node.

;; A method that adds a value in a tree. (writer method)
(define-method! btree (insert! self v)
  (if has-value? ; test if it has a value.
      (cond ((< v value) (become (insert! left v)))   ; dispatch to
                                                      ; its sub-node.
            ((> v value) (become (insert! right v))) ; ibid.
            (else (become #f))) ; do nothing when it already has the same value
      ;; store the value if it is an empty node, and create empty sub-nodes.
      (become #t :has-value? #t :value v
                 :left (make-empty-node) :right (make-empty-node))))
```

Figure 1.3: Binary tree in Schematic.

```
;; A method that concurrently sums up values in a tree.
(define-method btree (sum self)
  (if has-value?
      ;; Concurrently invoke the sum method both on the left and right
      ;; sub-nodes.
      (let ((left-channel (future (sum left)))
            (right-channel (future (sum right))))
        ;; Receive returned answers, and sum up values from the
        ;; sub-nodes and this node.
        (+ value (touch left-channel) (touch right-channel)))
      ;; An empty node immediately returns zero.
      0))
```

Figure 1.4: Binary tree in Schematic (continued).

This example shows the simplest usage of channels. A channel in Schematic is the first class object; this property enables flexible usage of channels in concurrent applications[122]. The callee of a method invocation can manipulate the channel that the return value of the current invocation will be placed; it can store the channel into an instance variable in order to reply in future, or can explicitly put a value into the channel so that the caller can continue the process after the touch operation. Also, a caller can specify its own channel as the reply channel of an invocation; this makes it possible to collect return values of multiple future invocations into one channel, and to forward the current reply channel to another invocation. Various usage of channels can be found in the other literature[122].

1.2 Contributions

This dissertation studies on a reflective concurrent object-oriented language ABCL/R3, especially on their reflective architecture design and compilation techniques using partial evaluation. The purposes of the reflective architecture design are to show that

extensibility and programmability of reflective architecture can be enlarged by assuming compilation using partial evaluation, and to present guidelines of reflective architecture that enable successful partial evaluation. The purposes of the compilation techniques are to investigate the frameworks to apply partial evaluation to the meta-level of reflective concurrent object-oriented programs, and to show the effectiveness of the partial evaluation.

The direct contributions of the dissertation include:

- Full-fledged meta-interpreters and their delegation-based extension mechanism, which are proposed in this study, were shown to be useful for implementing several programming constructs and meta-level controls in parallel and distributed applications[79].

- A compilation technique of a base-level method that is executed by meta-interpreters was presented. The technique, which translates the meta-interpreter definition into a sequential program, and then applies a partial evaluator for a sequential language, drastically improves run-time performance by orders of magnitude[74, 78].

- We proposed the preactions mechanism for online partial evaluators, which correctly handles I/O type side-effects. The technique was proved to be useful in compiling reflective programs that intrinsically contains I/O operations[74].

- We investigated the meta-object design of previous reflective concurrent object-oriented languages from the viewpoint of partial evaluation, and showed that the programming style in which meta-objects are defined as state transition machines causes difficulties in applying partial evaluation. We also showed that an approach that splits state-related operations in meta-object definitions makes partial evaluation easier, and presented a new meta-object design by exploiting Schematic's reader/writer methods[80, 81].

- An optimization technique that specializes a meta-object with respect to a base-level object by using partial evaluation was presented. The technique applies partial evaluation in a sequential language by translating meta-object definition.

The effectiveness of the technique was demonstrated by several benchmark applications, where our optimized meta-object runs those applications more than six times as fast as the unoptimized meta-objects[80,81].

1.3 Organization of the Dissertation

The rest of the dissertation is organized as follows: Chapter 2 gives a brief introduction to reflection and reviews implementation techniques of reflective languages. It also introduces partial evaluation.

Chapters 3 and 4 form the core part of the dissertation, which present meta-level design of our language ABCL/R3 and optimization techniques using partial evaluation. Chapter 3 focuses on meta-interpreters, which execute base-level expressions. Chapter 4 focuses on meta-objects which manage base-level objects.

Chapter 5 discusses related topics that are not covered in the previous two chapters. Chapter 6 concludes the dissertation with a list of contributions. The examples of meta-level programming and how they are optimized appear in appendices A and B.

Chapter 2

Reflective Languages and Implementation Techniques

This chapter first introduces the notion of reflection, and then reviews implementation techniques of reflective languages.

2.1 Computational Reflection

Computational reflection is a computational process that is performed on its own computation[72, 112]. A computational system, whose purpose is to solve a certain problem, manipulates data objects, which represent entities in the problem domain, according to some algorithm. A system is called reflective when its problem domain includes its own activity; i.e., how the system solves a problem.

In order to manipulate own activity, a reflective system has its *self-representation* as a data object, and establishes *causal-connection* between the representation and the system itself; i.e., modification of the self-representation is *reflected* on the system's state and behavior, and vice versa.

In many reflective programming languages, the causally-connected self-representation is realized by means of a *tower of interpreters*, in which an application program at level 0 (*base-level*) is executed by the interpreter at level 1 (*meta-level*), the level 1 is executed by the level 2 (*meta-meta-level*), and so forth. Modification of the self-representation is realized by a *level-shifting operation,* which switches current ex-

ecution level to the one level above. This framework automatically preserves casual-connection because programs are actually interpreted by the interpreters. In other words, the execution of a program at the lower level is achieved as indirect effects of manipulation of self-representation by the interpreter at the higher level.

The design of self-representation of a reflective system—called *meta-level architecture*—is crucial to the extensibility of the system. With carefully designed self-representation, the user can extend the system by manipulating abstract, not raw implementation. Such extensions can be:

portable. A self-representation of a reflective language is uniform over different implementations. Thus an extension to the language, which modifies and customizes objects in the self-representation, can also be applied to the other implementation of the same language even if it uses different internal structure for actual implementation. MOP (Metaobject Protocol) for CLOS (Common Lisp Object System)[55] is the successful example as it is available in many commercial and non-commercial CLOS implementations.

modular. In reflective languages, an extension is usually written as a 'difference' to the original self-representation. Moreover, many reflective languages employ modern language mechanisms and programming conventions, such as function closures and class inheritance, so as to make composition of differences easier.

scope-controlled. When a traditional language is extended by modifying its raw implementation, the modification usually affects whole execution of base-level programs. This is undesirable when the modification is intended to affect a specific portion of a base-level program. Some reflective languages support such a requirement by dynamically attaching/detaching modifications to the self-representation according to the portion of the base-level program currently interpreted.

2.2 Issues of Reflective Languages

Though there are number of advantages in reflective languages, there are several problems to be solved in practice:

Infinity: The infinite number of processes in the tower of meta-interpreters must be represented as a finite process.

Generally, a technique called *lazy creation* of meta-levels solves this problem. For example, the implementation of the first reflective language 3-Lisp[25, 112], starts with only the level-1 interpreter. When a level-shifting operation at level-n is requested (which is interpreted by an interpreter at level-$(n+1)$), it spawns a new interpreter at level-$(n+2)$ that executes subsequent operations at level-$(n+1)$.

There are also studies that give formal semantics to the lazy creation technique, where a tower of interpreters is understood as an interpreter with an infinite data structure called *meta-continuation*[23, 124].

This study, for the sake of simplicity, assumes a reflective system that has only the base-level and the meta-level. This simplification does not decrease the flexibility of reflective languages. In fact, most practical examples of reflective languages are implemented only by customizing the meta-level. Further discussion on the infinite tower can be found in Chapter 5.

Performance: The tower of interpreters in reflective languages causes a performance problem. A straightforward implementation poses tremendous amount of overheads. Compilation is not trivial because customized interpreters may execute base-level programs.

Roughly, execution with a straightforwardly implemented interpreter suffers orders of magnitude slowdown from the compiled execution. In fact, our previous study showed that ABCL/R (a reflective concurrent object-oriented language) is more than 100-fold slower than ABCL/1 (a non-reflective language)[75, 76]. Such serious performance overheads, if it were not solved, would easily overwhelm the benefits of reflective languages.

Many implementation techniques are investigated to tackle this problem. The following sections has detailed discussion on those techniques.

Level of abstraction: Appropriate level of abstraction should be chosen as a self-representation of a reflective language, so that the user's desired customization

can be described at the adequate cost. If the level of abstraction is too high, the user cannot describe desired customization. On the other hand, too low level of abstraction complicates the user's customization. Moreover, it could pose additional run-time overheads.

For example, 3-Lisp has expression, environment, and continuation as its self-representation. Under the representation, the user cannot control policies at the machine instruction level such as register allocation. On the other hand, if the self-representation were given at the machine instruction level, such as registers, instructions, and memory, it is inadequate for the customization at the high-level languages. Moreover, the interpretation for each machine-instruction execution would cause tremendous amount of overhead.

Programmability: The meta-level architecture should be designed so that the user can easily describe customizations. Especially, the properties discussed in Section 2.1 (i.e., portability, modularity, and scope-controllability) should be maximized.

The mechanisms exploited in existing reflective languages are as follows: CLOS MOP[53, 55], CodA MOP[86–88] and many other object-oriented reflective languages use the inheritance mechanisms to extend meta-level objects. AL-1/D[92–96] and aspect-oriented programming[100] provide multiple self-representations for a single system. Simmons' first class interpreters[111], MPC++[45–47], and EPP[42] provide ways to compose fragments of meta-level descriptions using function closures, delegation, and mix-ins. Reflection-oriented programming[114] exploits monads.

Unfortunately, those techniques come along with some run-time overheads. For example, an interpreter with a delegation mechanism or an interpreter consisting of a number of sub-objects would be less efficient than a monolithic interpreter because they have to perform dynamic dispatch. Moreover, those mechanisms could prevent the optimizations that can be applied to simpler meta-interpreters.

The last two issues suggest that as long as we are using interpreter based implementation, we cannot achieve fine-level of abstraction and good programmability, and

good performance at the same time.

2.3 Implementation Techniques of Reflective Languages

We categorize implementation techniques of reflective languages into the following four groups. (1) Reflective languages that are implemented without much attention to run-time performance usually have *naïvely implemented meta-interpreters.* (2) Others have meta-interpreters with *narrower target of interpretation* in order to reduce the interpretation overheads. Some of recent reflective languages substantially eliminate the overheads by moving interpretive execution from run-time to compile-time; (3) the *compile-time meta-object protocol* is an approach that explicitly runs the meta-level before execution of base-level programs; and (4) *the first Futamura projection*, which is the central technique in the dissertation, removes the interpretation from an interpreter definition. The last two techniques assume the meta-level and base-level programs are not dynamically modified. The discussion in the following chapters also follow this assumption.

2.3.1 Naïve Implementations

Many of early studies on reflection pursuit finite representation of the tower of meta-interpreters and elaborate meta-level architectures. Implementations in those studies usually have full-fledged interpreters. For example, Lisp-based reflective languages such as 3-Lisp[25, 112], Brown[124], Blond[23], Jefferson's simple reflective interpreter[49] and the Simmons' first class interpreters[111] have meta-circular interpreters that take an expression, environment, and continuation. Even object-oriented reflective languages have full-fledged meta-interpreters such 3-KRS[71], ABCL/R[125], AL-1/D[92–96], and ACT/R[126].

2.3.2 Narrowing Target of Interpretation

To improve the run-time efficiency, it is a natural idea to narrow the target of interpretation. In other words, it allows the user to customize the execution of a part, not whole, of a base-level program via reflection. To do this, the following techniques have

been proposed. All but the last one are used in ABCL/R2[75, 76, 84], a predecessor of ABCL/R3.

Non-reifying object. When an object in a program is executed under the default (i.e., not customized) interpreter throughout its life, the all program fragments for the object can be compiled. In ABCL/R2, we call such an object a *non-reifying object.* The compilation of non-reifying objects, which is initiated by user's declaration, is essentially the same as the one in non-reflective languages, except that the compiled objects should be able to interact with the other reflective objects.

RbCl, which is a reflective concurrent object-oriented language, allows the user to define non-reifying objects in a different language by using the *linguistic symbiosis* technique[40, 41].

Coarser grained interpreter. A full-fledged meta-interpreter is a function that dispatches an expression to a sub-function that corresponds to the given expression type. When we decide some expression types are not reflective, it is possible to compile the corresponding sub-expressions in a program, and to replace sub-functions for those expression types in the meta-interpreter with invocations to the compiled code.

In ABCL/R2, object and concurrency related expressions are reflective; i.e., expressions for variable access, message sending/receiving, and object creation are processed by interpreters. The *partial compilation* technique generates, from a base-level program, hybrid code consisting of expressions and compiled code where callbacks to the meta-interpreter are embedded in the compiled code.

Similar techniques can be found in other reflective languages. Some reflective languages based on impure object-oriented languages[1], such as CLOS MOP[55]

[1] 'Impure object-oriented languages,' as opposed to pure object-oriented languages, have some features that lacks abstraction of objects. For example, the control structures, primitive data types, arithmetic operations, functions are not objects in CLOS and C++. On the other hand, *everything* is object in pure object-oriented languages such as Smalltalk and Self.

and the early version of OpenC++[17, 19], define that only object-related expressions, such as method invocation, are reflective. As a result, the impure part of the languages can be compiled. There is also a study that provides a MOP only for method dispatch[105].

The problem of this approach is that the 'black-boxes' in the meta-interpreters make the meta-level architecture unclear.

Dynamic adjustment. A reflective application may use customized meta-objects to objects/expression-types according to the run-time condition. In such a case, it would be better if a reflective system could adjust the target of interpretation in response to the run-time reflective requests. Such a system first starts the application with compiled—faster—code for all objects/expression-types. When a request to install a customized meta-object for an object or an expression type, the system switches the execution of the selected object or expression-type from the compiled representation to the interpreted—slower—one.

ABCL/R2 gives three meta-object representations to objects: (1) No meta-object: a compiled object is executed. (2) Pseudo-meta-object: a meta-object in which limited functionality (e.g., it can only receive method invocation requests to the base-level object) is provided. (3) Full-fledged meta-object, in which all functions are provided. By default, an object is created with no meta-object, which is the fastest representation. It will be switched to more powerful, but less efficient representation according to the run-time access to its meta-object.

The lazy creation technique for the tower of interpreters is similar to this technique. A meta-interpreter is first implemented with native machine instructions. When a level-shifting operation is requested, it is switched to the expressions and data that are interpreted by another meta-interpreter.

Structural reflection. An extreme way to maximize efficiency is to make all types of expressions *not* reflective. The reflective operations that are allowed in this case are to access meta-level information of the system such as the class of an object, the list of the methods in a class, and how much memory is available. This style of reflection is called *structural reflection*, as opposed to *behavioral*

reflection. For example, Meta-Information-Protocols for C++[11] and Java Core Reflection[48] provide ways to *read* the class name, name and type of methods and instance variables, etc., but no ways to modify the interpretation of a program.

2.3.3 Compile-Time Metaobject Protocol

Reflective languages that have *compile-time MOP* perform the meta-level computation at compile-time. Since they leave no meta-level computation to the run-time, there are no problems of efficiency. The self-representation in those languages is usually an internal structure of a compiler (e.g., parser, code-generator, optimizer, etc.) or a source-to-source translator. Different from the traditional reflective languages (sometimes called *run-time MOP* as opposed to compile-time MOP), run-time state of a system, such as environment in a Lisp interpreter, is not available.

Intrigue[60] is a Scheme compiler that can customize low-level implementations compilation strategies, such as representation of a closure and the code generation routines, by means of meta-level programming. Anibus[103] is a parallelizing compiler whose parallelization strategies can be customized by using the compile-time MOP.

There are studies on compile-time MOP at source-to-source translation level, such as MPC++[45, 46] OpenC++[15, 16], EPP[42], and OpenJava[118]. In those languages, the user can define a function that takes a base-level expression as a parse tree, and returns a transformed expression. This is conceptually similar to Lisp macros, but practically different in that the transformation functions can be associated with base-level classes. This association enables to define class library where an expression *using* a class in the library is transformed by the MOP in the library.

Compile-time MOP can be designed even at binary level. OpenJIT[83] has a self-representation of a 'just-in-time compiler,' which translates virtual machine instructions into native machine instructions.

Aspect Oriented Programming[69, 70, 100] (AOP) can be regarded as compile-time MOP at the source-to-source level. In AOP, there are a base-level program and programs that customize the base-level program from different *aspects.* Those programs are *weaved* at the compile-time.

2.3.4 The First Futamura Projection

The first Futamura projection is another approach to shift meta-level computation to the compile-time. Different from the compile-time MOP, this approach gives a meta-interpreter as a self-representation to the user.

A meta-interpreter is a program which takes an expression from a base-level program, and executes specific operations that correspond to the type of the expression. When we have a meta-level program and a base-level program, the behavior of the base-level program, which is a series of operations executed by the meta-interpreter, can be determined before the execution. Moreover, the behavior can be represented as a program that consists of the operations that will be executed by the interpreter.

A technique to generate such a program is known as *the first Futamura projection*[29], which compiles a program by *partially evaluating* an interpreter definition with respect to an interpreted program. Since this technique is important part of our study, we give detailed discussion in the following section.

The partial evaluation technique itself has been known useful for reflective languages in many years. Danvy[22] and Ruf[106] pointed out relationship between the tower of interpreters in sequential languages and partial evaluation. Chiba and Masuda[18] proposed an optimization technique for the early version of OpenC++, which does the similar transformation to the one a partial evaluator does, but in an ad-hoc manner. Asai, et al.[6, 7] showed an interpreter in a reflective tower can be derived by partially evaluating two interpreters. However, construction of the actual system is not trivial. To the best of the author's knowledge, our language ABCL/R3 is the one of the first languages that uses the first Futamura projection technique to compile reflective programs in object-oriented concurrent languages.

2.4 Partial Evaluation, the First Futamura Projection, and Reflective Architectures

We first present a brief introduction to partial evaluation. Then present the basic idea of compilation, which is called the first Futamura projection. Finally, we discuss the freedom and restrictions on the reflective architecture that are brought by the use of partial evaluation.

2.4.1 Brief Introduction to Partial Evaluation

Partial evaluation is a program transformation technique that automatically specializes a program with respect to a part of program's input[50,51]. Let $p(x, y)$ be a program that takes two parameters. Partial evaluation of p with respect to x is to generate a specialized program of p by assuming that a certain value of the first parameter x. When the x's value is v, p_v is a result of partial evaluation, and often called *residual program.* This partial evaluation process is written as follows:

$$PE(p, v) = p_v.$$

The residual program p_v takes p's remaining parameter y, and has the following two properties: (1) The program p_v is correct. For any w, if $p(v, w)$ terminates[2] and $p(v, w) = z$, then $p_v(w)$ also terminates and $p_v(w) = z$. (2) The program p_v is efficient. The *static expressions* in p, which are p's expressions depending on constant values, the parameter x, and results of other static expressions, are eliminated in p_v.

Suppose there is a program that computes `n`'th `power` of `x` in Scheme:

```
(define (power n x)
  (if (= n 0)
      1
      (* x (power (- n 1) x))))
```

Partial evaluation of the program `power` with respect to 3 (as for the parameter `n`) is taken place in the following steps:

1. The body expression of `power` is symbolically executed under the environment: {`n` $\mapsto$ 3, `x` $\mapsto$ unknown}.

2. Since `n` is a 'known' value in the environment, the sub-expression `(= n 0)` is *static,* and reduced to false. The latter clause `(* x (power (- n 1) x))` is then selected. Its sub-expression, `(- n 1)` is reduced to 2. The entire expression is thus reduced to `(* x (power 2 x))`.

[2] When $p(v, w)$ does not terminate or reports an error, $p_v(w)$'s behavior is not defined.

3. The expression `(power 2 x)` is further expanded to the body expression of `power` with the environment: {`n` $\mapsto$ 2, `x` $\mapsto$ unknown}. It is reduced to `(* x (power 1 x))`, and then `(power 1 x)` to `(* x (power 0 x))`.

4. Finally, the execution terminates by reducing `(power 0 x)` to `1`.

The specialized program becomes like this:

```
(define (power-3 x)
  (* x (* x (* x 1))))
```

Note that static expressions (e.g., `(= n 0)`, `(- n 1)`, and recursive invocation of `power`) are eliminated in the specialized program; thus it is more efficient than the original one.

Whether an expression is static or dynamic is called the *binding-time* of the expression. There are two approaches to determine the binding-time. The one is called an *online* approach[107, 127], which determines during specialization of each expression. The other is called an *offline* approach, which determines the binding-time of every expression in a program by only using the binding-times of input parameters in advance to specialization process. The determination process in the latter approach is called *binding-time analysis* (BTA). The offline approach leads to a simpler and faster partial evaluator, but requires sophisticated BTA.

The important properties of binding-time are: (1) effectiveness of partial evaluation strongly depends on the binding-times of the target program, and (2) the accuracy of binding-times is affected by the *programming-style* of the target program. Especially, side-effects, polymorphism, and complicated data structure often cause problems without careful programming and sophisticated partial evaluation techniques.

2.4.2 The First Futamura Projection

When there are an interpreter definition and a program to be interpreted, partial evaluation of the interpreter with respect to the program returns a compiled program of the interpreted program. We call this application the *first Futamura projection*[29][3].

[3]Futamura proposed two more projections in his study. The second one is construction of a compiler: $PE(PE,L) = compiler$, and the third one is to construction of a compiler generator:

Let $L^{L'}$ be an interpreter of language L written in language L', and p^L be a program p written in L. The interpreter $L^{L'}$ takes a program p^L and input data for p^L, and returns an answer:

$$L^{L'}(p^L, x) = v.$$

A partial evaluator PE is a function that takes $L^{L'}$ and p^L, specializes $L^{L'}$ with respect to p^L, and returns a residual program $L_p^{L'}$, which is the specialized version of $L^{L'}$:

$$PE(L^{L'}, p^L) = L_p^{L'}.$$

The residual program $L_p^{L'}$ is correct to the execution of original interpreter $L^{L'}$:

$$L_p^{L'}(x) = v \qquad \text{if } L^{L'}(p^L, x) = v.$$

The above partial evaluation can be regarded as *compilation* from L to L' because it yields $L_p^{L'}$, which has the same behavior to p^L but is written in L', from the definition of p^L.

The residual program is not only correct but also more efficient than the interpreted execution. Suppose an interpreter shown in Figure 2.1 is partially evaluated with respect to an expression. The static expressions in `eval` are predicates of the conditional branch (e.g., `(const? exp)`) and selectors that extracts sub-expressions from `exp` (e.g., `(operator exp)` in the `application?` branch). The residual program, therefore, will not have those expressions. On the other hand, variable references, conditional branch (e.g., the `if` expression in the `conditional?` branch), primitive applications in the given expression will still appear in the residual program. As a result, the residual program will have the operations that are specified in the expression, but not the one that manipulates the expression itself. Consequently, the program is almost the same expression to the given one.

The first Futamura projection is useful for compiling reflective programs because an interpreter definition, which may be customized by the user's meta-level programming, is mere data for the partial evaluator in the projection. There are several

$PE(PE,PE)$ = *compiler-generator*. This study does not use those two projections, though they are important for accelerating compilation processes. Chapter 5 has the discussion on a technique for run-time specialization.

```
;; evaluate an expression under an environment.
(define (eval exp env)
  (cond ((const? exp) exp)
        ((variable? exp) (lookup exp env))
        ((conditional? exp)
         (if (eval (predicate exp) env)
             (eval (consequence exp) env)
             (eval (alternative exp) env)))
        ((application? exp)
         (let ((rator (eval (operator exp) env))
               (rands (eval-list (operands exp) env)))
           (apply rator rands)))))
```

Figure 2.1: A Simple interpreter definition.

studies that use partial evaluation to compile programs that are executed by customized interpreters[34, 36, 57, 108, 109] for Lisp, pure functional languages, and logic programming languages. Meta-interpreters in those studies, however, are monolithic and strictly separated from the base-level programs, unlike many reflective languages.

2.4.3 Impact of the first Futamura projection on Reflective Architectures

The use of the first Futamura projection brings new pieces of design consideration to meta-level architecture of reflective languages. As is mentioned in Section 2.3.2, many reflective languages have restricted meta-level architecture by narrowing the target of interpretation for the sake of run-time performance. However, use of partial evaluation, which resolves the interpretation overheads, gives larger degree of freedom in meta-level architecture. At the same time, it also poses some restrictions for successful partial evaluation.

As for freedom, full-fledged interpreters become affordable in terms of efficiency, as long as they can be partially evaluated effectively. Thus the meta-level architecture

becomes more clear to the users. Moreover, language mechanisms that support meta-level programming (e.g., dynamic dispatching, delegation, etc.) can be incorporated without incurring additional run-time overheads.

As for restriction, meta-level programming style should be suitable for partial evaluation. For example, mutable data and complicated data structures should be avoided as much as possible because they are sometimes harmful to the quality of residual programs. Moreover, it would be better to provide mechanisms or documentation to ensure that such harmful programming styles can be avoided in the users' customizations.

2.5 Summary

This chapter introduced computational reflection, and important notions of reflective languages such as *causally-connected self-representation* and *tower of meta-interpreters.* The major issues of reflective languages are the infinity in the execution model, run-time overheads of interpretation, and meta-level programmability. Many implementations of reflective languages often sacrifice adequate level of abstraction and programmability to reduce the run-time overheads. This chapter also reviewed implementation techniques of reflective languages, which are classified into four groups: naïve implementation, narrowing the target of interpretation, compile-time MOP, and the first Futamura projection.

The main implementation technique used in the dissertation is the first Futamura projection, which is based on partial evaluation. The use of the first Futamura projection to implement reflective language is promising to allow less restrictive, yet efficiently implemented meta-level architecture.

Chapter 3

Design and Implementation of Meta-interpreters

A *meta-interpreter* in ABCL/R3 is an object that evaluates base-level expressions as a traditional Lisp meta-circular interpreter does. A meta-interpreter, which is one of the meta-level objects representing internals of a base-level object, is a major source of extensibility but inefficiency of reflective languages. This chapter presents design and implementation issues of meta-interpreters. The meta-interpreters in ABCL/R3, which has fine-grained protocols for maximizing flexibility, are first discussed along with several concurrent programming examples. The latter half of the chapter presents an efficient implementation technique of meta-interpreters using partial evaluation and performance evaluation.

3.1 Design Issues

To provide various parallel language constructs, extensible languages based on reflection are attractive for both implementors and users. The first section of the chapter describes our proposed meta-interpreter design of a concurrent object-oriented language ABCL/R3, which has the following characteristics: (1) the full-fledged meta-interpreters offering a clear model for customization, (2) the fine-grained protocols and the delegation mechanism facilitating modular and scope-controlled meta-level programming, and (3) the *reflective annotations* realizing separation and cooperation

of base- and meta-level programs.

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

- Among various reflective capabilities, meta-interpreters are suitable abstraction to customize existing language constructs and to provide novel ones, as many previous studies show[7, 23, 24, 111, 112, 124]. This is also true, or even more important, for concurrent programming, as it often requires various language constructs to adapt special hardware, to implement application specific optimizations, etc.[26, 37, 38, 76, 84, 125, 126]

- Mechanisms that support easier meta-level programming should be provided as long as they do not pose critical performance impact. Specifically, modularity: the ability to compose meta-level programs, and scope-controllability: the ability to limit the effect of a meta-level program to specific part of base-level programs, are primary concerns.

The solutions in this study are:

- full-fledged interpreters with fine grained protocols by assuming partial evaluation

- delegation for modularity. The delegation relationship can be extended pseudo-dynamically manner—useful to match the scope at the base-level.

In the following sections, we show concurrent programming examples that require optimizations, as requirements for syntactic and semantic extensions to the language. We then show proposed meta-interpreter design. Finally, the use of customized meta-interpreters that solves the presented examples are demonstrated.

3.1.1 Examples of Extended Concurrent Programming Constructs

We first examine several concurrent programming constructs which improve programmability and efficiency of parallel and distributed applications. We will see desired a mechanism and syntactic support for each example. Especially, we focus on required mechanisms and desired syntactic support to provide such constructs.

3.1.1.1 Object Replication

Object replication greatly improves performance of programs in a distributed memory environment. When a program frequently accesses a remote object that is not frequently updated, creating a replica of the object within the local processor will substantially reduces the number of remote messages. Such an optimization has been supported several distributed languages; for example, Emerald[9, 52] provides a *call-by-move* mechanism that migrates objects in the parameter list of a method invocation to the receiver's site.

For concreteness, assume there are two vector objects `v1` and `v2` on different processors, and a method `product` (Figure 3.1) is called on the processor on which `v1` is placed. Since `v2` is a remote object, each invocation of `nth-element` on `v2` sends and waits on a remote message; the number of remote messages in this case is twice the length of `v2`.

If replication mechanism were available as a language feature, this program could be optimized by creating a replica of `v2` at the local processor during the computation of `product`. This optimization not only reduces the number of messages transferred, but also the time for waiting for answers to each request—so called *latency*. However, this is not a simple task: for effective execution, we must consider the following aspects: (1) how replicas are created and managed (mechanism), (2) how programmers specify creation of replicas (syntax), and (3) how to decide whether an object should be replicated (policy).

Mechanism: There could be a variety of replication algorithms one could provide as a built-in feature of a language. This is not simple as it may seem, because of interaction with other parts of the language. For example, an original object should be locked if instance variables of its replica could be updated and written back afterwards. Some language systems provide more than one variants for such a mechanism[8], but it would be better if such variants can be programmed by the user.

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

```
;; Definitions for the vector class.
(define-class vector () ...)                        ; Class definition.
(define-method vector (length self) ...)            ; Return the size of the
                                                     ; vector.
(define-method vector (nth-element self index) ; Return the element at the
  ...)                                               ; specified index.

;; Return the dot-product of the vector and v.
(define-method vector (product self v)
  (let ((size (length self))) ;The size of the vector.
    (let loop ((sum 0.0) (i 0))
      (if (= i size)
          sum                    ; Return the sum at the end of the loop.
          (loop (+ sum           ; Add the product of i'th elements to sum.
                   (* (nth-element self i)
                      (nth-element v i)))
                (+ i 1))))))     ; Increment the index.
```

The form `(let` l `((`v_0 e_0`) (`v_1 e_1`) ...)` e`)` in the figure is a shorthand of

```
(letrec ((l (lambda (v0 v1 ...) e))
  (l e0 e1 ...)))
```

in Scheme. It defines a loop with loop parameters v_0 v_1 ... whose initial values are e_0 e_1 .... The invocation of l in e specifies that the loop parameters in the next iteration of the loop have the values of the arguments of the invocation.

Figure 3.1: A naïvely written dot-product method.

Syntax: The usefulness of the extended mechanism is not only determined by what the mechanism does, but also how the mechanism can be used in programs. Therefore, non-intrusive syntax would help portability and clarity of programs, by separating optimization specifications from base-level programs. If the replication mechanism were provided as mere library functions, the programmer had to modify programs for each usage of the replication mechanism.

High Performance Fortran (HPF)'s distribution directives are declarative annotations (comments) which allow the programmers to control the distribution and replications in a non-intrusive manner to base-level programs[35]. The key idea is to provide directives as comments, so that they are non-intrusive. However, the syntax and semantics of HPF directives are fixed and not extensible. Instead, we propose the *reflective annotations* as non-intrusive syntactic extensions at the base-level, and ways of meta-programming to define the associated interpretation of annotations at the meta-level.

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

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

3.1.1.2 Latency Hiding

Latency hiding is an optimization technique to eliminate time to wait for remote messages, where the basic idea is to overlap local computation and remote communication. This is usually realized by modifying programs manually, i.e., by breaking up a single thread of control into multiple threads. The problem is that the modification

```
;;; Latency hiding versions of dot-product.
;;; (A) The element for the next iteration is prefetched.
(define-method vector (product self v)
  (let ((size (length self)))
    (let loop ((sum 0.0) (i 0)
               ;; requests the element for the first iteration.
               (prefetch-channel (future (nth-element v 0))))
      (if (= i size)
          sum
          (let ((next-prefetch-channel ;; requests the element for
                                       ;; the next iteration.
                 (if (< (+ i 1) size)
                     (future (nth-element v (+ i 1))))))
            (loop (+ sum
                     (* (nth-element self i)
                        (touch prefetch-channel))) ;; use of the prefetched value
                  (+ i 1)
                  next-prefetch-channel))))))
```

Figure 3.2: "Manual" latency hiding versions of dot-product (A).

to the program is not small, and varies according to the number of overlapped remote messages. Since the appropriate number of such messages is sensible to the ratio of computation speed and communication latency, the manual approach is not portable.

Figure 3.2 and 3.3 show two versions of the function `product`, which are manually modified for latency hiding from Figure 3.1. These two versions are different in the number of method invocation requests that are sent in advance to the actual use of the data (effectively, *prefetching*). In (A), only a request for the element that is used in the next iteration is sent in advance to its use, while requests for all the elements are sent before the computation in (B). The communication diagrams for the naïve and latency-hiding versions are shown in Figure 3.4.

We provide a mechanism for latency hiding, in which the programmer specifies

```
;;; Latency hiding versions of dot-product.
;;; (B) The whole remote elements are prefetched.
(define-method vector (product self v)
  (let ((size (length self))
        ;; performs future invocation of "nth-element" method on
        ;; the remote vector v, and builds a list of the resulted channels.
        (prefetch-channels
         (let prefetch-loop ((i 0))
           (if (= size i)
               '()
               (cons (future (nth-element v i)) ;; requests the i'th element
                     (prefetch-loop (+ i 1)))))))
    ;; an extra loop variable "channels" are introduced.
    (let loop ((sum 0.0) (i 0) (channels prefetch-channels))
      (if (= i size)
          sum
          (loop (+ sum
                   (* (nth-element self i)
                      (touch (car channels)))) ;; use of the prefetched value
                (+ i 1)
                (cdr channels))))))
```

Figure 3.3: "Manual" latency hiding versions of dot-product (B).

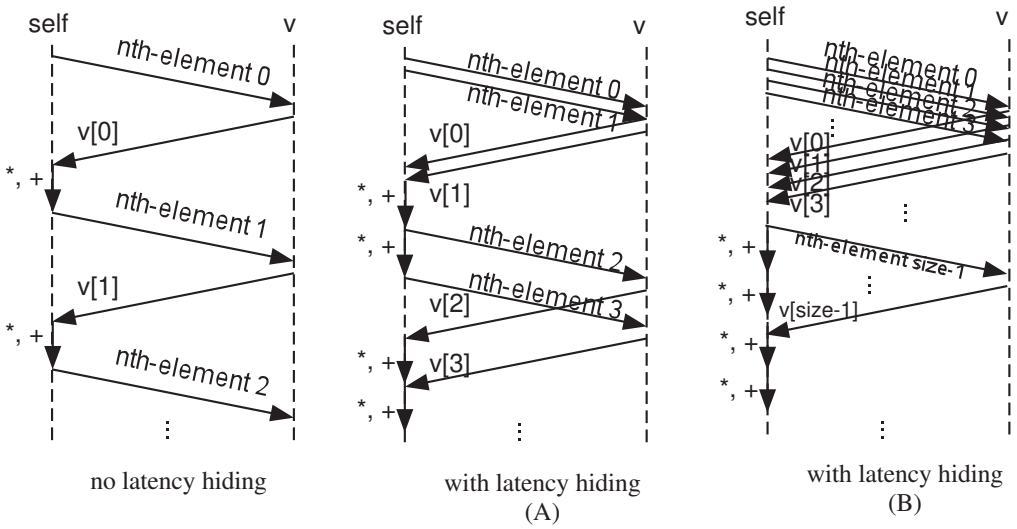


Figure 3.4: Communication diagrams of dot-product.

when and what method invocation should be requested for prefetching in advance to the actual use of the result of the prefetch in the expressions by means of annotations embedded in the original programs.

Assume the annotation for latency hiding has the following form.

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

This annotated expression is interpreted as follows. Before evaluation of the expression e, the expression e_a is evaluated. When e_a is a synchronous method invocation form (i.e., *present* type messages in ABCM[130]), it is executed as an asynchronous (i.e., *future* type) one. When the method invocation form that has the same method name and parameters values to e_a appears during the subsequent evaluation of e, it performs the `touch` operation to the reply box that is generated by the asynchronous invocation in the evaluation of e_a. When e_a is a compound expression such as a loop that performs multiple method invocations, all the subexpressions except for synchronous method invocations are evaluated as usual, and each synchronous invocation is executed as an asynchrounous invocation.

For example, there is an iterative computation that improves some value by using

data in arrays. The first iteration uses an initial value and a local array, and the second uses the result of the first iteration and a remote array. In such a case, fetching from the remote array can be performed during the first iteration. The following expression realizes such a prefetching by using the `prefetch` annotation:

```
(let* ((initial-value ...)
       (first-value (compute
                     initial-value
                     (get-data local-array first-index))))
  (compute first-value
           (get-data remote-array second-index)))
prefetch (get-data remote-array second-index)
```

Using this mechanism, the annotated latency hiding versions of `product` become simpler as is shown in Figure 3.5 and Figure 3.6. Note that without annotations, these two programs are identical to the original one in Figure 3.1.

3.1.1.3 Termination Detection

Some concurrent applications, such as search problems, invoke a large number of threads, where *termination detection* of all the threads is a difficult problem because there is no global control. Several algorithms (*cf.* [85, 104]) have been proposed to solve this problem.

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

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

Special forms `fork` and `fork/wait` are similar to the asynchronous invocation form `future`, except that they detect global termination. The form `fork/wait` in-

```
;;; Latency hiding versions of dot-product.

;;; (A) The element for the next iteration is prefetched.
(define-method vector (product self v)
  (let ((size (length self)))
    (let loop ((sum 0.0) (i 0))
      (if (= i size)
          sum
          (loop (+ sum
                   (* (nth-element self i)
                      (nth-element v i)))
                {prefetch (if (< (+ i 1) size) )(nth-element v (+ i 1))}
                ;; The element for the next iteration is requested before computing
                ;; the product for this iteration.
                (+ i 1))))
    {prefetch (nth-element v 0)} ;; The first remote element is requested before
))                               ;; the iteration begins.
```

Figure 3.5: Latency hiding versions of the dot-product function using annotations.

```
;;; (B) The whole remote elements are prefetched.
(define-method vector (product self v)
  (let ((size (length self)))
    (let loop ((sum 0.0) (i 0))
      (if (= i size)
          sum
          (loop (+ sum
                   (* (nth-element self i)
                      (nth-element v i)))
                (+ i 1))))
    ;; The following annotation is associated to the above loop; i.e., the whole remote
    ;; elements are requested before the beginning of the loop.
    {prefetch (let loop ((i 0))
                (if (< i size)
                    (begin
                      (nth-element v i)
                      (loop (+ i 1)))))}
))
```

Figure 3.6: Latency hiding versions of the dot-product function using annotations. (continued)

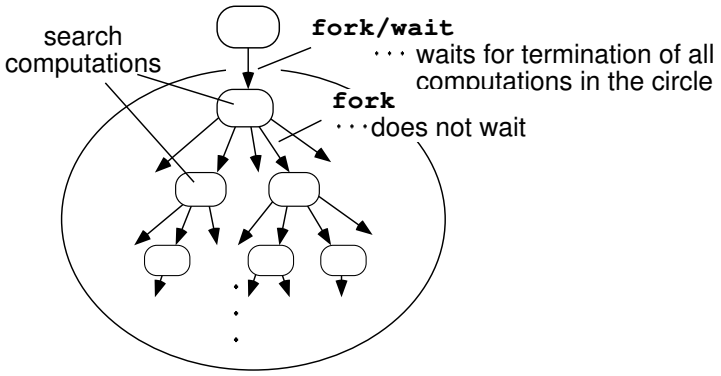


Figure 3.7: Constructs `fork/wait` and `fork` for termination detection.

vokes a specified method or function, and waits for the termination of all subsequent sibling computations invoked with `fork` (Figure 3.7).

For example, Figure 3.8 is a `n-queens` problem using this termination detection support. The annotation at the first line declares that a termination detection algorithm called `weight` will be used. The top-level caller invokes function `n-queens` using the `fork/wait` form. Subsequent recursive `n-queens` invocations are achieved with the `fork` form. The top-level caller waits for the termination of all sibling `n-queens` computations. Note that the definition of `n-queens` is independent of the underlying termination detection algorithms.

3.1.1.4 User-Level Scheduling

Application level information is often useful for controlling scheduling to improve performance. For example, the A*-search is an algorithm to find the best answer in terms of some evaluation function. It uses the estimated value of the answer, which is computed from the intermediate status, as a scheduling priority of a thread. As the branch-and-bound algorithms do, its pruning—terminating subcomputations that have no possibility to reach the best answer—is effective for reducing the search space.

Since such scheduling facilities are not provided in most language systems, programmers are forced to write a program that *explicitly* controls the order of execution.

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

(define-class n-queens () size counter)

;; an entry method for searching
(define-method! n-queens (solve self size)
  ;; set parameters to the instance variables, and then start searching
  (become
   (begin (fork/wait (search self 0 '())) ; invoke and wait for the
                                          ; termination.
          (print "Number of answers are: " (get-count counter)))
   :size size
   :counter (make-counter)))

;; a search process
(define-method n-queens (search self col rows)
  (if (= size col)
      (count-up counter) ; an answer is found
      (let loop ((row 0))
        (if (< row size)
            (begin ;; check for each row in the given column
              (if (not (attacked? size col rows row))
                  ;; invocation without waiting for the termination
                  (fork (n-queens self (+ col 1) (cons row rows))))
              (loop (+ row 1)))))))
```

Figure 3.8: Description of n-queens problem using `fork` and `fork/wait`.

Usually, this is realized with a user-level scheduler object as a server embedded in the base-level application code, and searcher objects as clients, as is shown in Figure 3.9, Figure 3.10 and Figure 3.11. (1) The scheduler activates a searcher object. (2) The object sends requests for object creation and activation, instead of creating its sub-objects, for the next search step. These requests are stored in the queue belonging to the scheduler object. (3) When the activated object finishes its execution, it yields its execution by sending a message to the scheduler. (4) The scheduler then selects a request having the highest priority from the queue, and creates and activates a search object that corresponds to the selected request. The queue of the scheduler is sorted by the priority value of each request, and the scheduler can prune requests from the queue.

One of the problem of this programming style is that the control flow in the original algorithm, which is represented as dashed arrows in the figure, is replaced with more complicated communications, which is represented as solid arrows. As a result, the program becomes unclear and difficult to maintain. Our goal is to provide syntactic support which hides such explicit communications with the scheduler, allowing a programmer to write their search algorithms in a 'natural' style of Figure 3.8.

3.1.2 The Meta-Interpreter Design

We propose the meta-interpreter design of ABCL/R3. The primary concern is programmability. It is designed so that it can easily implement various programming constructs, such as the ones shown before, by assuming partial evaluation based compilation. The major features are as follows:

Full-fledged meta-interpreter, which interprets every expressions in base-level programs, provides clear view of 'behavior' of programs. Such an otherwise sluggish interpreter can be efficiently implemented by our compilation technique.

Fine-grained methods minimize the amount of description of customizations.

Delegation mechanism enable to define modular customizations. This mechanism, different from the traditional inheritance mechanisms, allows to compose customizations dynamically.

```
;; a priority scheduler
(define-class scheduler ()
  searcher (queue '()))
(define-method! scheduler (request self parameters priority-value)
  (become #t :queue (cons (cons priority-value parameters)
                          queue)))

;; When a search node yielded, most promising search node in the
;; scheduling queue is scheduled.
(define-method! scheduler (yield self)
  (let ((sorted-queue (sort queue (lambda (pair) (car pair)))))
    (become
      (search searcher (cdr (car sorted-queue)))
      :sorted-queue (cdr sorted-queue))))

;; a searcher
(define-class a-star () scheduler)

(define-method a-star (begin-search self configuration)
  (become (search self (initial-parameters configuration))
          :scheduler (make-priority-scheduler self)))
```

Figure 3.9: Explicitly implemented search algorithm with a user-level scheduler.

```
;; a process for a search node
(define-method a-star (search self parameters)
  (let ((children-parameters
         (generate-child-nodes parameters)))
    ;; put parameters for the child-nodes into the scheduling queue
    (map (lambda (parameters-for-a-child)
           (let ((estimated-value (estimate parameters-for-a-child)))
             (request scheduler parameters-for-a-child estimated-value)))
         children-parameters)
    (yield scheduler)))
```

Figure 3.10: Explicityl implemented search algorithm with a user-level scheduler. (continued)

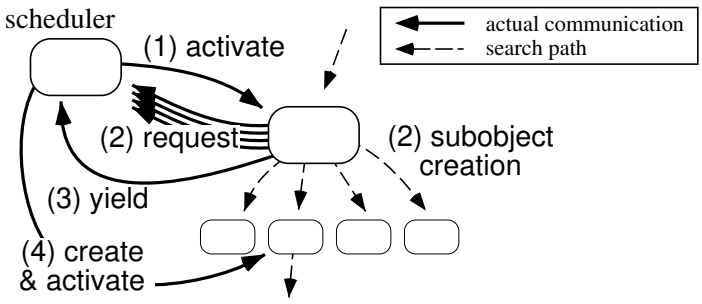


Figure 3.11: Explicit user-level scheduling system.

Reflective annotations serves as a flexible programmable directive to the meta-level from the base-level programs.

3.1.2.1 Fine-grained Methods

A meta-interpreter in ABCL/R3 defines the semantics of base-level programs in a similar manner to the traditional Lisp meta-circular interpreters. A prominent feature is that it defines the semantics by using a number of fine-grained methods, so that user's customizations can reuse many of existing methods. Basically, for each syntactic construct x, a method **eval-**x parses the expression and passes the parsed sub-expressions to a method **do-**x that performs actions. Part of the method definitions in the *primary*—i.e., the default—meta-interpreter are as follows:

(eval-entry self exps env): The method **eval-entry** serves as an entry point of a base-level method/function[1]. The argument **exps** is unevaluated expressions of the method, and **env** is an environment, which binds instance variables of an object, and formal parameters of the method. The default method evaluates subexpressions in **exps** in a sequence. Since the method **do-begin**, which is defined for the **begin** construct, sequentially evaluates expressions, **eval-entry** merely passes **exps** to **do-begin**.

(eval self exp env): The method **eval** serves as a dispatcher—it calls an appropriate sub-method, such as **eval-var**, **eval-if** and **eval-method-call**, according to the expression type. Each sub-method, whose method name is prefixed '**eval-**,' parses the given expression, and then calls another method, which is prefixed '**do-**,' in order to execute desired action.

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

(eval-if self exp env): The method **eval-if** handles a conditional brach. it merely parses (i.e., takes the second, third, and fourth elements, and supplies `#f` for

[1] In ABCL/R3 (and Schematic), base-level programs can use functions (in an ordinary sense) in addition to methods. This means that meta-interpreters receive the expressions in base-level functions, as well as those in methods. We write, however, that the meta-interpreters operate on methods in the following discussion.

an omitted fourth element) the given expression, and then invokes the **do-if** method of **self** with parsed elements and **env**.

(do-if self exp1 exp2 exp3 env): The method **do-if**, like the one in traditional meta-circular interpreters, evaluates **exp1** by invoking method **eval** of **self**, and then conditionally evaluates **exp2** or **exp3**.

(eval-become self exp env): The method **eval-become** is for `become` forms, which update instance variables. It merely parses the sub-expressions and calls **do-become**.

(do-become self body vars exps env): When **exp** is `(become r :v0 e0 :v1 e1 ...)`, **do-become** firstly evaluates `e0`, `e1`, ..., and builds a vector with the evaluated values. The vector is then sent to the meta-object via a *state-update-channel,* which is stored in **env**. (The mechanism to manage instance variables in meta-objects are discussed in the next chapter.) Finally, it continues the evaluation of `r` by invoking **eval** of **self**.

(eval-invocation self exp env): The method **eval-invocation** handles method- and function-invocation forms in base-level programs. First, it examines **exp** to determine (1) the invocation type (e.g., future type or present type), (2) the invocation form (i.e., the method-name, receiver object, and arguments), and (3) optional arguments such as an explicit reply channel, a site where the invocation will be performed, and any other user defined options. Then it calls **do-invocation**.

(do-invocation self type form options env): The method **do-invocation** evaluates the arguments in the invocation form, and then generates a message object that contains the method-name, arguments, reply-channel, and other optional arguments. Finally it invokes the method **method-call** of the meta-object of the receiver object (i.e., target object). If necessary, it waits for an answer to the invocation.

3.1.2.2 Delegation Mechanism

ABCL/R3 has a delegation mechanism to customize meta-interpreters, instead of the inheritance mechanism. The delegation mechanism uses a 'chain' of objects for dispatching. When a method is invoked on a chain of objects, it first searches the method table of the first object in the chain. If the first object does not have the named method, or explicit delegation is performed in the method of the first object, method tables of objects in the subsequent positions in the chain will be examined in turn.

The ABCL/R3's delegation mechanism allows a chain to be extended dynamically. It is beneficial to describe scope-controlled extensions, and to compose those extensions.

Assume that we define a set of meta-interpreters that print trace messages during evaluation of predicate part of each conditional expression. When it evaluates an expression `(g (if (f x) y z))`, for example, it prints the evaluation process of `f` and `x`. An approach to implement such a tracing mechanism might be to define a pair of interpreters; the first interpreter prints trace messages when it evaluates any expression, and the second interpreter, which evaluates all the expressions in a standard manner except for conditionals, creates the first interpreter and let it evaluate predicate part of conditional expressions.

It is difficult to add the tracing mechanism to other interpreters by using static inheritance mechanisms. This is because the static inheritance mechanisms require that the class to be extended should be known. Therefore, in order to add the tracing mechanism to an extended interpreter (an interpreter with the latency hiding mechanisms, for example), we have to define sub-classes from the extended interpreter (a set of interpreters that has the latency hiding mechanisms and the tracing mechanism).

We therefore employ delegation, which enables to dynamically extend behavior of objects, as a mechanims to extend interpreters.

the composed behavior can be naturally implemented by dynamically extending the tracing evaluator before evaluating `(f x)`, no matter which evaluator is used for the enclosing expressions.

Forms and functions for the delegation mechanism are as follows:

(define-delegation-class ⟨class-name⟩ ()
⟨slot⟩ ...)

It declares a new delegation class. It has the essentially same syntax to the ordinary class declarations, except it allows only an empty list to be specified as superclasses.

(define-method ⟨class-name⟩
(⟨method-name⟩ ⟨self-var⟩ ⟨arg⟩ ...)
⟨exp⟩ ...)

It defines a method for the specified delegation class. The variable ⟨self-var⟩ is bound to the front-end in a delegation chain, and a special variable **super** is bound to the 'next' element in the delegation chain. Note that no writer methods (i.e., **define-method!**) can be defined for delegation classes.

(make-empty-chain):

It returns an empty delegation 'chain,' which is a basis for any delegation chain.

(extend-chain ⟨chain⟩ ⟨class⟩ ⟨arguments⟩):

It creates and returns an extended chain of objects based on the given ⟨chain⟩. The first object in the created chain is a newly created object belonging to the specified ⟨class⟩ whose instance variables are initialized by ⟨arguments⟩.

Figure 3.12 is an example that uses the delegation mechanism. The class `noisy-predicate-eval` only defines a method `do-if`, which execute base-level conditional expressions. When invoked, it extends the current chain with `noisy-eval`, and evaluates the predicate (`exp1`) by the extended chain (`ex-eval`). Note that the remaining expressions (`exp2` and `exp3`) are not traced since they are evaluated by `self`.

The class `noisy-eval` is an evaluator that prints an expression for each step. In the method `eval` of `noisy-eval`, the form `(eval super exp env)` invokes the method `eval` of the 'next object in the chain,' which will be determined at run-time.

The notion of delegation is not itself a novel idea, as it is used in Self[113, 123] and other languages. Even though Self allows to dynamically change delegation chains, the run-time system re-compiles methods whenever a chain is changed. Contrastingly, this study uses partial evaluation to compile delegation chains.

```
;;; declaration of a new delegation class called noisy-predicate-eval.
(define-delegation-class noisy-predicate-eval ())

;;; do-if is overridden.
(define-method noisy-predicate-eval (do-if self exp1 exp2 exp3 env)
  ;; It first extends the delegation chain.
  (let ((ex-eval (extend-chain self 'noisy-eval '())))
    (if (eval ex-eval exp1 env) ; It evaluates the predicate under
                                ;    the extended evaluator.
      (eval self exp2 env)      ; It evaluates one of the consequences under
      (eval self exp3 env))))   ;    the original evaluator self.

;;; declaration of another delegation class.
(define-delegation-class noisy-eval ())

(define-method noisy-eval (eval self exp env)
  (display exp) (newline) ; prints the current expression.
  (eval super exp env))   ; delegates the actual evaluation.
```

Figure 3.12: Example of meta-interpreter customization using delegation.

3.1.2.3 Reflective Annotation

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

body{*keyword args...*}

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

(eval-annotation self keyword args body env):

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

Since the method definition can be overridden by user-defined methods, the above interpretation can be changed, as will be shown in Sections 3.1.3.1 and 3.1.3.2.

3.1.3 Implementation of Customized Language Constructs

We have seen several language constructs which can be beneficial for parallel programming. This section shows how these constructs are implemented using meta-interpreters in ABCL/R3. In those implementations, meta-interpreters are used to:

- change the behavior of existing language constructs; e.g., a method invocation is interpreted as a prefetch request or use of the prefetched value in the latency-hiding example.
- non-intrusive annotations; e.g., the annotation for requesting object replication.
- construct new language constructs; e.g., the creation form of a thread with the termination-detection facility.

3.1.3.1 Object Replication

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

First, we define two subclasses of `metaobject`: a class `replicatable` for objects that can create their replicas, and a class `replica-meta` for replicated objects. (Figure 3.13) A method `copy-object` creates a replica on a specified processor (`p`), which is defined as follows:

```
;;; metaobjects of objects that can create replicas
(define-class replicatable (metaobject))

;;; metaobjects of replicated objects
(define-class replica-meta (metaobject)
  original) ; an additional instance variable

;;; creation of a replica of an object
(define-method replicatable (copy-object p &reply-to r)
  (future
    (replica-meta :state-vars state-vars :state-values state-values
                  :methods methods :original self)
    :on p :reply-to r))
```

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

Syntax Here, we show an example syntax that creates replicas. The syntax uses the *reflective annotation* so that base-level programmers can exploit the replication mechanism without modifying the structure of the original programs. The annotation to create replicas is written as follows:

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

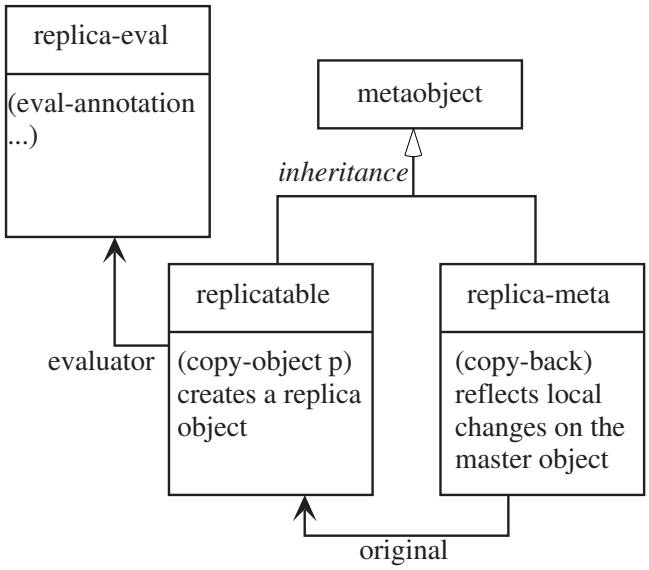


Figure 3.13: Meta-level objects for object replication.

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

We can add annotations to the function `product` shown in Figure 3.1, so as to employ replicas. The program with annotations is shown in Figure 3.14. Note that the only difference between this program and the original one is the addition of the annotation.

As stated in the previous section, the interpretation of annotations can be modified by overriding method `eval-annotation`. Here, we define a class `replica-eval` and a method `eval-annotation`, as is shown in Figure 3.15. In the figure, for an annotated expression `(let ...){replicate (v) :when (< 20 size)}`, `replica-eval` first evaluates `(< 20 size)`. If true, it then looks up variable `v` and creates a copy by invoking the method `copy-object` of the meta-object of `v`. The expression `(let ...)` is evaluated under a newly constructed environment that binds `v` to its replica. Before returning from the method, it invokes the method `copy-back` of the replica,

```
;;; specify the customized meta-object and evaluator
(define-class vector ()
  ...
  :metaobject-creator replicatable
  :evaluator-creator replica-eval)

(define-method vector (product self v)
  (let ((size (length self)))
    (let loop ((sum 0.0) (i 0))
      (if (= i size)
          sum
          (loop (+ sum
                   (* (nth-element self i)
                      (nth-element v i)))
                (+ i 1))))
    {replicate (2) :when (< 20 size)}
    ;; before beginning of the loop, when 20 < size,
    ;; a copy of v is created at the self's site, and
    ;; accesses to v in the loop is redirected to the copy.
))
```

Figure 3.14: Dot-product function using a replica.

so that instance variable in the replica is written back to the original's.

3.1.3.2 Latency Hiding

Here, we show how the latency hiding mechanism described in Section 3.1.1.2 is implemented at the meta-level. The implementation consists of two parts: invoking methods to prefetch the arguments before their usage in an expression, and substitution of results of the prefetch where needed.

Let us review the annotation syntax that requests prefetch method invocations, which is proposed in Section 3.1.1.2:

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

To perform method invocations in e_a, we define two evaluator classes `prefetch-eval` and `prefetch-annotation-eval`, and a method for each—definitions are shown in Figure 3.16, Figure 3.17 and Figure 3.18

Let's see how those evaluators interpret the following expression:

```
(begin (some-computation) (m obj)){prefetch (m obj)}
```

1. Since this is an annotated expression, `eval-annotation` of `prefetch-eval` is invoked. It extends the evaluator chain with `prefetch-annotation-eval`, and let the extended chain evaluate '`(m obj)`' in the argument of the annotation.

2. The method `do-invocation` of `prefetch-annotation-eval` is invoked. It invokes the method in a future type (i.e., asynchronous invocation),

3. and records the 'reply-box' of the future invocation in the environment.

4. Then `prefetch-eval` starts evaluation of the body of the annotated expression, '`(begin ...)`.' At the evaluation of '`(m obj)`,' since its method name and parameters are recorded, `do-invocation` of `prefetch-eval` touches the recorded reply-box, instead of actually invoking the method.

3.1.3.3 Termination Detection

Here, we show that an automatic termination detection mechanism is implemented at the meta-level of ABCL/R3 using two layers of delegating evaluators. The first

```
;;; Evaluator for replica-creating annotations.
(define-delegation-class replica-eval ())

(define-method replica-eval
               (eval-annotation self keyword args body env)
  (cond
   ;; annotations other than replicate
   ((not (eq? keyword 'replicate))
    (eval-annotation super keyword args body env))
   ;; when c yields true
   ((eval self (replica-predicate args) env)
    (let* ((target-vars (replica-target args));; get v1, v2, ...
           (originals (map (lambda (v) (lookup env v)) target-vars))
           ;; create replicas
           (replicas
            (map (lambda (obj)
                   (den-of (copy-object (meta-of obj) (this-pe))))
                 originals))
           ;; create an extended environment
           (ex-env (extend env target-vars replicas)))
      (let ((answer (eval self body ex-env))) ; evaluate the body
        ;; copy-back replicas
        (map (lambda (replica) (copy-back (meta-of replica)))
             replicas))
        answer))                        ; return the answer of the body
    ;; when c yields false
    (else (eval self body env))))
```

Figure 3.15: Interpretation of replica-creating annotations.

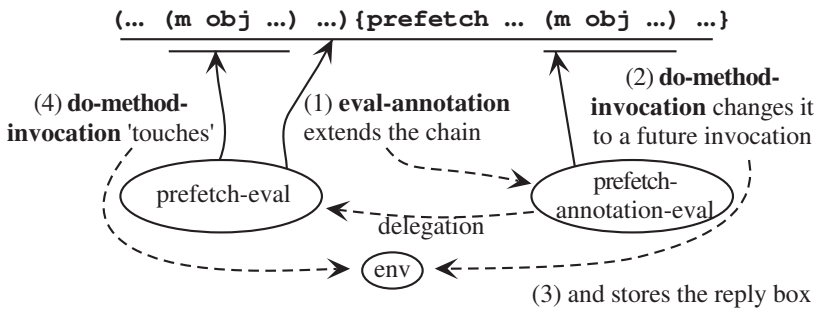


Figure 3.16: Behavior of customized interpreters for latency hiding.

one defines each specific termination detection algorithm, and the second one defines the syntax commonly used in all termination detection algorithms. (Figure 3.19)

At the syntax layer, we define an evaluator class `TD-eval`, which simply dispatches forms `(fork/wait ...)` and `(fork ...)` to the methods `eval-fork/wait` and `eval-fork`, respectively.

At the frontmost layer, an evaluator class is defined for each termination detection algorithm. Figure 3.20 presents the simplest one in which an acknowledgment message is returned for each `fork` invocation. Other algorithms—e.g., the one using global weight[85, 104]—can be implemented in similar ways.

The overview of the algorithm is as follows (operations written in the slanted font are performed at the meta-level):

(1) A method/function is invoked.

(2) *A reply box is created for each child.*

(3) A child (i.e., sub-computation) is forked. *The reply box is passed onto the child along with the invocation.*

(4) *It waits for acknowledgment messages from all of its children.*

(5) *Each child returns an acknowledgment message when it finishes.*

(6) *When all acknowledgment messages are collected, it returns an acknowledgment message to its own parent.*

```
;;; for expressions that may have {prefetch ...}
(define-degelation-class prefetch-eval ())

;;; interpretation of the annotation
(define-method prefetch-eval (eval-annotation self keyword args body env)
  (if (eq? keyword 'prefetch)
    ;; extend the chain withprefetch-annotation-eval
    (let ((eval-a (extend self 'prefetch-annotation-eval '())))
      ;; evaluate expressions in the annotation.
      ;; method invocations in args are regarded as prefetch requests.
      (do-begin eval-a args env)
      ;; evaluate the body expression
      (eval self body env))
    ;; other annotations
    (eval-annotation super keyword args body env)))

;;; method invocation in ordinary expressions
(define-method prefetch-eval (do-invocation self type form options env)
  ;; check whether the method is already invoked in an annotation
  (let ((rbox (prefetched? env type form)))
    (if rbox
      ;; if it is already invoked, do touch
      (touch rbox)
      ;; otherwise, invoke as usual
      (do-invocation super type from env))))
```

Figure 3.17: Interpreter definition for latency hiding.

```
;;; for expressions in annotations
(define-degelation-class prefetch-annotation-eval ())

;;; method invocation in the annotation
(define-method prefetch-anno-eval (do-invocation self type form options env)
  ;; change the invocation type to the 'future'.
  (let ((rbox
         (do-invocation super 'future form options env)))
    ;; remember the returned reply box in the environment.
    (remember-prefetched-method env type form rbox)))
```

Figure 3.18: Interpreter definition for latency hiding. (continued)

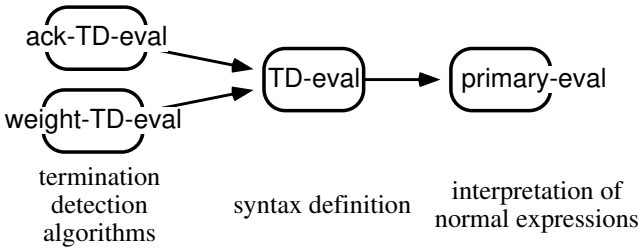


Figure 3.19: Delegation paths of evaluators for termination detection.

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

3.1.3.4 User-Level Scheduling

In Section 3.1.1.4, we have seen an example of user-level scheduling that is achieved by explicitly sending messages in application programs. Here, we show a similar scheduling mechanism that is achieved by *implicitly* communicating with the scheduler object. An evaluator class `schedule-eval` and two methods are defined: (`do-invocation`) instead of invoking a method, a message data is sent to the scheduler object; (`eval-entry`) at the end of each method execution, it notifies the scheduler to yield its thread of control. We assume that the scheduler object is accessible by evaluating `(lookup-meta env 'scheduler)` where `env` is the current environment, and has methods `put!` and `get!`.

3.2 Implementation Issues

In ABCL/R3, base-level programs, whose behavior is defined by full-fledged meta-interpreters, are compiled by using partial evaluation technique, or what we call the first Futamura projection. Although the first Futamura projection has been studied for long years, it is not trivial to apply to ABCL/R3's meta-interpreters due to concurrency, dynamic dispatching, etc.

This section presents our framework that compiles ABCL/R3 programs under customized meta-interpreters. The techniques we have developed are as follows:

- *Pre-processing* that converts object-oriented meta-interpreters, which are using the delegation mechanism, into functions of a sequential language Scheme, so that they become applicable to partial evaluation.
- An extension to partial evaluation called *preactions*, which preserves the order and the number of *I/O side effects*.

```
(define-delegation-class ack-TD-eval ())

(define-method ack-TD-eval (eval-entry self exp env)
  ;; at the beginning of a method: create a queue to record reply boxes
  (let* ((new-env (extend-meta env 'rboxes (make-queue)))
         ;; run the body of the method
         (result (eval-entry super exp new-env)))
    ;; at the end of a method: wait for termination of all children
    (for-each (lambda (rbox) (touch rbox))
      (pop-all! (lookup-meta new-env 'rboxes)))              ; (4)
    ;; notify its parent of the termination
    (reply #t (lookup-meta env 'ack))                        ; (5,6)
    result))

(define-method ack-TD-eval (do-invocation self type form options env)
  (if (eq? type 'fork)
      ;; create a reply box
      (let ((rbox (make-reply-box)))                         ; (2)
        ;; and remember it in the queue
        (push! (lookup-meta env 'rboxes) rbox)
        ;; invoke the method with the created reply box
        (do-invocation super 'future form
                       (cons (cons 'ack rbox) options) env)) ; (3)
      (do-invocation super type form options env))) ; for the other forms
```

Figure 3.20: Meta-interpreter implementing termination-detection algorithm using acknowledgment messages.

```
;;; an evaluator class for user-customized scheduling
(define-delegation-class schedule-eval ())

;;; reference to the scheduler is stored in the environment
(define-method schedule-eval (get-scheduler self env)
  (lookup-meta env 'scheduler))

;;; send a request to the scheduler for method invocation
(define-method schedule-eval (do-invocation self type form options env)
  (if (eq? type 'future)
    (let ((thunk (lambda ()
                   (do-invocation super type form options env))))
      (put! (get-scheduler self env) thunk))
    (do-invocation super type form options env)))

;;; yield the control to the scheduler
(define-method schedule-eval (eval-entry self exp env)
  ;; evaluates the body of the method.
  (eval-entry super exp env)
  ;; at the end of the method, it runs a thunk in the scheduling queue.
  (let ((thunk (get! (get-scheduler self env))))
    (thunk)))
```

Figure 3.21: Meta-interpreter implementing customized scheduling.

- *Post-processing* that further optimizes the partially evaluated programs before passing it on to the back-end compiler.

We have developed a prototype compiler for an object-oriented concurrent reflective language ABCL/R3 according to our compilation framework. Benchmarks show that: (1) interpretation overhead is effectively eliminated, i.e., the programs compiled by our compiler exhibit almost identical performance to the ones compiled by non-reflective compilers, and is more than 100 times faster compared to interpreter execution; and (2) parallel applications on a massively parallel processor Fujitsu AP1000 optimized via meta-level programming adds only small overhead compared to hand-crafted source-level optimizations, and runs *faster* than non-optimized base-level programs compiled by a non-reflective compiler. This facilitates creation of a portable, meta-level class framework for optimization and language extensions.

3.2.1 Overview and Problems

3.2.1.1 A Simple Compilation Example in ABCL/R3

We give an overview of our compilation framework with a simple meta-interpreter example shown in Figures 3.22, 3.23, and 3.24. The base-level program defines a broker, which assigns a job request to one of servers according to the estimated cost of the job. The meta-level embodies a simple tracing system over some variable references. A method `request` (Figure 3.24) of the class `broker` asks each server object the estimated cost and throws the job at the least one. Here, we customize the meta-level interpreter of the base-level algorithm so that when variables `worker2` and `job` are referenced, the reference events are reported to the object `*console*` by messages `notify`.

In ABCL/R3, the default meta-circular interpreter is defined as methods of the primary evaluator object `primary-eval` (the definition is omitted). The above customization is achieved by defining a new evaluator object `watch-eval` to override the method `eval-var` that defines the behavior of variable references (Figure 3.23). The method sends a notification to the object `*console*` if the name of the referenced variable matches `worker2` or `job`. The execution of all the other expressions is sent to the delegate (i.e., `super`), which is a default meta-interpreter `primary-eval` in

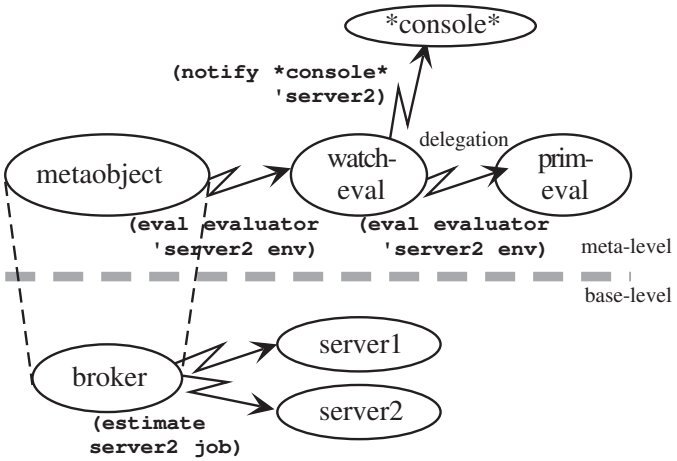


Figure 3.22: Example of customized meta-interpreters in ABCL/R3.

```
;;; customized interpreter to monitor variable references.
(define-delegation-class watch-eval ()
  (*console* (get-console-object)))

(define-method watch-eval (eval-var self var env)
  ;; notifies if the name of the variable matches to the list.
  (if (memq var '(server2 job))
      (notify *console* var))
  ;; requests the delegate to perform variable reference.
  (eval-var super var env))
```

Figure 3.23: Customized meta-interpreters for monitoring variable references.

```
(define-class broker ()
  server1 server2                  ; instance variables for servers.
  :evaluator-creator watch-eval)   ; specify the customized interpreter

(define-method broker (request self job)
  ;; it acquires estimated cost by each server,
  (let ((cost1 (estimate server1 job))
        (cost2 (estimate server2 job)))
    ;; and throws the job at who returned the lower cost
    (start-job (if (< cost1 cost2)
                   server1 server2)
               job)))
```

Figure 3.24: A base-level program monitored by the customized interpreter.

this case.

Since the customization changes the semantics of the language from the original ABCL/R3, a naïve implementation has to execute the compiled base-level program under the customized meta-interpreter, instead of directly executing the base-level program. This execution is more than 100-times slower as we will see in Section 3.2.3.1. The changes of the semantics, however, can also be realized by changes of the base-level program that is executed under the default (i.e., unchanged) semantics. For example, the semantics that reports specific variable references can be realized by a base-level program that is inserted a notification expression for each designated variable references. By all means, having the programmer do so manually throughout the entire program would be quite cumbersome. Figure 3.25 shows such a manually modified program; in fact, our proposed compilation framework automatically generates a very similar program to this.

As mentioned earlier, the principal technique of our compiler is the first Futamura projection[29] that eliminates meta-level interpretation by using partial evaluation[50, 51]. The readers should note that traditional inlining optimization techniques such as the ones in Self[12–14], if applied to the meta-interpreters, would have replicated

```
(define-method broker (request self job)
  (let ((cost1 (estimate server1 (begin (notify *console* 'job) job)))
        (cost2 (estimate (begin (notify *console* 'server2) server2)
                         (begin (notify *console* 'job) job))))
    (start-job (if (< cost1 cost2)
                   server1 (begin (notify *console* 'server2) server2))
               (begin (notify *console* 'job) job))))
```

The underlined expressions are manually inserted for notifying variable references.

Figure 3.25: A base-level program with manually inserted notifications.

almost *the entire* interpreter code, instead of the notification code. Thus, partial evaluation is a quite essential part of the compilation process. However, simply applying traditional partial evaluation techniques is insufficient. Below, we review the basic idea of compiling reflective programs using partial evaluation, and the problems when applied to concurrent objects, as is mentioned in Section 2.4.

3.2.1.2 Problems in existing Partial Evaluation Techniques when Applied to Concurrent Objects

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

Concurrent meta-system. As the previous studies show[76, 94, 125], it is natural to design the meta-system of a concurrent object-oriented language with concurrent objects. However, it is difficult to eliminate the meta-level interpretation by partially evaluating the *entire* meta-system, because of the concurrency and indeterminacy of concurrent objects. To the best of our knowledge, partial evaluation studies that deal with meta-circular interpreters assume functional- or logic-programming languages.

Solution: Although the meta-level architecture of ABCL/R3 consists of a number of concurrent objects, meta-interpreters for each meta-object do not interfere with each other. Therefore, when we focus on meta-interpreters, we can use a partial evaluator for sequential languages. For each meta-object, the system converts associated meta-interpreter definitions into functions, and separately apply partial evaluation to them. Interactions to other objects are regarded as side-effects.

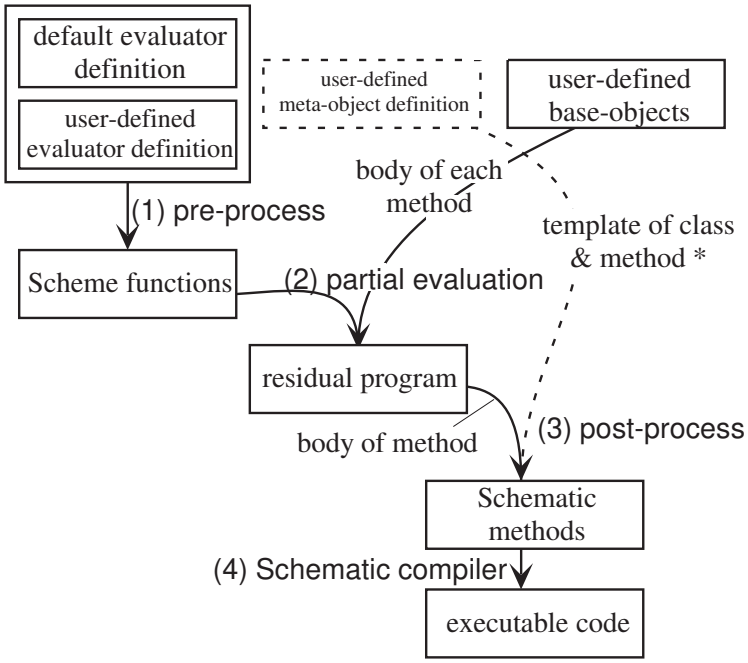
Dynamic dispatching. The delegation mechanism, which dynamically determines an appropriate method for an invocation, may cause a partial evaluator to yield uncompiled results. This is because, for each function/method invocation, if the body of the invoked function/method were not determined at the partial evaluation time, the partial evaluator would not perform further specialization.

Solution: Our system restricts method invocations of delegation chains to be resolved at partial evaluation time. Otherwise, it reports an error[2].

Side-effect in programs. The meta-interpreter definitions may have side-effecting operations, with which simple partial evaluators may incorrectly translate programs. They include: (1) a method invocation at the base-level is represented by a method invocation of a respective meta-object, which is treated as I/O type side-effecting operations; (2) user defined meta-interpreters may assign some values to variables at the meta-level in order to record some information; (3) in many meta-circular interpreters, the assignment operation at the base-level may be realized by the destructive assignment operations (e.g., `set-cdr!`) into environment data structures.

Solution: For (1), we propose a partial evaluation mechanism called *preaction* for preserving the characteristics of I/O operations (including method invocations of objects other than meta-interpreters). With this mechanism, the number and the order of operations in the interaction are preserved after partial evaluation. As for (2), we simply prohibit assignments in meta-interpreters. Finally, for (3), the meta-interpreters implement an assignment in base-level

[2]Errors are reported during the partial evaluation process. With an offline partial evaluator, such errors can be captured by checking *binding time* of each delegation chain.



* Generation of class and method templates from user-defined meta-objects are explained in Chapter 4.

Figure 3.26: Compilation phases of ABCL/R3.

programs by message transmission to a meta-object, so that no explicit assignments appear in the meta-interpreters.

3.2.2 Compilation Scheme

We have developed a compilation scheme of ABCL/R3 methods that are executed by customized meta-interpreters. To implement the above solutions, we divide the compilation into four phases (Figure 3.26) each of which performs the following:

(a) *pre-processing*: the meta-interpreter definitions (including both default and user defined ones) are converted into a set of Scheme functions to be partially evaluated;

(b) *partial evaluation*: the converted functions are specialized with respect to each base-level method, yielding a set of residual expressions;

(c) *post-processing*: the residual expressions are further converted into a Schematic program by adding class/method interface (this process will be explained in Chapter 4); and

(d) *back-end compilation*: the generated program is compiled into an executable code by the Schematic compiler.

Details of each step are as follows.

N.B. As this chapter focuses on the meta-interpreters, the following descriptions assume that no customized meta-*object,* except for customized meta-*interpreters* are used.

3.2.2.1 Pre-Processing: Conversion from Meta-interpreters into Scheme Functions

The pre-processing phase translates class and method definitions of meta-interpreters into Scheme functions. This is not a complicated task because the meta-interpreters are written in essentially functional style.

We represent a delegation chain as a pair of delegation object lists, where the first list is used for dispatching and the second is for 'self.' A delegation object in the lists is a pair of a class name, and values of instance variables:

$$\begin{aligned} \mathit{Chain} &= \mathit{List}(\mathit{Dobj}) \times \mathit{List}(\mathit{Dobj}) && \text{(delegation chain)} \\ \mathit{Dobj} &= \mathit{Class} \times \mathit{List}(\mathit{Val}) && \text{(delegation object)} \end{aligned}$$

where *Class* is a symbol and *Val* means a domain of any data. The functions that manipulate delegation chains are defined as follows:

$$\begin{aligned} \texttt{(make-empty-chain)} &\equiv \langle [\,], [\,] \rangle \\ \texttt{(extend-chain } \langle _, c \rangle \ n \ a\texttt{)} &\equiv \langle c', c' \rangle \ \text{ where } c' = \langle n, a \rangle :: c \\ \texttt{(self-of } \langle _, c \rangle\texttt{)} &\equiv \langle c, c \rangle \\ \texttt{(super-of } \langle h :: t, c \rangle\texttt{)} &\equiv \langle t, c \rangle \\ \texttt{(class-of } \langle \langle n, _ \rangle :: _, _ \rangle\texttt{)} &\equiv n \\ \texttt{(nth-arg-of } \langle \langle _, a \rangle :: _, _ \rangle \ n\texttt{)} &\equiv n\text{'th element of } a \end{aligned}$$

where $\langle x, y \rangle$ constructs a pair of x and y, $h :: t$ creates an list of t by putting h in front of t. Both operations are actually implemented by `cons`. The variable '_' in the left hand side of each definition denotes 'don't care' in pattern matching.

For each method of a delegation class, it generates a function with a unique name. Below is the translation rule:

$\mathcal{P}[\![$`(define-method` n `(`m s a `...)` e `...)`$]\!]$
= `(define (`n`**`m t a `...)`
`(let ((`s `(self-of` t`))`
`(super (super-of` t`))`
`(`v_0 `(nth-arg-of` t `0))`
`(`v_1 `(nth-arg-of` t `1))`
`...`
`)`
e `...))`

where v_0, v_1, ...are the instance variable names of class n.

For each method name m, it also generates a dispatching function. The function takes a same number of arguments to the respective methods, and has conditional branches each of which calls function $n{*}{*}m$, when a method m is defined for class n, and the `class-of` the first argument is n. If the class does not match to any branch, it recursively calls the dispatcher by replacing the first argument with its `super-of`.

Figure 3.27 shows the translated and generated functions for the example program in Figure 3.23.

Though it does not appear in the above example, other ABCL/R3 specific forms, such as `future`, are translated into a form that invokes an *unknown* function.

3.2.2.2 Partially Evaluating Meta-Level Code

Next, the pre-processed meta-level program is specialized with respect to each base-level method using a partial evaluator. We use an online partial evaluator for Scheme[5], which properly operates on programs with I/O type side-effects using our invented *preactions* mechanism.

```
;; a function for method eval-var of class watch-eval
(define (watch-eval**eval-var t0 var env)
  (let ((self (self-of t0))
        (super (super-of t0)))
    (if (memq var '(server2 job))
        (notify *console* var))
    (eval-var super var env)))

;; a dispatching function for eval-var
(define (eval-var self var env)
  (cond ((empty-chain? self) (error 'eval-var "does not understand"))
        ((eq? (class-of self) 'watch-eval)
         (watch-eval**eval-var self var env))
        ((eq? (class-of self) 'primary)
         (primary**eval-var self var env))
        (else (eval-var (super-of self) var env))))
```

Figure 3.27: Translated meta-interpreter functions.

$$
\begin{aligned}
\mathcal{PE} &: Exp \rightarrow (Sym \rightarrow Sval) \rightarrow Sval \\
Sval &= Const(Bval) + Pair(Sval \times Sval) \\
&\quad + Lam(List(Sym), Exp, Env) + Top(Sym, List(Sval)) \\
Bval &= Num + Bool + \cdots \\
Env &= Sym \rightarrow Sval
\end{aligned}
$$

Figure 3.28: Domains for partial evaluation

(rules)

$$
\begin{array}{rcl}
\mathcal{PE}[\![c]\!]\rho & = & const(c) \qquad c \text{ is constant} \\
\mathcal{PE}[\![v]\!]\rho & = & \text{if } v \notin Dom(\rho) \text{ then } top(\texttt{var},[v]) \text{ else } \rho(v) \qquad v \text{ is variable} \\
\mathcal{PE}[\![\texttt{(if } e_0\ e_1\ e_2\texttt{)}]\!]\rho & = & \text{case } r_0 \text{ of} \\
 & & \quad const(true) : \mathcal{PE}[\![e_1]\!]\rho \\
 & & \quad const(false) : \mathcal{PE}[\![e_2]\!]\rho \\
 & & \quad \text{otherwise} \quad : top(\texttt{if},[r_0,\mathcal{PE}[\![e_1]\!]\rho,\mathcal{PE}[\![e_2]\!]\rho]) \\
 & & \text{where } r_0 = \mathcal{PE}[\![e_0]\!]\rho \\
\mathcal{PE}[\![\texttt{(}e_0\ \ldots\ e_m\texttt{)}]\!]\rho & = & \text{case } \langle residualize(r_0,\ldots,r_m), r_0\rangle \text{ of} \\
 & & \quad \langle true,_\rangle \mid \langle false, top(_,_)\rangle \quad : top(\texttt{apply},[r_0,\ldots,r_m]) \\
 & & \quad \langle false, const(o)\rangle \quad : o(r_1,r_2,\ldots,r_m) \\
 & & \quad \langle false, lam([x_1,\ldots,x_m],e,\rho')\rangle \quad : \mathcal{PE}[\![e]\!]\rho'[r_i/x_i]\ (i=1\ldots m) \\
 & & \text{where } r_i = \mathcal{PE}[\![e_i]\!]\rho\ (i=0\ldots m) \\
\multicolumn{3}{l}{\mathcal{PE}[\![\texttt{(lambda (}x_1\ldots x_m\texttt{) }e\texttt{)}]\!]\rho} \\
 & = & lam([x_1,\ldots,x_m],e,\rho)
\end{array}
$$

(initial environment)

$$\rho_0 = [const(plus)/\texttt{+}, const(cons)/\texttt{cons}, const(car)/\texttt{car}, \ldots]$$

(operators)

$$
\begin{array}{rcl}
plus & = & \lambda r_0 r_1.\text{case } \langle r_0, r_1\rangle \text{ of} \\
 & & \quad \langle const(n_0), const(n_1)\rangle : const(n_0+n_1) \\
 & & \quad \text{otherwise} \quad : top(\texttt{+},[r_0,r_1]) \\
cons & = & \lambda r_0 r_1.pair(r_0,r_1) \\
car & = & \lambda r.\text{case } r \text{ of} \\
 & & \quad pair(r_0,r_1) : r_0 \\
 & & \quad \text{otherwise} \quad : top(\texttt{car},[r])
\end{array}
$$

Figure 3.29: Simple partial evaluation rules.

Handling Side-Effects (1): Preactions for I/O type side-effects

The proposed compilation framework depends on a partial evaluation technique. However, simple partial evaluators turn out to be insufficient since they may not correctly handle side-effects. Figure 3.28 and Figure 3.29 shows simplified rules of an online partial evaluator, which uses graph representation[107, 127]. Basically, a partial evaluator PE takes an expression and an environment, and returns a graph structure called *symbolic value* (*Sval*). The rules are similar to those of interpreters of a language, except that PE returns a symbolic value, which represents a program fragment. The first rule means that a constant symbolic value is returned for a constant expression. The rule for a variable returns a 'top' value containing the name of the variable when the variable is free in the current environment, otherwise, it returns a value associated for the variable in the environment. The rule for the `if` form first evaluates the conditional expression e_0. If the result is a constant value, the corresponding sub-expression will be partially evaluated. Otherwise, it returns a symbolic value *top*(`if`, [...]) representing a if-form that dynamically branches to either of sub-expressions. The rule for an application form denotes that (1) it returns either an application form when a termination detection algorithm *residualize* (which is not specified here) tells so, or the value of the operator is unknown; (2) it applies the partially evaluated arguments to the predefined operator; or (3) it further processes the body of the lambda closure. The rule for a lambda form merely creates a symbolic value a lambda-closure with its formal parameter names, expression, and environment. The initial environment has constant symbolic values for primitive operators. The behavior of primitive operators, such as `+`, `cons` and `car`, are defined in the bottom part of the figure. They either returns a symbolic value that represents a dynamic application of the primitive, or the result of the primitive application at partial evaluation time, depending on whether the values of arguments have sufficient information. For example, if two constant symbolic values are applied to the primitive plus (`+`), a constant symbolic value contatining the result of addition is returned. Otherwise, the symbolic value that repsents an addition is returned. Note that the primitive car performs its operation whenever the parameter is known to be a pair symbolic value, regardless the value of its car- and cdr-part.

One of the advantages of the graph representation is its capability to handle

partially static data—a data structure that has both static and dynamic values. For example, the following expression, which generates a cons cell whose car part is static and cdr part is dynamic, is effectively partially evaluated:

$$\begin{aligned}&\mathcal{PE}[\![\texttt{((lambda (p) (cons (* (car p) 2) (* (cdr p) 2)))}\\&\quad\texttt{(cons 3 x))}]\!]\rho_0\\&= \mathit{pair}(\mathit{const}(6), \mathit{top}(\texttt{apply}, [\texttt{*}, \mathit{top}(\texttt{var}, [x]), \mathit{const}(2)]))\\&= \texttt{(cons 6 (* x 2))}\end{aligned}$$

(N.B. The bottom line is a Scheme expression, which is reconstructed from the immediate result of partial evaluation.)

Such a capability is useful to handle environments in meta-interpreters.

Base-level concurrent object-oriented programs involve I/O operations such as message passing, synchronization among objects, etc., that are different from side-effects caused by assignments. (Hereafter, we will refer to these side-effects as the *I/O type side-effects,* as opposed to the side-effects by assignments.) It might seem that such operations could be merely treated as function calls that are executed at run-time (i.e., not subject to unfolding during partial evaluation) by simply extending a partial evaluator for functional languages.

However, such a partial evaluator may move or duplicate operations during its execution, and as a result, I/O operations may be eliminated or duplicated, or may appear in a different order to the original one in the residual program(Figure 3.30).

To solve this problem, we devise a mechanism called *preaction*, which properly preserves the trace of I/O operations in symbolic values of online partial evaluators. Preactions of a symbolic value can be regarded as a history of I/O operations that should be performed before the use of the value. For example, the value of a form:

```
(begin (m obj) 123)
```

is 123, but the action `(m obj)` should be performed before the value is returned. In our partial evaluator, such a value is represented as:

$$\langle\!\langle \mathit{top}(\texttt{apply}, [\mathit{top}(\texttt{var}, [\texttt{m}]), \mathit{top}(\texttt{var}, [\texttt{obj}])])\rangle\!\rangle_{\mathit{const}(123)}$$

The partial evaluation rules extended with preactions is shown in Figure 3.31. An extended symbolic value has a sequence of symbolic values as its preactions. When

1. Disappearance:

$$\mathcal{PE}[\![\texttt{(* 5 (begin (m obj) 2))}]\!] = 10$$

2. Wrong ordering:

$$\mathcal{PE}[\![\texttt{(let ((x (m1 obj1))) (begin (m2 obj2) x))}]\!]$$
$$= \texttt{ (begin (m2 obj2) (m1 obj1))}$$

3. Duplication

$$\mathcal{PE}[\![\texttt{(let ((x (m obj))) (cons x x))}]\!]$$
$$= \texttt{ (cons (m obj) (m obj))}$$

Figure 3.30: Examples that I/O side-effects are not properly preserved.

a compound expression is partially evaluated, the preactions of the expression will be the series of preactions in the partially evaluated result of its sub-expressions. The rule for a constant expression returns a constant symbolic value with an empty precation, because no operations should be performed for generating the constant value. The rule for an if-expression shows a typical use of preactions. First, it evaluates the conditional expression e_0. The result ${}^{A_0}r_0$ means that the value of the expression itself is r_0, and that operations in A_0 are performed to obatin r_0. Therefore, when r_0 is true and the result of partial evaluation of its first branch e_1 is ${}^{A_1}r_1$, then the result of the entire if-expression should be ${}^{A_0::A_1}r_1$, meaning that the value of the expression is r_1, but operations in A_0, and A_1 should be performed to obtain r_1. The rule for an application form generates a symbolic value with a series of preactions, which consist of (1) the preactions in the partially evaluated result of sub-expressions $(A_0 :: \cdots :: A_m)$, (2) the preactions generated during the partial evalaution of the body of a lambda closure (A'), and (3) the symbolic value of the application form itself (r). When the application form is a side-effecting operation, the operation itslef is registered in the preactions of the result of the form.

Some rules place the same operation in the body and preactions, which may

$$
\begin{aligned}
\mathcal{PE}_p &: Exp \to (Sym \to Sval') \to Sval' \\
Sval' &= List(Sval) \times Sval
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{PE}_p[\![c]\!]\rho &= {}^{\langle\langle\rangle\rangle} const(c) \qquad c \text{ is constant} \\
\mathcal{PE}_p[\![\texttt{(if } e_0\ e_1\ e_2\texttt{)}]\!]\rho &= \text{case } r_0 \text{ of} \\
&\qquad {}^{A_0} const(true) : {}^{A_0 :: A_1} r_1 \\
&\qquad {}^{A_0} const(false) : {}^{A_0 :: A_2} r_2 \\
&\qquad \text{otherwise} : {}^{A_0} top(\texttt{if}, [r_0, {}^{A_1} r_1, {}^{A_2} r_2]) \\
&\quad \text{where } {}^{A_i} r_i = \mathcal{PE}_p[\![e_i]\!]\rho
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{PE}_p[\![\texttt{(}e_0 \ldots e_m\texttt{)}]\!]\rho \\
&\quad = \text{let } {}^{A_i} r_i = \mathcal{PE}_p[\![e_i]\!]\rho \ (i = 0 \ldots m), \\
&\qquad {}^{A'} r = \text{case } \langle residualize(r_0, \ldots, r_m), r_0 \rangle \text{ of} \\
&\qquad\qquad \langle true, _ \rangle \mid \langle false, top(_, _) \rangle \ : {}^{\langle\langle\rangle\rangle} top(\texttt{apply}, [r_0, \ldots, r_m]) \\
&\qquad\qquad \langle false, const(o) \rangle \ : {}^{\langle\langle\rangle\rangle} o(r_1, r_2, \ldots, r_m) \\
&\qquad\qquad \langle false, lam([x_1, \ldots], e, \rho') \rangle \ : \mathcal{PE}_p[\![e]\!]\rho'[r_i/x_i] \\
&\qquad\qquad\qquad\qquad (i = 1 \ldots m) \\
&\qquad \text{in } {}^{A_0 :: \cdots :: A_m :: A' :: \langle\langle r \rangle\rangle} r \\
&\mathcal{PE}_p[\![\texttt{(lambda (}x_1 \ldots x_m\texttt{) } e\texttt{)}]\!]\rho \\
&\qquad\qquad = {}^{\langle\langle\rangle\rangle} lam([x_1, \ldots, x_m], e, \rho)
\end{aligned}
$$

${}^A r$ is a symbolic value r with preactions A. $A_0 :: A_1$ concatenates preactions A_0 and A_1.

Figure 3.31: Extended partial evaluation rules with preactions.

cause duplication. Our partial evaluator uses a graph (DAG) structure to represent symbolic values, in order to avoid duplication. When a certain value is used by multiple expressions, it is shared in the graph structure during partial evaluation. At the final phase of the partial evaluation, the shared nodes in the graph are converted to let-forms so that the sharing could be expressed as references to the let-bound variables in the let-body[3].

As an example, consider the following expression is being partially evaluated:

```
(+ 5 (begin (m obj) 2))
```

Since `m` and `obj` are unknown,

$$\mathcal{PE}[\![\texttt{(m obj)}]\!]\rho_0 = top(\texttt{apply}, [top(\texttt{var}, [\texttt{m}]), top(\texttt{var}, [\texttt{obj}])]) \quad (\equiv r_0).$$

The rule for `begin` copies this value as a preaction:

$$\mathcal{PE}[\![\texttt{(begin (m obj) 2)}]\!]\rho_0 = {}^{\langle\langle r_0 \rangle\rangle} const(2).$$

The rule for application copies the preactions of sub-expressions, and also performs computation by using the results of sub-expressions:

$$\mathcal{PE}[\![\texttt{(+ 5 (begin (m obj) 2))}]\!]\rho_0 = {}^{\langle\langle r_0 \rangle\rangle} const(7).$$

This result is further translated into the following expression:

```
(begin (m obj) 7)
```

which properly preserves the I/O operations in the original program.

Handling Side-Effects (2): Instance Variable Assignments

As mentioned, partial evaluators have difficulty in handling assignments to variables and data structures. Although there are studies on partial evaluation that address this problem[4,5,97], they still require heavy global analysis and programming style specialized for partial evaluation.

Fortunately, the programming model of ABCL/R3 (and Schematic) assures that assignment to instance variables can be performed at most once for each method

[3]This conversion is similar to a technique called *lambda-lifting.*

execution, and that the modified values are observed only by method invocations that are processed after the assignment (i.e., even the method that executes the assignment does not observe). As a result, a meta-interpreter can safely represents a base-level assignment operation (`become`) as a message to the meta-object, which is treated as an I/O type side-effecting operation by partial evaluators.

As is presented in Section 3.1.2.1, the default meta-interpreter of ABCL/R3 evaluates a `become` form via the following method:

```
(define primary-eval (do-become self body vars exps env)
  (let ((values (map (lambda (e) (eval self e env)) exps))
        (channel (lookup-meta env 'state-update-channel)))
    (reply (generate-instance-variable-set vars values)
           channel)
    (eval self body env)))
```

The method first evaluates the expressions for the updated instance variables. It then gets a channel to send a set of instance variables to the meta-object by evaluating `lookup-meta`. The updated instance variables are packed by evaluating function `generate-instance-variable-set`, and are sent to the channel (`reply`). After that, it evaluates the result expression of the `become` form (`body`). Obviously, the assignment operation is replaced by the `reply` form, which generates only an I/O type side-effect.

Alternative Approach Though the current version of ABCL/R3 handles assignments by the above scheme, there could be another approach. Below, a technique to handle base-level assignments by converting meta-interpreters into *store-passing style*. This approach would be useful when the base-level language directly allows assignment operations, such as `set!` in Scheme. (In fact, ABCL/f, on which the previous version of ABCL/R3 is based, is such a language; thus the previous version of our compiler uses the following technique.)

The alternative approach is to (1) design the meta-level to interpret base-level assignment operations without using side-effects, and (2) reconstruct assignments *after* the partial evaluation through post-processing.

The meta-interpreter 'shown' to the user for reflective programming is in *direct style,* in which the environment is represented as an association list. During pre-processing, the definition is converted into *store passing style* in addition to CPS (continuation passing style) conversion, so that an assignment operation at the base-level is represented as copying of an environment list at the meta-level. The resulting evaluator functions processed by the partial evaluator takes three arguments: an expression, an environment, and a continuation, which is a function that takes two arguments: a result and an updated environment. The store passing style is a standard technique to describe semantics of imperative language in functional framework.

Assume that standard assignment operations to instance variables (`set!`) are allowed in base-level methods. Since store passing style represents an assignment as a creation of a new 'store' that holds the updated values, the original variable might not be updated at all in the residual program. To resolve this, we insert functions that explicitly update instance variables at the end of method execution, and then reconstruct the actual assignment forms during post-processing. In the compiled program, execution of assignment operations might be delayed until the end of a method, but this is not a problem for ABCL/R3 since the order of assignment operations within a method cannot be observed from other objects. For example, suppose a class `account` has a method `withdraw` defined as follows:

```
;;; class definition
(define-class account () (current 0))
;;; method definition
(define-method account (withdraw amount)
  ;; if the request is too much,
  (if (< current amount)
    0              ; do nothing.
    (begin         ; otherwise, update the account.
      (set! current (- current amount))
      amount)))
```

When the compiler partially evaluates the default meta-interpreter `eval` with the method `withdraw`, the expression shown in Figure 3.32 is passed on to the partial evaluator by the pre-processor.

```
(eval primary-evaluator
      '(if (< current amount) ; expression
         0
         (begin (set! current (- current amount))
                amount))
      (list (cons 'current 0) (cons 'amount 1) ...) ; env.
      (lambda (result env store) ; continuation
        (let ((channel (store-ref store
                         (lookup-meta env 'state-update-channel))))
          (reply (generate-instance-variable-set
                  '(current amount)
                  (lookup-variables '(current amount) env store))
                 channel))
          result)
      (list current amount ...)) ; store
```

(Note that variables *current* and *amount* are regarded as 'unknown' by the partial evaluator.)

Figure 3.32: Expression to be passed onto the partial evaluator.

```
(if (< current amount)
  (begin
    (reply (vector current)  state-update-channel)
    0)
  (begin
    (reply (vector (- current amount))  state-update-channel)
    amount))
```

This result is generated under the assumption that the function `generate-instance-variable-set` eventually creates a vector containing the values of instance variables. Some redundancies, which will be removed in the post-processing phase, are already removed here for the clarity.

Figure 3.33: Residual code yielded by the partial evaluator.

The function `eval` interprets the assignment operation in the expression as copying of the environment value. At the end of the method, the continuation, which generates and sends an updated set of instance variables to the meta-object, is invoked. The code yielded by the partial evaluator is shown in Figure 3.33.

3.2.2.3 Post-processing

Residual programs from the partial evaluator, like the one shown in Section 3.2.2.2, is not itself runnable. Moreover, they may have redundancies that could be harmful to the optimizations of the back-end compiler. Residual programs are converted and optimized into Schematic programs in the post-processing phase, including:

Removing redundancies: Redundancies in the residual code, such as unnecessary let-bindings, unused variable references, nested `begin` forms, etc., are removed. The removal would not be necessary if the backend compiler were powerful enough to optimize those redundancies.

Reconstructing concurrency constructs: Some concurrency constructs, such as `future`, have been translated into function calls and lambda closures. The post-processing will recover those constructs by finding converted function calls in the residual program.

Adding a method interface: The residual code is converted into a method definition of Schematic so that it has the same method interface as the original one. As the method invocation is managed by meta-objects, this conversion is presented in the next chapter.

3.2.3 Performance Evaluation

We executed benchmark programs to evaluate the performance of the proposed compilation framework. The evaluation measures efficiency of programs executed under (default and customized) meta-interpreters. The rest part of a meta-object is assumed to be default one, and executed without interpretation.

The benchmark programs were executed on the early version of ABCL/R3 system, which is built on top of ABCL/f[121]. Therefore, the comparison is made between ABCL/R3, ABCL/f (as a non-reflective concurrent system), and Common Lisp (as a non-reflective sequential system).

3.2.3.1 Basic Performance: Interpretation Overhead

We have performed preliminary benchmarks using a ABCL/R3 compiler based on our framework. The first benchmarks compare the sequential execution speed of the the interpreter and our compiler to illustrate the effectiveness of 'compiling away' the unnecessary interpretation. Sequential benchmark programs (Boyer[30] and n-queens problem) are written in ABCL/R3 without using parallel constructs, nor reflective operations (although side-effects are employed). The programs are executed in three styles: (NR) compiled without the meta-level and directly executed, (INT) executed by a CPS interpreter for ABCL/R3, and (PE) the meta-level is effectively 'compiled away' using our compiler. Programs are executed on a workstation (SUN Sparcstation 10: SuperSparc 50MHz, 128MB memory) with two Common Lisp compilers (Allegro

Table 3.1: Performance comparison between compiled and interpreted executions.

benchmark	elapsed time (sec.)			improvement	residual overheads
applications	PE	INT	NR	INT/PE	INT/NR
Boyer	2.02	2349	2.06	1143	0.99
	1.71	269	1.62	166	1.058
8-queens	0.050	390	0.043	9073	1.16
	0.190	34.6	0.191	182	0.999
10-queens	1.14	9363	1.19	7901	0.965
	4.19	1011	4.45	227	0.940

Top and bottom numbers in each row correspond to the execution on Allegro Common Lisp and CMU Common Lisp, respectively.

Common Lisp 4.1 and CMU Common Lisp 17e) as the back-end compiler[4].

From Table 3.1, we can observe that (1) the proposed compilation scheme exhibits equivalent performance to traditional (i.e., non-reflective) compilers, and (2) compared to naïve interpretation, our compilation scheme improves performance more than 100-fold[5].

[4]For this benchmark, we used Common Lisp compilers, instead of the ABCL/*f* compiler as the back-end compiler for the following reason. The ABCL/*f* compiler used in the benchmarks does not support function closures, which is necessary for execution of the interpreter in (INT). In order to do a *fair* comparison, we judged that we should employ the same back-end compiler. Fortunately the sequential part of ABCL/R3 is almost identical to Common Lisp; thus, we can easily convert sequential ABCL/R3 program into Common Lisp programs by replacing message sends with function calls, for example. Note that this was done for benchmark purposes only; since under normal circumstances the partial evaluator unfolds possible function applications, the residual code, compiled with the ABCL/*f* compiler, does not contain function closures.

[5]The interpreter used in this benchmark is not highly optimized. However, it is worth pointing out that previous studies to optimize/minimize interpreters still result in a factor of 10 times slower execution compared to the non-reflective compilers even with limited 'openness'[18, 76].

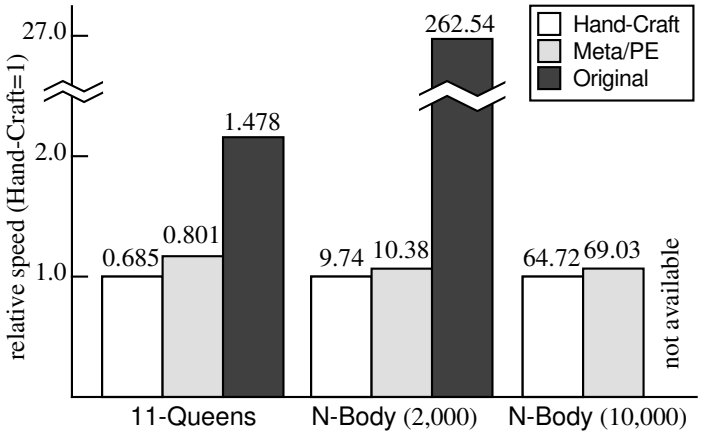
3.2.3.2 Overhead of Meta-Level Programming in Parallel Applications

The next benchmark is to measure the overhead caused by meta-level programming in parallel applications. We compare the executions in three ways. (Original) The original program without meta-level optimizations is directly compiled by the ABCL/f compiler, and executed on Fujitsu AP1000, a massively parallel processors with 64 Sparc-based nodes and very fast torus network interconnection[110]. (Hand-craft) The application is *manually* optimized (see below) and compiled by the ABCL/f compiler. (Meta) The same optimizations are extracted and separately specified as a meta-level class library, and the original program at the base-level is not modified except for a few annotations; these programs are compiled together by our compiler, and executed.

Target application programs are as follows:

Parallel Search: The first base-level application is a simple parallel search program (n-queens problem). Each object is generated as a node in the search tree. Optimizations in Hand-craft and Meta are: (1) *Locality control*—child nodes (objects) at deep levels in the search tree are created at the same processor as their parents' in order to reduce remote communication overhead (the default is to randomly choose a processor). (2) *Weighted termination detection*[104]—'weight' is propagated along the search tree in order to detect the end of a search process as presented in Section 3.1.3.3. By default, the detection is achieved by collecting acknowledgments in the search tree; therefore, intermediate search nodes cannot be released until all its descendant nodes terminate. The meta-level program and its compiled code in the Meta case are given in Appendix A.

N-Body Simulation The second base-level application is a parallel Barnes-Hut N-body simulation algorithm. The optimization technique employed in a hand-tuned ABCL/f code is to cache sub-space data, and exhibits comparable performance to highly optimized algorithm presented[32]. In Hand-craft, method calls that access subspaces in the base-level program are modified to first look-up the cache. In ABCL/R3, this optimization is separately described at the meta-level; a customized meta-interpreter is defined that looks up the cache on specific method calls.



The height of each bar shows elapsed time relative to Hand-craft. Figures on top of each bar are real elapsed time in seconds. The Original execution of N-Body (10,000) failed because of memory exhaustion.

Figure 3.34: Comparison of overhead of meta-level programming.

The graph in Figure 3.34 shows the benchmark results of above two applications: 11-queens problem, and 2,000/10,000 particles N-body simulations. All programs are executed on Fujitsu AP1000 (64 nodes, each has a 25MHz Sparc processor and 16MB memory). From the graph, we can observe that the Meta execution (1) significantly improves the performance of the Original program, and (2) has only small overhead compared to the Hand-craft one, while encapsulating the optimizations into the meta-level. (In the n-queens problem, the overhead was approximately 17%. In the N-body simulation, the overhead in both cases was approximately 7%.) Consequently, we have achieved high efficiency as well as good programmability and re-usability at the same time.

The source of the overhead is mainly that (1) the partial evaluator converts a loop in the base-level program into recursive functions, which is less efficient in ABCL/f, (2) management of 'weights' for termination detection is implemented as separate methods, while they are inlined into the search function in the Hand-craft case, (3) unnecessary assignments of instance variables are performed because of the tech-

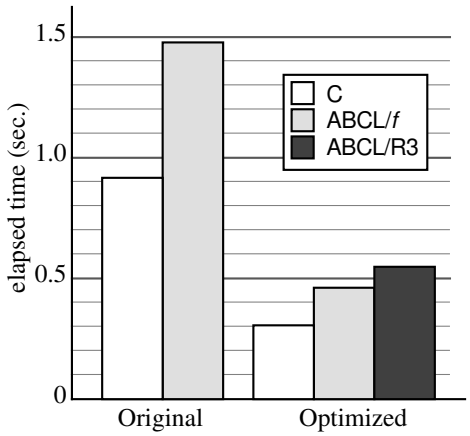


Figure 3.35: Elapsed time for 11-queens problem in C, ABCL/f, and ABCL/R3.

nique described in Section 3.2.2.2. The overhead could be reduced by doing further optimizations such as eager inlining as in Self[12–14], and static flow analysis.

To investigate the baseline efficiency of above programs, we also executed benchmark programs written in C with a message-passing library, against those written in ABCL/f and ABCL/R3 on the AP1000. The left bars in Figure 3.35 indicate elapsed times for the execution of 11-queens problem (Original) written in C and ABCL/f. The right bars are optimized ones in C, ABCL/f, and ABCL/R3. Only the *locality control* technique is employed here; it is achieved by modifying the base-level application (C and ABCL/f), or customizing the meta-level (ABCL/R3). We observe that (1) ABCL/f is only 1.5–1.6 times slower than C, and (2) the optimization effectively improves performance about by 3-fold both in ABCL/f and C.

3.3 Summary

This chapter proposed a highly extensible meta-interpreter design for ABCL/R3, and proposed a compilation framework where ABCL/R3 programs, define under customized meta-interpreters, are translated into efficient code using partial evaluation. They allow the ABCL/R3 users to straightforwardly describe language extensions, which can be used with very low run-time overheads.

The meta-interpreters are designed so as to maximize the extensibility for the syntax and semantics of the base-level languages in an inexpensive manner assuming the compilation technique using partial evaluation. In addition, customization of meta-interpreters can be easily re-used by using standard object-oriented techniques such as delegation in the same way as sequential reflective object-oriented languages (e.g., CLOS) allow. Reflective annotations allow the programmers to write directives to the meta-level programs as comments to a base-level program. The interpretation of annotations can also be customized by modifying the meta-interpreters.

This chapter also presented descriptions of several common concurrent programming strategies. Those include object replication, latency hiding, termination detection, and customized scheduling, and are straightforwardly realized as language constructs by using customized meta-interpreters. Those strategies can be incorporated into existing base-level programs without modifying their structures.

The latter half of the chapter presented a compilation framework based on partial evaluation that almost completely 'compiles away' the overhead of meta-level interpretation. The techniques that make partial evaluation possible include (1) preprocessing that converts object-oriented interpreter definitions to Scheme functions, (2) a new partial evaluation technique called *preactions* that preserves trace of I/O type side-effects, and (3) avoiding assignment-type side-effects in meta-interpreters by prohibiting to have writer methods, which results in a simpler compilation framework.

Benchmarks indicated that (1) meta-interpreters that are partially evaluated with respect to sequential programs in our framework exhibit equivalent performance to the programs compiled by non-reflective compilers, (2) the partially evaluated meta-interpreters are faster than the merely compiled ones by orders of magnitude, (3) execution of customized meta-interpreters that have optimization algorithms for concurrent applications pose only 10–30% overheads, compared to the programs that had been hand-tuned by embedding the optimizations, and compiled by a non-reflective compiler.

Chapter 4

Design and Implementation of Meta-objects

A *meta-object,* in the dissertation, is a meta-level object that defines behavior of the respective base-level object *except for interpretation of expressions in base-level methods,*[1] which is defined by a *meta-interpreter.* More specifically, a meta-object contains the class, list of methods, and list of instance variable names of the base-level object, and defines how to find an appropriate method, mutually exclude multiple invocation requests, and update instance variables.

Similar to meta-interpreters, meta-objects provide a mechanism to customize behavior of base-level objects by means of interpretive execution, which imposes run-time overheads. As is presented in the previous chapter, partial evaluation of meta-programs with respect to base-level programs is a promising technique to eliminate the overheads.

Unfortunately, naïve application of partial evaluators to meta-object definitions does not yield effective result. Although a number of studies on partial evaluation of interpreters have been made[29, 51], meta-objects, as a target of partial evaluation, exhibit difficulties that do not appear in partial evaluation of interpreters.

This is because (1) the design of meta-objects in existing reflective languages is not suitable for partial evaluation, and (2) there are few partial evaluators that can

[1] In a broad sense, a meta-object refers to any object at the meta-level. In the dissertation, however, we distinguish between *meta-objects* in a narrow sense and *meta-level objects.*

deal with concurrent objects. We therefore redesigned meta-objects with consideration to the application of partial evaluation, and here we will show an optimization framework for the resulting meta-objects.

This chapter is organized as follows: The first section discusses meta-object design; it describes why partial evaluation of meta-objects is difficult is described by reviewing an existing meta-object design, and then presents our proposed meta-object design. The second section discusses implementation; the optimization framework of meta-object by using partial evaluation, and our performance evaluation of optimized meta-objects are presented.

4.1 Design Issues

4.1.1 Problems of Existing Meta-object Design

Many concurrent object-oriented languages have mutual exclusion mechanisms to assure consistency. A conservative, commonly found, approach is to mutually exclude all method executions on an object. This approach alleviates the programmers' concern about interference with multiple read/write operations on an instance variable.

The mutual exclusion mechanism in a language drastically affects the meta-object design. This is because (1) the meta-objects explicitly implement the mechanism of base-level objects, and (2) the meta-objects, themselves, are implicitly controlled by a certain mutual exclusion mechanism, which is usually the same one as base-level objects.

In order to meet the above requirements, a meta-object is defined as a *state transition machine* in previous reflective languages. For example, Figure 4.1 and Figure 4.2 are a simplified definition[2] of the default meta-object in the language ABCL/R[125]. Its state transition diagram can be illustrated as in Figure 4.3.

A method invocation on a base-level object is represented by an invocation of the method `receive!`[3] on its meta-object. In `receive!`, the message (an object that contains the method name and arguments) is immediately put into the message

[2]The syntax of the definition is that of in Schematic's[122] for the sake of uniformity.

[3]The exclamation mark in the method name conventionally indicates that the method may change the object's state.

```
;;; Class definition
(define-class metaobj ()
  mode queue state methods evaluator) ; instance variables

;;; Method definition for class metaobj
(define-method! metaobj (receive! self message)
  ;; Here, self is bound to the meta-object itself.
  (put! queue message)
  (if (eq? mode 'dormant)              ; If it is dormant, the received
    (begin (set! mode 'active)         ; message is accepted immediately.
           (future (accept! self)))))

;;; method dispatch
(define-method! metaobj (accept! self)
  (let* ((mes (get! queue))                ; Get a message from the queue.
         (m (find methods mes))            ; method lookup
         ;; creation of an evaluation env.
         (env (make-env self (formals m) mes)))
    (future (eval evaluator (exps m) env self))); evaluation
```

Figure 4.1: Definition of an ABCL/R meta-object.

```
;;; end of method execution
(define-method! metaobj (finish! self)
  (if (empty? queue)              ; Check the queue for pending messages.
      (set! mode 'dormant)        ; If none, turn into the dormant mode.
      (future (accept! self))))   ; Otherwise, accept one of them.

;;; meta-interpreter
(define-method! metaobj (eval self exp env owner)
  ;; It evaluates exp under env.  When finished, it invokes finish! of owner.
  (cond ((constant? exp) (finish! owner exp))
        ((variable? exp) (lookup env exp owner))
        ...))
```

Figure 4.2: Definition of an ABCL/R meta-object. (contiuned)

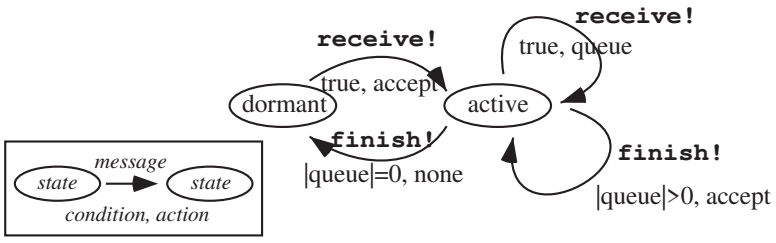


Figure 4.3: State transition diagram of an ABCL/R meta-object.

queue (`queue`), so that it will eventually be processed. If the object is not processing any methods (i.e., `mode` is `'dormant`), the meta-object changes `mode` to `'active` and calls the method `accept!`.

The method `accept!` gets one message from `queue` and lets the evaluator execute the matching method for the message. The evaluator interprets expressions of the method recursively, and when it reaches the end of the base-level method, it invokes the method `finish!` of the meta-object. The method `finish!` examines `queue` for any pending messages received during the evaluation. If `queue` is empty, the meta-object changes `mode` to `'dormant`. Otherwise, it invokes `accept!` again for further execution.

When we apply partial evaluation to this meta-object definition with respect to a certain base-level object, the result is far from satisfactory. The reasons are the following:

- Since the meta-object is defined as a state transition machine, its behavior cannot be determined without static information on some key instance variables such as `mode` and `queue`. In the methods of the meta-object, several conditional expressions by which the control flow branches depend on those variables. A branch with a 'dynamic' condition makes partial evaluation difficult. For example, if the return value of `(get! queue)` in the method `accept!` were "unknown" (dynamic) at the specialization time, method dispatch (`(find methods m)`) and interpretation of the method body (`(eval evaluator exp env self)`) would be left unspecialized. This means that a large amount of interpretive computation cannot be eliminated by merely applying partial evaluation.

  Although modern partial evaluators, such as the ones using the continuation-passing style (CPS), can handle programs with dynamic branches, they duplicate the continuation (i.e., rest of the computation) for both branches in the conditional expression. This means that the code size will explode if the number of dynamic branches in a meta-object increases. In fact, it does in practical reflective languages; the full meta-object in ABCL/R has the *waiting* state in addition to *dormant* and *active*; and the meta-object in AL-1/D has 6 states (*dormant, ready, run, waiting, migrating,* and *stop*)[94].

- Information that should be "known" (static) to the partial evaluator is transferred via instance variables between consecutive method invocations. Such information is not available on the receiver's side unless data structures are analyzed extensively. For example, the value of `(get! queue)` in `accept!`, which would be the value of `message` in `receive!`, is crucial for specialization, but obtaining it requires analysis of `queue`. This requirement sometimes become overwhelming because `queue` might be a user-defined object.

- The key instance variables are mutable; i.e., their values are changed during execution. The execution model of the meta-objects—ABCM/1[130, chapter 2] in this case—however, specifies that method invocations will be processed in FIFO order in each object. We thus have to anticipate that the execution of two consecutive methods may be interleaved. Conservatively, we have to assume that values of mutable instance variables become unknown after method invocations. For example, assume that the method `receive!` invokes the method `accept!`. The variable `queue` at the beginning of `accept!` may have a value different from the one in `receive!` because other methods can be executed before the execution of `accept!`. Though there are partial evaluators that can deal with mutable variables, they regard a mutable variable as unknown (dynamic) unless they can statically determine all update operations to the variable[4,5].

The last two problems could be solved by the partial evaluation technique called 'the trick'[51, Section 4.8.3], which specializes the program by manually giving a set of possible return values of `(get! queue)` beforehand. However, it would be better if we could solve the problem without using such an awkward and ad-hoc technique.

For the above reasons, a partial evaluator conservatively regards most variables as 'dynamic.' Without much of 'static' information, the partial evaluator yields a program that still performs almost all the computation as the program for the original meta-object does.

4.1.2 A New Meta-object Design

We propose, for a reflective concurrent object-oriented language ABCL/R3, a meta-object design that can be effectively optimized by partial evaluation. The key idea is

```
;;; Class definition
(define-class metaobj ()
  lock state-variables state-values methods evaluator)

;;; Reception of a message
(define-method metaobj (receive self message)
  (if (writer? (selector message))          ; check message type
      (accept-W self message)               ; for a writer method
      (accept self message 'dummy)))        ; for a reader method

;;; Processing for a writer method
(define-method metaobj (accept-W self message)
  (let ((c (make-channel)))                 ; channel for receiving
                                            ; updated state
    (acquire! lock)                         ; mutual exclusion begins
    (let ((result (accept self messages c)))
      (cell-set! state-values (touch c))    ; update instance variables
      (release! lock)                       ; end of mutual exclusion
      result)))
```

Figure 4.4: Our new meta-object design.

to separate state-related operations from the other operations using the reader and writer methods of Schematic, which is explained in Section 1.1.3.

4.1.2.1 Overview

The outline of a new meta-object design solving the problems discussed in Section 4.1.1 is shown in Figure 4.4 and Figure 4.5, in which we exploit the reader/writer methods of Schematic. Our design has the following characteristics:

- The behavior of the meta-object is principally defined in the reader methods. Operations that deal with mutable data are defined separately as writer methods or as method invocations on external objects. For example, values

```
;;; Method lookup and invocation
(define-method metaobj (accept self message update-channel)
  (let* ((m (find methods message))     ; method lookup
         (env (make-env self (formals m) message)))
    (future (eval-entry evaluator (method-body m) env update-channel))))

;;; Creation of an evaluation environment
(define-method metaobj (make-env self formals message)
  (extend-env
    (make-env-from-alist
      (make-alist state-variables (cell-ref state-values)))
    (make-alist formals (message-parameters message))))
```

Figure 4.5: Our new meta-object design. (continued)

of instance variables that are mutable are packed in the mutable vector object `state-values`, and accesses to `state-values` are effected by using the writer methods `cell-set!` and `cell-ref`.

- The meta-object straightforwardly processes each method invocation request and provides mutual exclusion by using blocking operations (e.g., `acquire!` and `release!`). As a result, the meta-object is no longer a state-transition machine. The reader methods, which can be invoked without mutual exclusion, make it possible to define such a meta-object. If the meta-objects were defined with only writer methods, use of the blocking operations would easily lead to deadlock.

- For mutual exclusion, a meta-object has the instance variable `lock` in place of `mode` and `queue`. By default, `lock` is a simple semaphore that has the operations `acquire!` and `release!`. The user can replace `lock` with an arbitrary object, such as a FIFO queue and a priority queue, by means of the meta-level programming.

These characteristics solve the application problems of partial evaluation that were discussed in Section 4.1.1. (1) Under the execution model of Schematic[122], it is safe to assume that consecutive invocations of reader methods are not interrupted by other activities; we therefore can use most of partial evaluation techniques for sequential languages by regarding the reader methods as functions. (2) Since the "known" (static) information is propagated through the arguments of the method invocations, the partial evaluators easily use such information for specialization. (3) The mutual exclusion mechanism, which is implemented by the blocking operations, gets rid of the dynamic branches (conditionals with dynamic predicates) that would cause a termination-detection problem during specialization.

4.1.2.2 Protocols

Instance Variables.

`lock`: The variable `lock` is for mutual exclusion of writer methods. By default, this has a simple 'mutex' variable with `acquire!` and `release!` operations. By customizing this variable, various scheduling policies such as FIFO scheduling and priority scheduling can be realized. Since it represents the dynamic state of the base-level object, it is treated as a dynamic value during partial evaluation.

`state-variables`: The variable `state-variables` has the *names* of the base-level instance variables as a list. Since the names of instance variables are fixed (as stated before, our compilation technique assumes that the base-level and meta-level programs are statically given), this variable becomes a static value during partial evaluation.

`state-values`: The variable `state-values` has a mutable cell that contains a vector of *values* of the base-level instance variables. As this value represents the state of the base-level object, it is also dynamic for the partial evaluator.

`methods`: The variable `methods` has a set of base-level methods. The function `(find methods message)` searches a method specified in the `message`. This variable is static.

`evaluator:` The variable `evaluator` is a meta-interpreter object, which is discussed in the previous chapter. The meta-object evaluates base-level methods by invoking method `eval` and `eval-entry` of `evaluator`.

Methods. How the methods in Figure 4.4 and Figure 4.5 handle messages sent to the base-level object is explained as follows:

`(receive self message):` When a method of a base-level object is invoked, the method `receive` of the respective meta-object is actually invoked. The parameter `message` contains the necessary for the method invocation, including the selector name (i.e., the name of the method), the parameters to the method, and the 'reply-box' of the invocation (i.e., where the return value of the method is sent to). The method `receive` simply proceeds to invoke methods `accept-W` or `accept`, depending on the type of the base-level method that is to be invoked.

`(accept-W self message):` The method `accept-W` wraps the method `accept` in the code for mutual exclusion and update of base-level instance variables. It first creates a channel `c` by calling a primitive `make-channel`, then evaluates `(acquire! lock)`, and then calls the method `accept` of the same object with `c`.

In `accept`, a base-level method is selected and executed, as described below. Eventually, a vector of updated instance variables is sent to `c`, in response to the evaluation of `become` form in the base-level method. The evaluation of the form `(touch c)` extracts the vector from `c`; and the vector is assigned to `state-values` by evaluating the `cell-set!` form.

Finally, it evaluates `(release! lock)`, and returns the result of the base-level method.

Since the method `accept-W` itself does not modify its state, it is defined as a reader method. The mutual exclusion of (base-level) writer methods are achieved by performing `acquire!` and `release!` methods to an object `lock`. The instance variables of base-level object is recorded in an object `state-values`, and updated by evaluating `cell-set!`.

`(accept self message update-channel)`: The method `accept` defines the method invocation process that is common to reader and writer methods.

It first looks up a method for a given message by evaluating `(find method message)`, and then creates an evaluation environment by evaluating `(make-env ...)`. The body of the selected base-level method `(method-body m)` is then evaluated under the created environment by the method `eval` of `evaluator`.

`(make-env self formals message)`: The method `make-env` is auxiliary to `accept`, and creates an evaluation environment for the given formal parameters and actual parameters in the message. More formally, the created environment maps from the m'th element in `formals` to the m'th element in the actual parameter list in `message`, and from the n'th element in `state-variables` to the n'th element in the value of `(cell-ref state-values)`.

Note that the values of base-level instance variables are extracted before the execution of the body of the method, and the same values are used throughout the method execution. In other words, it will not observe the result of `cell-set!` once the environment is created. This conforms to the execution model of Schematic's reader/writer methods, explained in Section 1.1.3.

`(eval-entry evaluator exps env update-channel)`: The method `eval-entry` and other auxiliary methods are the methods of the meta-interpreter object `evaluator`. As explained in the previous chapter, they form a meta-circular interpreter that evaluates the expressions `exps` under the environment `env`.

The last parameter `last-channel` is for sending information on the instance variables by the `become` form in `exps`. For example, a base-level object has two instance variables `x` and `y`, and whose values are 0 and 1, respectively. When a base-level expression `(become e :x 123)` is being evaluated, the evaluator creates a vector containing 123 and 1, and then sends the vector to `update-channel`. The vector, as explained in the protocol of `accept-W`, will eventually be stored in `state-values` of the meta-object.

Compared to the meta-objects in ABCL/R, the proposed meta-objects have not only different programming style, but also give an extended semantics to the base-level

objects. In other words, the proposed meta-objects are not only defined by *using* the reader/writer methods, but also they *define* the semantics of reader/writer methods in base-level objects.

4.2 Implementation Issues

4.2.1 Optimization Using Partial Evaluation

In our proposed meta-objects, most operations are defined in the reader methods, and a few invocations on external objects are used for mutual exclusion and state modification. As we stated earlier, the meta-objects can, from the viewpoint of partial evaluation, be regarded as functional programs with I/O-type side-effects. In this section we describe an optimization framework for our meta-objects by using partial evaluation.

The biggest problem we face in using partial evaluation is that there are no partial evaluators appropriate for our purpose because the meta-object is written in a *concurrent* object-oriented language. Although there are studies on partial evaluators for concurrent languages[31,39,73], they focus on concurrency and pay little attention to the support of features crucial to sequential languages, such as function closures and data structures.

Our solution is to translate meta-objects into a sequential program and use a partial evaluator for a sequential language. Partial evaluation is applied for each base-level method invocation; i.e., the specialization point is a base-level method invocation. Since the methods of meta-objects exhibit almost sequential behavior, the partial evaluator for a sequential language can effectively optimize the meta-objects. Concurrency in the meta-objects will be residualized as applications to primitives.

Another problem is compatibility with other objects. The optimized object should support meta-level operations that are defined in the original meta-object. At the same time, the object should behave like a base-level object so that it can be used with other base-level objects. To satisfy these two requirements, our framework generates an object that combines the base- and meta-level objects in a single level. The object has the same methods that are in the original base-level object, and the body of each method is a specialized code of the meta-object.

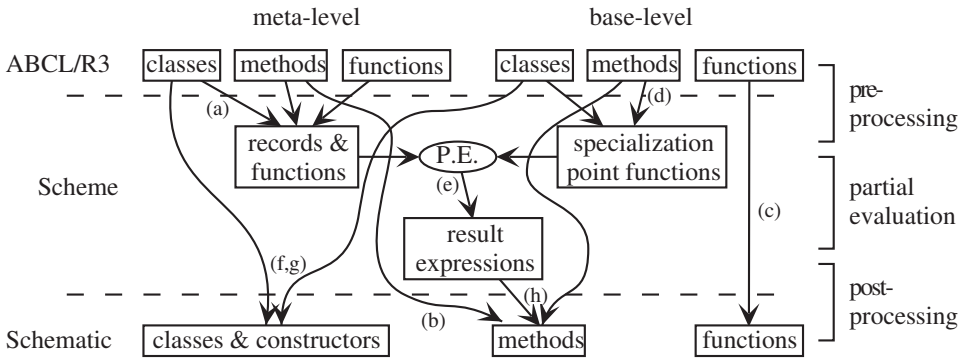


Figure 4.6: Overview of our optimization framework.

Figure 4.6 shows the overview of our optimization framework, in which there are three steps: (1) translation from ABCL/R3 to Scheme, (2) partial evaluation, and (3) translation from Scheme to Schematic. In the following subsections we explain each step in detail by using an example base-level program (Figure 4.7) and the default meta-object `metaobj` (Figure 4.4 and Figure 4.5).

4.2.1.1 Preprocessing

Meta-object definitions are translated into a Scheme program so that they can be processed by a Scheme partial evaluator (Figure 4.6(a)). A meta-level object is converted into a record[4] whose fields are its class name and values of instance variables. A reader method is converted into a dispatching function and a class-specific function. The former examines the class-name field in the receiver and calls a matching class-specific function. For example, the class `metaobject` is converted as follows:

- For each class, a record declaration and a constructor function are created. In the following example, we assume that the `define-record` declares a record type, and `make-metaobject` is a constructor of the record object.

  ;;; record declaration for class `metaobject`

[4]Since our partial evaluator does not natively support records, we further translate the record into `cons`-cells.

```
;;; 2d-point
(define-class point () x y)

;;; returns the distance from the origin—a reader method
(define-method point (distance self)
  (sqrt (+ (square x) (square y))))

;;; moves a point—a writer method
(define-method! point (move! self dx dy)
  (become #t :x (+ x dx) :y (+ y dy)))
```

Figure 4.7: Example base-level program.

```
(define-record metaobject (lock state-variables ...))

;;; constructor function for class metaobject
(define (metaobject lock state-variables ...)
  (make-metaobject lock state-variables ...))
```

- For each method of a class, a class-specific function is created. It first extracts the instance variables, and then performs the body of the method.

```
;;; class specific function for method receive of class metaobject
(define (metaobject*receive self message)
  (let ((lock (metaobject-lock self)) ; extracts instance variables
        (state-variables (metaobject-state-variables self))
        ...)
    (if (writer? (selector message))  ; body of the method
        ...)))

;;; class specific function for method accept-W of class metaobject
(define (metaobject*accept-W self message)
   ...)
```

- For each method name, a dispatcher function is created. It simply examines the record type of the first argument, and calls an appropriate function to the

```
;;; dispatching function for method receive
(define (receive self message)
  (cond ((metaobject?  self)
         (metaobject*receive self message))
        ((user-defined-meta? self)
         (user-defined-meta*receive self message))
        ...))
```

Invocations of writer methods that are defined at the meta-level should not be performed during the partial evaluation because they will modify the state of objects. Therefore, the writer methods are not passed to the partial evaluators but are instead simply copied into the resulting Schematic program (Figure 4.6(b)).

No translations are needed for the base-level definitions, since they are used as data for the meta-level program. Functions, however, are simply copied to the resulting Schematic program (Figure 4.6(c)).

4.2.1.2 Partial Evaluation

We partially evaluate the meta-level program for each *base-level method invocation.* For example, given the base-level program like that in Figure 4.7, the meta-level computation that will be processed is the one corresponding to the following base-level method invocation:

$$\texttt{(move!}\ p\ dx\ dy\texttt{)}$$
$$\text{where } p = \texttt{point}\{\texttt{x} = x,\ \texttt{y} = y\}.$$

The variables written in italic font (e.g., dx, dy, x, and y) are dynamic data. The data denoted by the variable p is partially static; it is known as an object of class `point`, but values of instance variables `x` and `y` are dynamic (unknown).

The corresponding meta-level computation is the following expression:

$$\texttt{(receive}\ \textit{mobj message}\texttt{)}$$

```
(define (specalization-point-move!-point state-values lock dx dy)
  (let ((mobj (metaobject 'point '((distance (self) ...) ...)
                          '(x y) state-values lock
                          (make-evaluator)))
        (message (message 'move! (list dx dy))))
    (*this-is-self* mobj)
    (receive mobj message)))
```

(The form `(*this-is-self* mobj)` is inserted to detect the identity of 'self' from the result of partial evaluation. This will be explained in the postprocessing step.)

Figure 4.8: Specialization point function for method `move!` of class `point`.

where

$$
\begin{aligned}
\mathit{mobj} \quad &= \texttt{metaobj}\{\texttt{class} = \texttt{'point}, \\
&\qquad \texttt{methods} = \texttt{'((distance (self) ...)} \ \ldots), \\
&\qquad \texttt{state-vars} = \texttt{'(x y)}, \ \texttt{state-values} = s, \ \texttt{lock} = l, \\
&\qquad \texttt{evaluator} = \texttt{(make-evaluator)}\}, \\
\mathit{message} &= \texttt{message}\{\texttt{selector} = \texttt{'move!}, \ \texttt{arguments} = \texttt{(list}\ \mathit{dx}\ \mathit{dy}\texttt{)}\}.
\end{aligned}
$$

To partially evaluate a meta-level computation like the above one, we generate a specialization point function for each base-level method (Figure 4.6(d)). The function takes as its arguments a vector of instance variables, lock, and parameters for the method. When called, it creates *mobj* and *message*, and it invokes the method `receive` on *mobj* (Figure 4.8). The function is specialized under the assumption that all the arguments are dynamic.

An online partial evaluator for Scheme[5] (Figure 4.6(e)) specializes not only the methods of `metaobj`, but also those of `evaluator`[5]. The compilation techniques of the meta-interpreter are described in Chapter 3.

[5]For convenience in executing the benchmark programs, instead of using a real meta-interpter we used a *fake evaluator* that directly executes the body of methods. This will be discussed in Section 4.2.2.

4.2.1.3 Postprocessing

The final step is to translate the results of partial evaluation (in Scheme) back into concurrent objects (in Schematic). Since our system performs partial evaluation in Scheme, the resulted program is also in Scheme. For example, our partial evaluator generates the following lambda expression from Figure 4.8:

```
(lambda (state-values lock dx dy)
  (let ((mobj (list 'metaobject 'point ...)))
    (*this-is-self* mobj)
    (acquire! lock)
    (let* ((state-update-channel0 (make-channel))
           (values0 (cell-ref state-values))
           (x0 (vector-ref values0 0))
           (y0 (vector-ref values0 1))
           (g0 (vector (+ x0 dx) (+ y0 dy))))
      (reply g0 state-update-channel0)
      (let ((new-state0 (touch state-update-channel0)))
        (cell-set! state-values new-state0)
        (release! lock)
        #t))))
```

Using this code, the purpose of the final step is to generate class declarations, constructor functions, and methods as shown in Figure 4.9 and Figure 4.10.

- For each combination of base- and meta-level classes, a specialized class is defined (Figure 4.6(f)). Since the class is a specialized version of the meta-level class, it has the same instance variables as the original meta-object. (E.g., the class `metaobject**point` in Figure 4.9.)

- A function that mimics the base-level constructor is defined for each specialized class (Figure 4.6(g)). For example, the function `point` in Figure 4.9 is a base-level constructor that creates an object belonging to class `metaobject**point` with proper initial values.

- Methods of the specialized classes are defined (Figure 4.6(h)). The name of each method is the same as that of the original base-level method. (The method `distance` and `move!` of class `metaobject**point` in Figure 4.9 and Figure 4.10 are examples.) The specialized object therefore has the same interface as the original base-level program. The body of the method is the result of partial evaluation. Note that because the generated methods are specialized versions of `receive` of the meta-object, they should be defined as reader methods regardless of the type of the corresponding base-level method.

- There is a problem around the identity of 'self,' which is solved by a *marker function.* Since a meta-object translated into cons-cells at the preprocessing, we may lose the identity of the object after partial evaluation. If the reference to the 'self' is returned as a result, assigned to a variable, or passed to other object, the residual program returns, assigns, or passes a reference to the cons-cells, which are invalid after the partial evaluation.

  To avoid this, we insert a form that applys the cons-cells for the 'self' to a marker function `*this-is-self*` as is shown in Figure 4.8. After partial evaluation, the postprocess scans the residual program to detect the value applied to the marker function. The identity of self is recoverd by merely replacing the value with variable `self`. Since our partial evaluator is using the graph representation[5, 107, 127], the application form and the cons-cells are correctly preserved during the partial evaluation.

When a meta-object is specialized with respect to a reader method, the optimized method has the essentially same definition as the original base-level method, except for the indirect accesses to the instance variables (cf. the method `distance` in Figure 4.9). When it is specialized with respect to a writer method, on the other hand, the optimized method evidently contains extra operations. Although most of the operations in the optimized method are the same as the operations performed in a writer method in Schematic, others are amenable to further optimization. For example, the newly created vector of instance variables `g0` is handed over by means of `reply` and `touch` operations in the same thread because our current partial evaluator regards those operations as mere "unknown" functions. An optimized method

```
;;; a combined class of metaobject w.r.t. point
(define-class metaobject**point ()
  class methods state-vars state-values lock evaluator)

;;; constructor
(define (point x y)
  (metaobject**point
   (quote *metaobject*) (quote *methods*) (quote (x y))
   (make-cell (vector x y)) (make-lock) (quote *evaluator*)))

;;; reader method
(define-method metaobject**point (distance self)
  (begin (let* ((values0 (cell-ref state-values))
                (x0 (vector-ref values0 0))
                (y0 (vector-ref values0 1))
                (g0 (square x0))
                (g1 (square y0)))
           (sqrt (+ g0 g1)))))
```

Figure 4.9: Result of optimization (the underlined expressions come from the base-level method).

```
;;; writer method
(define-method metaobject**point (move! self dx dy)
  (begin (acquire! lock)
         (let* ((state-update-channel0 (make-channel))
                (values0 (cell-ref state-values))
                (x0 (vector-ref values0 0))
                (y0 (vector-ref values0 1))
                (g0 (vector (+ x0 dx) (+ y0 dy))))
           (reply g0 state-update-channel0)
           (let ((new-state0 (touch state-update-channel0)))
             (cell-set! state-values new-state0)
             (release! lock)
             #t))))
```

Figure 4.10: Result of optimization (continued)

less extra operations could be produced by using partial evaluators for concurrent languages or by applying static analysis for concurrent programs[44,58,59] to the resulting code.

4.2.2 Performance Evaluation

To evaluate the efficiency of our partially evaluated meta-objects, we executed benchmark programs in the following three ways:

PE(partially evaluated): The default meta-object was partially evaluated with respect to each benchmark program, and the generated code was further compiled by Schematic. This showed the performance of our optimization framework.

INT(interpreted): The default meta-object was directly compiled by Schematic, and then the compiled code interpreted the benchmark programs. This showed the performance of naïvely implemented meta-objects.

NR(nonreflective): The benchmark programs were directly compiled by Schematic[6]. This showed the performance of nonreflective languages.

All programs were executed on Sun UltraEnterprise 4000 that had 1.2GB memory, 14 UltraSparc processors,[7] each operating at 167MHz, and was running SunOS 5.5.1.

The differences between the PE and INT performances show the amount of speedup gained by partial evaluation, while the differences between the PE and NR performance show the *residual overheads*—the overheads that the partial evaluator fails to eliminate.

The overheads solely caused by the meta-objects, were evaluated by executing the body expressions in PE and INT without meta-interpreters. For example, when a base-level program has an expression "`(distance p)`," then a meta-object looks up `distance` in its method table and extracts instance variables from `p`. However, the method body "`(sqrt (+ (square x) (square y)))`" should be executed directly. To do this, we generate a *fake evaluator* for each base-level class (Figure 4.11). Without fake evaluators, interpretive execution of method bodies would make an overwhelmingly large contribution to the execution time in INT. The fake evaluators are also useful for skipping over the partial evaluation of meta-interpreters whenever a base-level object uses only the default meta-interpreter.

4.2.2.1 Base-level Applications

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

Null Readers and Null Writers: Elapsed time for 1,000,000 method invocations was measured by repeatedly calling a null method on an object. We tested objects with different numbers of instance variables (i) and tested methods with different numbers of arguments (j). The average time over some parameter combinations ($i \in \{0,5,10\}, j \in \{1,5,10\}$) are shown as a representative result.

[6] Our Schematic compiler has some overheads for concurrent execution; a sequential program (Richards) compiled by a sequential Scheme compiler (DEC Scheme-to-C) was faster than the one compiled by Schematic by a factor of 5.4.

[7] Though we used a multi-processor machine, the programs are executed on a single processor execution.

```
;;; Class definition
(define-class evaluator**point ())

;;; The method called by the meta-object.
(define-method evaluator**point (eval-begin self method-name exp env)
  (cond ((eq? method-name 'distance) ; for method distance
         (let ((x (lookup 'x env)) (y (lookup 'y env)))
           (sqrt (+ (square x) (square y)))))
        ((eq? method-name 'move!)    ; for method move!
         (let ((x (lookup 'x env)) (y (lookup 'y env))
               (dx (lookup 'dx env)) (dy (lookup 'dy env)))
           (let ((new-values (vector (+ x dx) (+ y dy))))
             (update self new-values))))))
```

Each clause of the `cond` form in `eval-begin` corresponds to the method of the base-level class `point`. A clause is selected by the argument `method-name`. The body part of a clause has the code for extracting the base-level arguments and instance variables and for the method body. A `become` form in the original program is converted into an invocation of the `update` method of the meta-object, which takes a vector of the updated instance variables as an argument.

Figure 4.11: "Fake" evaluator for `point`.

Become: Elapsed time for 1,000,000 invocations of writer methods which update instance variables was measured by repeatedly calling a method that immediately performs `become`. We tested objects with different numbers of updated variables (k), and the average time over the parameter combinations $i = 10, j = 1, k \in \{1, 5, 10\}$ is shown as the representative result[8].

Richards: The Richards benchmark is an operating system simulation that is used as a nontrivial program in evaluating several object-oriented languages[14].

RNA: RNA is a parallel search program for predicting RNA secondary structures[91, 119]. This program uses an object to share information on the best answers that have been discovered among concurrently running threads. Each thread in the system constantly checks the object, and terminates (i.e., *prunes*) itself when there are no chances to find a better answer than the best answers in the object. When a thread find a new answer, it updates the best answers entry in the object for the sake of pruning other threads.

Since Richards and RNA use both functions and methods, their executions show how the efficiency of the meta-objects affects overall execution speed in realistic applications.

The results are summarized in Table 4.1. As the "improvement" column shows, the programs in PE are more than four times faster than the ones in INT. This improvement is significant even in realistic applications such as Richards and RNA, whose speeds are increased by factors of 6.8 and 30.8, respectively.

As the "residual overheads" column shows, the programs in PE are slower than the ones in NR by factors of 1.1–3.0. These overheads are mainly due to the limitations of current partial evaluators, as we have pointed out in Section 4.2.1.3. In fact, when we further optimized the partially evaluated meta-objects for Become by hand—eliminating obvious channel communications, etc.—the average factor by which programs are slowed because of residual overheads was reduced to 1.4.

[8] The combination of the values of i and j yields the worst result in Null Writers.

Table 4.1: Performance improvement and residual overheads.

benchmark applications	elapsed time (sec.) PE	INT	NR	improvement INT/PE	residual overheads PE/NR
Null Readers	3.2	107.7	2.3	33.6	1.4
Null Writers	40.7	190.8	16.9	4.7	2.4
Become	46.6	272.8	15.7	5.9	3.0
(w/manual opt.)	(21.3)			(12.8)	(1.4)
Richards	20.7	140.7	9.4	6.8	2.1
RNA	1.7	53.3	1.6	30.8	1.1

4.2.2.2 Performance of Customized Meta-objects

The above benchmark programs were executed under the default meta-objects, but of more practical interest is the efficiency of *customized* meta-objects. The next benchmark program was a bounded-buffer that uses the guarded method invocation mechanism, which is implemented by a customized meta-object. Since the guarded methods are not directly supported in Schematic, we simulated them by user-level programming, in which objects are programmed to check the guard conditions and to suspend/continue their invocation requests. The programs are described in Appendix B.

Table 4.2 shows the elapsed time for 1,000 read/write operations from/to a bounded buffer whose size is 10. The PE buffer shows almost the same efficiency as does the NR one. This result could be understood as that the the overheads caused by frequent method invocations in NR cancel out the residual overheads in the PE buffer. The NR buffer uses three methods in order to represent a guarded method. On the other hand, the PE buffer uses only one because the partial evaluator successfully inlines the methods of the meta-object that deal with the guarded methods.

The partially evaluated meta-objects are approximately 10 percent faster than the interpreted ones (INT). This improvement is less significant than that observed with

Table 4.2: Performance of bounded buffer with guarded methods.

	elapsed time (sec.)			improvement	residual overheads
	PE	INT	NR	INT/PE	PE/NR
Bounded Buffer	3.94	4.46	3.96	1.13	0.99

the previous benchmarks. We conjecture that this is because each of these benchmark programs requires a large number of context switches, and context-switching is expensive in the current Schematic implementation. The time spent for context-switching is thus so great that the efficiency differences between the three programs are relatively small.

4.3 Related Work

In CLOS Meta-Object Protocols (MOP), meta-level methods are split into functional and procedural ones for caching (or *memoization*)[55, 56]. This splitting approach in principle similar to our meta-object design, but the memoization technique requires more careful protocol design because the unit of specialization is function. Thus the "functional" methods cannot include operations that touch dynamic data. On the other hand, such operations can be written in our reader methods, since the partial evaluator automatically residualizes them.

There are several studies that specialize meta-objects with respect to base-level programs. For example, an implementation of the early version of OpenC++[18][9] specializes meta-objects. Although the technique is effective for some cases, it is limited since it is based on "idiom recognition." The proposed technique in the dissertation, which is based on partial evaluation, is effective to a larger variety of meta-objects.

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

4.4 Summary

In this chapter, a method for designing meta-objects in the reflective language ABCL/R3 is described, and a framework for their optimization using partial evaluation is presented. In the meta-object's description, operations that are state-related are separated from operations that are not, and it is this separation that makes partial evaluation effective. The meta-objects and their reader methods are translated into records and functions in Scheme, and they are then optimized by using a Scheme partial evaluator. The optimized code is a combination of the base-level and meta-level programs, a combination from which most interpretive operations at the meta-level (such as the method dispatch and the manipulation of the environment) have been removed. Effectiveness of this optimization framework is shown by benchmark programs in which the partially evaluated objects run significantly faster than the interpretive meta-objects. Moreover, the partial evaluation lets a program with customized meta-objects run as efficiently as an equivalent nonreflective program.

Chapter 5

Discussions

5.1 Dynamic Modification

In the dissertation, we concentrated on the situation where both base- and meta-level programs are not modified at run-time. This is a reasonable assumption for many practical reflective applications, and a crucial prerequisite for the first Futamura projection.

However, reflection has potential to exploit dynamic modifications. Since base-level programs are treated as data at the meta-level, it can elegantly describe processes that changes base-level programs. Since a meta-level program can embody a policy of the language system, replacement of meta-level programs can realize dynamic changes of policies.

From the viewpoint of implementation, types of dynamic modifications can be classified by whether meta-level objects and base-level objects can be modified dynamically, and whether the combination of them—which meta-objects are applied to a base-level object—can be specified dynamically:

1. No ability to modify dynamically; i.e., both base- and meta-level object definitions are known at the compile-time, and the combination of them are also known. The techniques in the dissertation assume this case, where meta-level objects are partially evaluated with respect to base-level objects.

2. Only combinations of base- and meta-level objects can be specified dynamically.

3. Base-level objects can be modified dynamically, but meta-level objects are unmodifiable.

4. Both base-level and meta-level objects can be modified dynamically.

For the last three cases, it is possible to dynamically apply partial evaluation (and subsequent compilation) whenever the system gets definitions of meta-level and base-level objects. However, sluggishness of partial evaluation would easily sap the overall performance.

Possible solutions for the case 2. and 3. are *code versioning* and *dynamic code generation,* respectively. We briefly describe those techniques below.

5.1.1 Code Versioning

When definition of meta-level and base-level objects is statically given, it is possible to apply partial evaluation, and support dynamic modifications to the meta-interpreter outside of the partial evaluation framework. Firstly we have restricted ABCL/R3 so that a base-level object is allowed to choose its meta-level objects (i.e., meta-object and meta-interpreter) only at its creation-time[1]. For a single method of a base-level object and definitions of meta-level objects (specified by a pair of classes of meta-objects and meta-interpreters), *multiple compiled methods* are generated for each interpreter. On creating an object, appropriate compiled object is selected according to the specified meta-level objects.

For brevity, we describe meta-object classes as $MO_1, MO_2, \ldots$, meta-interpreter classes as $MI_1, MI_2, \ldots$, and base-level object classes as $B_1, B_2, \ldots$. Our compilation framework partially evaluates a meta-object MO_i and a meta-interpreter MI_j with respect to a base-level B_k, and yields a combined class definition C_{ijk}, for all combinations of i, j, and k. (Needless to say, if the meta-object classes and meta-interpreter classes that are used by a base-level class are statically known, only possible combinations should be generated.) At run-time, when a base-level object of class B_k is being created with a meta-object MO_i and meta-interpreter MI_j, then the pre-compiled definition C_{ijk} will be used for actual run.

[1] Replacement after object's creation could be possible with more elaborate run-time support.

An obvious problem of this approach is space efficiency. Since the number of specialized classes is product of the numbers of meta-object, meta-interpreter, and base-object classes, this technique cannot be applied without heuristics that reduces the number of combinations.

5.1.2 Dynamic Code Generation

Using a dynamic code generation technique, the specialization process could be accelerated so that meta-level objects can be dynamically specialized at run-time. Dynamic code generation[21,27,28,61,62,99,115] is a technique to construct a specializer for each target program. The construct specializer receives an input for the target program, and directly generates specialized code in machine instructions. By preparing compiled machine instructions at the construction-time of the specializer, the code generation is tremendously faster than the partial evaluation and subsequent compilation process.

Let L be a high-level language (e.g., Scheme), M be a machine language, p^L be a program written in L, p^M be a compiled version of p, and $DCGG$ be a dynamic code generator-generator for language L. $DCGG$ creates a code generator specialized to a given program:

$$DCGG(p^L) = DCG^M_{p^L},$$

The generated program $DCG^M_{p^L}$ is a code generator that takes one of the p^L's inputs, and generates a specialized version of p^L:

$$DCG^M_{p^L}(x) = p^M_x. \tag{5.1}$$

The result p^M_x is a compiled function that takes the remaining argument of p^L, and returns the same answer to p^L:

$$p^M_x(y) = r \qquad \text{if } p^L(x,y) \text{ returns } r.$$

Let C be a compiler that compiles a program in L. Application of partial evaluation and compilation can basically generate the same program to p^M_x:

$$PE(p^L, x) = p^L_x, \text{ and then} \tag{5.2}$$

$$C(p^L_x) = p^M_x. \tag{5.3}$$

The advantages of dynamic code generation is (a) the execution of $DCG^M_{p^L}$ does not *interpret* p, and (b) the specialization process (5.1) is much more efficient than the partial evaluation (5.2) plus subsequent compilation (5.3) processes, since $DCG^M_{p^L}$ directly generates compiled code.

We could use this technique to reflective language by applying a customized meta-level program L' to the dynamic code generator-generator, having a code generator $DCG^M_{L'}$:

$$DCGG(L') = DCG^M_{L'}.$$

The code generator $DCG^M_{L'}$ can compile program in L':

$$DCG^M_{L'}(p^{L'}) = {L'}^M_p.$$

The advantages of the dynamic code generation technique are as follows:

- The code generator for a meta-level program ($DCG^M_{L'}$) can be constructed without base-level program (p). This means that when a base-level program is modified, the system only applies the second process to obtain a new specialized code. Using partial evaluators, on the other hand, it has to partially evaluate the meta-level object.

- The specialization process (i.e., generation of ${L'}^M_p$) is faster, as it directly generates machine instructions. Using partial evaluation, since it is a source-to-source transformation technique, compilation of partially evaluated programs is needed to obtain machine instructions.

Despite those advantages, there are a number of difficulties in order to bring the technique into practical reflective systems[116, 117]. They are beyond the scope of this dissertation.

5.2 Infinite Tower

The dissertation also assumes that the target reflective language consists of only the base- and meta-levels. Many models of reflective languages, on the other hand, have an infinite tower of meta-levels.

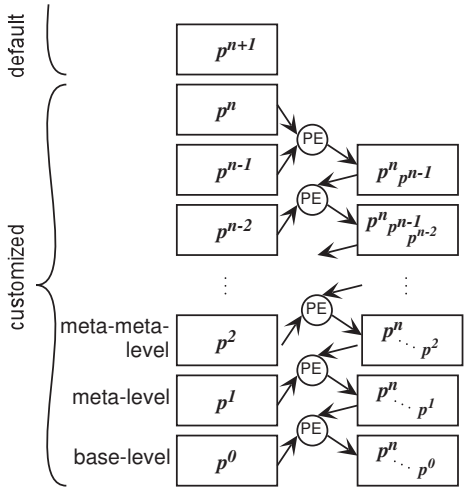


Figure 5.1: Compilation of a reflective tower.

We claim that most practical applications customize only the meta-level. Moreover, our proposed techniques could be useful to applications that exploit meta-meta-level and above, by repeatedly applying partial evaluation process.

Assume we have an application program which customizes up to the n'th meta-level; i.e., $(n+1)$'th meta-level and above are default. To compile such an application, we follow the steps shown in Figure 5.1. We first specialize the definition at the n'th meta-level (hereafter we write p^n) with respect to p^{n-1}. The generated program, which we write $p^n_{p^{n-1}}$, serves as the meta-level for p^{n-2}, and runs without meta-levels. Then further specialization yields $p^n_{p^{n-1}\,p^{n-2}}$, and so forth. After enough repetition, we will have a program that is a "specialized the whole meta-level with respect to the base-level program."

Several researchers have pointed out similar ideas[51, chapter 6][109], and Asai showed that an application with customized meta-meta-level is successfully specialized[7].

The effectiveness of the repeated specialization in our reflective language ABCL/R3 depends on whether the specialization processes preserve binding-times of programs and the restrictions that are placed for successful partial evaluation. For exam-

ple, we have a meta-interpreter definition m^1, whose meta-interpreter m^2 is also customized. Then we first specialize the meta-meta-interpreter with respect to the meta-interpreter:

$$PE(m^2, m^1) = m^2_{m^1}.$$

In order to partially evaluate the resulted meta-interpreter $m^2_{m^1}$ with respect to a base-level program, methods of $m^2_{m^1}$ must have a static expression as its argument, otherwise it is hardly specialized.

Since preservation of those properties is a subtle problem, we have not reached any conclusion. However, we have a prospect that our approach—separating meta-objects' method into readers and writers, restricting methods of meta-interpreters to only readers—would also useful to preserve such desired properties.

5.3 Coexistence with Compile-time MOP

As is introduced in Section 2.3.3, the compile-time MOP performs meta-level computation before the execution of base-level programs. Since the execution stages are separated, the meta-level computation is defined as a program transformer, similar to Lisp macros.

Since the compile-time MOP indirectly modifies semantics of base-level languages by means of program transformation, it is considered more difficult than the run-time one. However, there are some advantages. It has no 'residual overheads,' since it clearly separates the compile-time and run-time computation. It also has chances to give clearer description of global transformation; e.g., changing function names all at once, optimizations based on global type analysis, etc.

We claim that those compile-time and run-time MOPs are not conflicting, but they should coexist and complement each other. In fact, several studies are working on this issue[20, 118].

Simple program transformers can be integrated to our approach, in which meta-level objects interprets base-level programs, without degrading run-time performance. Let `transformer` be a function that takes an expression and returns a transformed expression which is supposed to be performed at the compile-time. The following simple meta-interpreter class applies the transformer to every base-level expression:

```
(define-delegation-class transforming-eval ())
(define-method transforming-eval (eval-entry self exp env)
  (eval-entry super (transformer exp) env))
```

(The class overrides the method `eval-entry`, so that expressions are not transformed doubly.)

Since `exp` is static data, the application of `exp` to `transformer` can be taken place at the partial evaluation time, imposing no overheads upon the run-time execution.

However, this simple implementation of transformers has drawbacks. (1) The transformation process in this approach is slower than the one in compile-time MOP implementations, since partial evaluators usually interpretively execute target programs (i.e., `transformer`, in this case). (2) `Transformer`s can not exploit global information, since they are requested to be pure functional for successful partial evaluation.

5.4 Preactions

We proposed the preactions mechanism to preserve traces of I/O-type side-effects in online partial evaluation. Although the preactions are originally proposed in this study, there are similar techniques in offline partial evaluators, such as Similix's let-insertion[51, chapter 5]. We believe that preaction could be a good basis to handle side-effects in general. In fact, it is extended so that programs containing assignments to data structures (e.g., Scheme's `set-car!`) are correctly partially evaluated[5], though it is beyond the thesis.

Chapter 6

Conclusion

6.1 Summary of the Dissertation

The dissertation presented a reflective architecture of a concurrent object-oriented language, and its optimization frameworks using partial evaluation. The main contributions are as follows:

Framework to apply partial evaluation to meta-interpreters: (Chapter 3) It achieves orders of magnitude improvement in execution of customized meta-interpreters. The framework also enables to the re-usable and flexible meta-interpreter design by allowing the fine grain methods and the delegation mechanism.

Delegation mechanism to extend meta-interpreters: (Chapter 3) Instead of the traditional inheritance model, it enables to define compositional and scope-controlled extensions to the meta-interpreters. The mechanism is especially useful to modify the semantics of existing programming constructs that appear in a specific form.

Reflective annotations: (Chapter 3) They allow the programmer to embed various directives to the meta-level in base-level programs, without modifying the structure of the programs. The semantics of annotations at the meta-level can be defined in reflective ways.

New meta-object design suitable for partial evaluation: (Chapter 4) Unlike previous meta-object designs that implement state transition machines, our new meta-objects have straight control-flow and separated description of state-related operations, by exploiting the Schematic's reader/writer methods model, and can be partially evaluated effectively.

Framework to apply partial evaluation to meta-objects: (Chapter 4) It effectively specializes a meta-object definition by translating the definition into Scheme functions and records, then applies a Scheme partial evaluator with respect to each base-level method, and finally generates a Schematic object definition that combines properties of the base-level object and the meta-level object in a single level.

As far as the author knows, it is the first reflective concurrent object-oriented language that achieves the performance close to non-reflective languages. The flexibility is demonstrated by several programming examples, and the efficiency is demonstrated by benchmark programs.

In addition, there is also a contribution to partial evaluation technology:

Preactions: (Chapter 3) The extended partial evaluation rules with preactions keep track of history of side-effecting operations to the symbolic values, and preserve correct traces of I/O type side-effects during partial evaluation.

The technique is proved to be useful for partial evaluation of reflective programs, especially in concurrent and object-oriented environments, where inter-object communication is represented as I/O operations. The preaction technique is further extended to support assignment type operations[5].

6.2 Future Direction

Over the past decade, the notion of reflection and meta-level architectures has become widely accepted. For example, Java adopts the notion of reflection, though it is limited[48]. Other than programming languages, many recent systems are also incorporating meta-models for extensibility, and allow to define tailored system for

specific purposes. Those systems include operating systems[128, 129], a window system[101] modeling languages[102], documentation and data exchange format[10], and management of evolving software[68, 82, 90].

We believe that the notions and techniques presented in the dissertation would also be useful to those extensible systems. Especially, (1) not only ad-hoc mechanisms for extension, but more generic approach like meta-interpreters could be applicable to those systems, without imposing serious performance drawbacks, (2) partial evaluation and the similar specialization techniques would be useful to optimize such systems, and (3) experiences in designing our meta-level architecture (e.g., separation of state-related operations, etc.) would be transfered to such systems for successful partial evaluation.

Bibliography

[1] ACM. *Proceedings of Conference on Programming Language Design and Implementation (PLDI'96)*, Vol. 31(5) of *ACM SIGPLAN Notices*, Philadelphia, PA, May 1996.

[2] ACM. *Proceedings of Conference on Programming Language Design and Implementation (PLDI'97)*, Vol. 32(5) of *ACM SIGPLAN Notices*, Las Vegas, NV, June 1997.

[3] ACM. *Proceedings of Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97)*, Vol. 32(12) of *ACM SIGPLAN*, Amsterdam, June 1997.

[4] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[5] Kenichi Asai, Hidehiko Masuhara, and Akinori Yonezawa. Partial evaluation of call-by-value lambda-calculus with side-effects. In *Proceedings of Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97)* [3], pp. 12–21.

[6] Kenichi Asai, Satoshi Matsuoka, and Akinori Yonezawa. Duplication and partial evaluation to implement reflective languages. In *Proceedings of OOPSLA '93 Workshop on Object-Oriented Reflection and Metalevel Architectures*, Washington, D.C., September 1993.

[7] Kenichi Asai, Satoshi Matsuoka, and Akinori Yonezawa. Duplication and partial evaluation —for a better understanding of reflective languages—. *Lisp and Symbolic Computation*, Vol. 9, pp. 203–241, 1996.

[8] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of Symposium on Principles & Practice of Parallel Programming (PPOPP)*, Seattle, WA, March 1990.

[9] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. In Norman Meyrowitz, editor, *Proceedings of Object-Oriented Programming Systems, Languages, and Applications*, Vol. 21(11) of *ACM SIGPLAN Notices*, Portland, OR, October 1986. ACM, ACM.

[10] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0. Technical Report REC-xml-19980210, World Wide Web Consortium, February 1998.

[11] Frank Buschmann, Konard Kiefer, Frances Paulisch, and Michael Stal. The meta-information-protocol: Run-time type information for C++, November 1992.

[12] Craig Chambers and David Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-type object-oriented programs. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*. ACM, June 1990.

[13] Craig Chambers and David Ungar. Making pure object-oriented languages practical. In *Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 1–15, Phoenix, Arizona, October 1991.

[14] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of SELF, a dynamically-type object-oriented language based on prototypes. In *Proceedings of OOPSLA'89 (SIGPLAN Notices Vol.24, No.10)*, pp. 49–70, New Orleans, LA, October 1989. ACM.

[15] Shigeru Chiba. A metaobject protocol for C++. In Loomis [66], pp. 285–299.

[16] Shigeru Chiba. *A Metaobject Protocol for Enabling Better C++ Libraries*. PhD thesis, Department of Information Science, Faculty of Science, University of Tokyo, November 1996.

[17] Shigeru Chiba and Takashi Masuda. Designing an extensible distributed language with a meta-level architecture. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, 1993. LNCS 707.

[18] Shigeru Chiba and Takashi Masuda. Open C++ and its optimization. In *Proceedings of OOPSLA '93 Workshop on Object-Oriented Reflection and Metalevel Architectures*, Washington, D.C., September 1993.

[19] Shigeru Chiba and Takashi Masuda. A reflective language OpenC++ and its application to distributed computing. *Computer Software*, Vol. 11, No. 3, pp. 33–48, May 1994. (In Japanese).

[20] Shigeru Chiba and Michiaki Tatsubori. A yet another java.lang.Class. In *ECOOP'98 Workshop on Reflective Object-Oriented Programming Systems*, Brussels, Belgium, July 1998.

[21] Charles Consel and Francois Noël. A general approach for run-time specialization and its application to C. INRIA/IRISA Technical Report draft 946, July 1995.

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

[23] Olivier Danvy and Karoline Malmkjær. Intensions and extensions in the reflective tower. In *Proceedings of Conference on LISP and Functional Programming*, pp. 327–341. ACM, 1988.

[24] Jim des Rivières. Control-related meta-level facilities in LISP. In P. Maes and D. Nardi, editors, *Meta-Level Architectures and Reflection*, pp. 101–110, 1988.

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

[26] Adriana Lopes Diaz. *An Object-Oriented Reflective Simulation Environment for Distributed Algorithms*. PhD thesis, Ottawa-Carleton Institute for Computer Science, School of Computer Science, Carleton University, Ottawa, Ontario, Canada, April 1996.

[27] Dawson R. Engler. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI'96)* [1], pp. 160–170.

[28] Dawson R. Engler and Todd A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS'94)*, pp. 263–272, San Jose, CA, October 1994. ACM. (published as SIGPLAN Notices Vol.29, No.11).

[29] Yoshihiko Futamura. Partial evaluation of computation process—an approach to a compiler-compiler. *Systems, Computers, Controls*, Vol. 2, No. 5, pp. 45–50, 1971.

[30] Richard P Gabriel. *Performance and Evaluation of Lisp Systems*. MIT Press, 1985.

[31] Marc Gengler and Matthieu Martel. Self-applicable partial evaluation for the pi-calculus. In *Proceedings of Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97)* [3].

[32] Ananth Y. Grama, Vipin Kumar, and Ahmed Sameh. Scalable parallel formulation of the Barnes-Hut method for n-body simulations. In *Proceedings of Supercomputing*, pp. 439–448, 1994.

[33] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *Transactions on Programming Languages and Systems*, Vol. 7, No. 4, pp. 501–538, April 1985.

[34] Anders Haraldsson. A partial evaluator, and its use for compiling iterative statements in Lisp. In *Conference record of Symposium on Principles of Programming Languages*, pp. 195–202, Tuscon, Arizona, 1978. ACM.

[35] High Performance Fortran Forum. High performance Fortran language specification. *Scientific Programming*, Vol. 2, No. 1, June 1993.

[36] Michanel Hirsch, William Silverman, and Ehud Shapiro. Computation control and protection in the Logix system. In Ehud Shapiro, editor, *Concurrent Prolog: Collected Papers*, Vol. 2, chapter 20, pp. 28–45. MIT Press, 1987.

[37] Yasuaki Honda and Mario Tokoro. Soft real-time programming through reflection. In Yonezawa and Smith [131], pp. 12–23.

[38] Masakazu Hori and Koichiro Ochimizu. Shared data management mechanism for software development based on a reflective object-oriented model. *Computer Software*, Vol. 13, No. 1, pp. 37–54, January 1996.

[39] Haruo Hosoya, Naoki Kobayashi, and Akinori Yonezawa. Partial evaluation scheme for concurrent languages and its correctness. In *Euro-Par'96 Parallel Processing*, No. 1123 in Lecture Notes in Computer Science, pp. 625–632, 1996.

[40] Yuuji Ichisugi. *A Reflective Object-Oriented Concurrent Language for Distributed Environments*. Ph. D. Thesis, University of Tokyo, March 1993.

[41] Yuuji Ichisugi, Satoshi Matsuoka, and Akinori Yonezawa. RbCl: A reflective object-oriented concurrent language without a run-time kernel. In Yonezawa and Smith [131], pp. 24–35.

[42] Yuuji Ichisugi and Yves Roudier. Extensible Java preprocessor kit and tiny data-parallel Java. In *International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE'97)*, No. 1343 in Lecture Notes in Computer Science. Springer-Verlag, December 1997.

[43] Atsushi Igarashi. Study on mechanisms for multi-object synchronization and their implementation. Senior's thesis, Department of Information Science, University of Tokyo, 1994., 1994.

[44] Atsushi Igarashi and Naoki Kobayashi. Type-based analysis of usage of communication channels for concurrent programming languages. In *Proceedings of International Static Analysis Symposium (SAS'97)*, Vol. 1302 of *Lecture Notes in Computer Science*, pp. 187–201. Springer-Verlag, 1997.

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

[46] Yutaka Ishikawa, Atsushi Hori, Mitsuhisa Sato, Motohiko Matsuda, Jorg Nolte, Hiroshi Tezuka, Hiroki Konaka, Munenori Maeda, and Kazuto Kubota. Design and implementation of metalevel architecture in C++: MPC++ approach. In Kiczales [54], pp. 153–166. Proceedings are available at http://jerry.cs.uiuc.edu/reflection/reflection96/index.html.

[47] Yutaka Ishikawa, Atsushi Hori, Hiroshi Tezuka, Motohiko Matsuda, Hiroki Konaka, Munenori Maeda, Takashi Tomokyo, Jörg Nolte, and Mitsuhisa Sato. MPC++. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, chapter 11, pp. 429–464. MIT Press, 1996.

[48] JavaSoft, Mountain View, CA. *Java Core Reflection: API and Specification*, January 1997.

[49] Stanley Jefferson and Daniel P. Friedman. A simple reflective interpreter. In Yonezawa and Smith [131], pp. 48–55.

[50] Neil D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, Vol. 28, No. 3, pp. 480–503, September 1996.

[51] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[52] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *Transactions on Computer Systems*, Vol. 6, No. 1, pp. 109–133, February 1988.

[53] Gregor Kiczales. Towards a new model of abstraction in software engineering. In Yonezawa and Smith [131].

[54] Gregor Kiczales, editor. *Reflection'96*, San Francisco, California, April 1996. Proceedings are available at http://jerry.cs.uiuc.edu/reflection/reflection96/index.html.

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

[56] Gregor Kiczales and Luis Rodriguez. Efficient method dispatch in PCL. In *Proceedings of Conference on LISP and Functional Programming*, pp. 99–105, Nice, France, June 1990.

[57] Amir Kishon, Paul Hudak, and Charles Consel. Monitoring semantics: A formal framework for specifying, implementing, and reasoning about execution monitors. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*, pp. 338–352, Toronto, Ont., June 1991. (published as ACM SIGPLAN Notices 26(6).).

[58] Naoki Kobayashi, Motoki Nakade, and Akinori Yonezawa. Static analysis of communication for asynchronous concurrent programming languages. In *Second International Static Analysis Symposium (SAS'95)*, Vol. 983 of *Lecture Notes in Computer Science*, pp. 225–242. Springer-Verlag, 1995.

[59] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *Conference record of Symposium on Principles of Programming Languages*, pp. 358–371, January 1996.

[60] John Lamping, Gregor Kiczales, Luis Rodriguez, and Erik Ruf. An architecture for an open compiler. In Yonezawa and Smith [131], pp. 95–106.

[61] Peter Lee and Mark Leone. Optimizing ML with run-time code generation. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI'96)* [1], pp. 137–148.

[62] Mark Leone and Peter Lee. Lightweight run-time code generation. In *Partial Evaluation and Semantics-Based Program Manipulation*, pp. 97–106, Orlando, FL, June 1994. ACM SIGPLAN. published as Technical Report 94/9, Department of Computer Science, The University of Melbourne.

[63] Barbara Liskov. Distributed programming in Argus. *Communications of the ACM*, Vol. 31, No. 3, pp. 300–312, March 1988.

[64] Barbara Liskov, Dorothy Curtis, Paul Johnson, and Robert Scheifler. Implementation of Argus. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, pp. 111–122, 1987.

[65] Barbara Liskov and Robert Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *Transactions on Programming Languages and Systems*, Vol. 5, No. 3, pp. 381–404, July 1983.

[66] Mary E. S. Loomis, editor. *Proceedings of Object-Oriented Programming Systems, Languages and Applications*, Vol. 30(10) of *ACM SIGPLAN Notices*, Austin, TX, October 1995. ACM.

[67] Cristina Videira Lopes. Adaptive parameter passing. In Kokichi Futatsugi and Satoshi Matsuoka, editors, *Object Cechnologies for Advanced Software (ISOTAS'96)*, Vol. 1049 of *Lecture Notes in Computer Science*, pp. 118–136, Kanazawa, Japan, March 1996. Springer-Verlag.

[68] Cristina Videira Lopes. AP/S++: Case-study of a MOP for purposes of software evolution. In Kiczales [54], pp. 167–184. Proceedings are available at http://jerry.cs.uiuc.edu/reflection/reflection96/index.html.

[69] Cristina Videira Lopes, Gregor Kiczales, Gail Murphy, and Arthur Lee. Workshop on aspect-oriented programming. In *International Conference on Software Engineering*, Vol. II, pp. 298–299, Kyoto, April 1998.

[70] Cristina Lopes, Kim Mens, Bedir Tekinerdogan, and Gregor Kiczales. Aspect-oriented workshop at ECOOP'97. In *Workshop Reader of the European Conference on Object-Oriented Programming (ECOOP)*, No. 1357 in Lecture Notes in Computer Science, Finland, June 1997.

[71] Pattie Maes. Concepts and experiments in computational reflection. In Meyrowitz [89], pp. 147–155.

[72] Pattie Maes. Issues in computational reflection. In P. Maes and D. Nardi, editors, *Meta-Level Architecture and Reflection*, pp. 21–35. Elsevier Science, North-Holland, 1988.

[73] Mihnea Marinescu and Benjamin Goldberg. Partial evaluation techniques for concurrent programs. In *Proceedings of Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97)* [3], pp. 47–62.

[74] Hidehiko Masuhara, Satoshi Matsuoka, Kenichi Asai, and Akinori Yonezawa. Compiling away the meta-level in object-oriented concurrent reflective languages using partial evaluation. In Loomis [66], pp. 300–315.

[75] Hidehiko Masuhara, Satoshi Matsuoka, and Takuo Watanabe. Design and implementation of an object-oriented concurrent reflective language ABCL/R2. *Computer Software*, Vol. 11, No. 3, pp. 15–32, May 1994. (In Japanese).

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

[77] Hidehiko Masuhara, Satoshi Matsuoka, and Akinori Yonezawa. A compilation technique for parallel reflective language systems using partial evaluation. In *Joint Symposium on Parallel Processing (JSPP)*, pp. 273–280, Fukuoka, May 1995. (In Japanese).

[78] Hidehiko Masuhara, Satoshi Matsuoka, and Akinori Yonezawa. A compilation technique for parallel reflective language systems using partial evaluation. *Transactions of Information Processing Society of Japan*, Vol. 37, No. 7, pp. 1290–1298, July 1996. revised version of [77], in Japanese.

[79] Hidehiko Masuhara, Satoshi Matsuoka, and Akinori Yonezawa. Implementing parallel language constructs using a reflective object-oriented language. In Kiczales [54], pp. 79–91. Proceedings are available at http://jerry.cs.uiuc.edu/reflection/reflection96/index.html.

[80] Hidehiko Masuhara and Akinori Yonezawa. Design and partial evaluation of meta-objects for a concurrent reflective language. In Eric Jul, editor, *European Conference on Object-Orienetd Programming (ECOOP'98)*, Vol. 1445 of *Lecture Notes in Computer Science*, pp. 418–439, Brussels, Belgium, July 1998. Springer-Verlag.

[81] Hidehiko Masuhara and Akinori Yonezawa. Reasoning-conscious meta-object design of a reflective concurrent language. *Computer Software*, Vol. 15, No. 4, pp. 62–66, July 1998. (In Japanese).

[82] Hidehiko Masuhara and Akinori Yonezawa. A reflective approach to support software evolution. In Takuya Katayama, editor, *International Workshop on Priciples of Software Evolution (IWPSE'98)*, pp. 135–139, Kyoto, Japan, April 1998.

[83] Satoshi Matsuoka. The OpenJIT project – opening-up the Java JIT compiler, April 1998.

[84] Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, pp. 231–250, 1991.

[85] F. Mattern. Global quiescence detection based on credit distribution and recovery. *Inf. Proc. Lett.*, Vol. 30, No. 4, pp. 195–200, 1989.

[86] Jeff McAffer. The CodA MOP. In *Proceedings of OOPSLA'93 Workshop on Object-Oriented Reflection and Metalevel Architectures*, Washington, D.C., September 1993.

[87] Jeff McAffer. Meta-level architecture support for distributed objects. In *IWOOS*, 1995. submitted.

[88] Jeff McAffer. Meta-level programming with CodA. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, pp. 190–214, 1995.

[89] Norman Meyrowitz, editor. *Proceedings of Object-Oriented Programming Systems, Languages, and Applications*, Vol. 22(12) of *ACM SIGPLAN Notices*, Orlando, FL, October 1987. ACM, ACM.

[90] Mira Mezini. *Variation-Oriented Programming Beyond Classes and Inheritance*. PhD thesis, Department of Electrical Engineering and Computer Science, University of Siegen, August 1997.

[91] Akihiro Nakaya, K. Yamamoto, and Akinori Yonezawa. RNA secondary structure prediction using highly parallel computers. *Compt. Appl. Biosci.*, Vol. 11, pp. 685–692, 1995.

[92] Hideaki Okamura. *A Study on Multi-Model Reflection Framework in Distributed Environments*. PhD thesis, Department of Computer Science, Keio University, 3-14-1 Hiyoshi, Kohoku-ku, Yokohama, 223, JAPAN, February 1995.

[93] Hideaki Okamura and Yutaka Ishikawa. Object location control using meta-level programming. In Mario Tokoro and Remo Pareschi, editors, *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, Vol. 821 of *Lecture Notes in Computer Science*, pp. 299–319. Springer-Verlag, July 1994.

[94] Hideaki Okamura, Yutaka Ishikawa, and Mario Tokoro. AL-1/D: Distributed programming system with multi-model reflection framework. In Yonezawa and Smith [131], pp. 36–47.

[95] Hideaki Okamura, Yutaka Ishikawa, and Mario Tokoro. Metalevel decomposition in AL-1/D. In Nishio and Yonezawa, editors, *ISOTAS: Proceedings of the International Symposium on Object Technologies for Advanced Software (Lecture Notes in Computer Science 742)*, pp. 110–127. Springer-Verlag, 1993.

[96] Hideaki Okamura, Yutaka Ishikawa, and Mario Tokoro. Object migration on reflective distributed programming system AL-1/D. *Computer Software*, Vol. 11, No. 3, pp. 49–64, May 1994. (In Japanese).

[97] Peter Ørbæk. Pope: An on-line partial evaluator. Technical report, DAIMI?, June 1994. to appear.

[98] Yoshihiro Oyama, Kenjiro Taura, and Akinori Yonezawa. An efficient compilation framework for languages based on a concurrent process calculus. In *Proceedings of Euro-Par '97 Object-Oriented Programming*, Lecture Notes in Computer Science, Passau, Germany, August 1997. Springer-Verlag.

[99] Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI'97)* [2], pp. 109–121.

[100] Xerox PARC Aspect-Oriented Programming Project. Aspect-oriented programming. position paper, 1996.

[101] Ramana Rao. Implementational reflection in Silica. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, pp. 251–266, 1991.

[102] Rational Software. UML semantics. http://www.rational.com/uml/html/semantics/, September 1997.

[103] Luis Rodriguez, Jr. A study on the viability of a production-quality metaobject protocol-based statically parallelizing compiler. In Yonezawa and Smith [131], pp. 107–112.

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

[105] John R. Rose. A minimal metaobject protocol for dynamic dispatch. In *Proceedings of OOPSLA'91 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1991.

[106] Erik Ruf. Partial evaluation in reflective system implementation. In *Proceedings of OOPSLA'93 Workshop on Object-Oriented Reflection and Metalevel Architectures*, Washington, D.C., September 1993.

[107] Erik Ruf. *Topics in Online Partial Evaluation*. PhD thesis, Stanford University, March 1993. (Technical Report CSL-TR-93-563).

[108] Shmuel Safra. Partial evaluation of concurrent prolog and its implications. Technical Report CS86–24, Weizmann Institute of Science, 1986.

[109] Shmuel Safra and Ehud Shapiro. Meta interpreters for real. In Ehud Shapiro, editor, *Concurrent Prolog: Collected Papers*, Vol. 2, chapter 25, pp. 166–179. MIT Press, 1987.

[110] Toshiyuki Shimizu, Takeshi Horie, and Hiroaki Ishihata. Low-latency message communication support for the AP1000. In *The 19th Annual International Symposium on Computer Architecture*, Vol. 20(2), pp. 288–297, Gold Coast, Australia, May 1992.

[111] John W. Simmons II, Stanley Jefferson, and Daniel P. Friedman. Language extension via first-class interpreters. Technical Report Technical Report No. 362, Computing Science Department, Indiana University, Bloomington, Indiana, September 1992.

[112] Brian Cantwell Smith. Reflection and semantics in Lisp. In *Conference record of Symposium on Principles of Programming Languages*, pp. 23–35, 1984.

[113] Randall B. Smith, John Maloney, and David Ungar. The Self-4.0 user interface: Manifesting a system-wide vision of concreteness, uniformity, and flexibility. In Loomis [66], pp. 47–60.

[114] Jonathan M. Sobel and Daniel P. Friedman. An introduction to reflection-oriented programming. In Kiczales [54]. Proceedings are available at http://jerry.cs.uiuc.edu/reflection/reflection96/index.html.

[115] Michael Sperber and Peter Thiemann. Two for the price of one: Composing partial evaluation and compilation. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI'97)* [2], pp. 215–225.

[116] Yuuya Sugita, Hidehiko Masuhara, and Ken'ichi Harada. An efficient implementation of a reflective language using a dynamic code generation technique.

In *Proceedings of the JSSST SIGOOC 1998 Workshop on Systems for Programming and Applications (SPA'98) http://www.brl.ntt.co.jp/ooc/spa98/proceedings/*, Kusatsu, Japan, March 1998. Japan Society for Software Science and Technology (JSSST). (In Japanese).

[117] Yuuya Sugita, Hidehiko Masuhara, Kenichi Harada, and Akinori Yonezawa. On-the-fly specialization of reflective programs using dynamic code generation techniques. In Jean-Charles Fabre and Shigeru Chiba, editors, *OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, Vol. 98–4 of *Technical Report of Center for Computational Physics, University of Tsukuba*, pp. 21–25, Vancouver, B.C., Canada, October 1998.

[118] Michiaki Tatsubori and Shigeru Chiba. A compile-time MOP intuitive for its user. In *Proceedings of the JSSST SIGOOC 1998 Workshop on Systems for Programming and Applications (SPA'98) http://www.brl.ntt.co.jp/ooc/spa98/proceedings/*, Kusatsu, Japan, March 1998. Japan Society for Software Science and Technology (JSSST).

[119] Kenjiro Taura. *Efficient and Reusable Implementation of Fine-Grain Multithreading and Garbage Collection on Distributed-Memory Parallel Computers.* PhD thesis, Department of Information Science, Faculty of Science, University of Tokyo, 1997.

[120] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. An efficient implementation scheme of concurrent object-oriented languages on stock multicomputers. In *Proceedings of Symposium on Principles & Practice of Parallel Programming (PPOPP)*, San Diego, CA, May 1993. ACM SIGPLAN.

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

[122] Kenjiro Taura and Akinori Yonezawa. Schematic: A concurrent object-oriented extension to scheme. In *Proceedings of Workshop on Object-Based Parallel and*

Distributed Computation, No. 1107 in Lecture Notes in Computer Science, pp. 59–82. Springer-Verlag, 1996.

[123] David Ungar and Randall B. Smith. Self: The power of simplicity. In Meyrowitz [89], pp. 227–242.

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

[125] Takuo Watanabe and Akinori Yonezawa. Reflection in an object-oriented concurrent language. In *Proceedings of OOPSLA'88 (SIGPLAN Notices Vol.23, No.11)*, pp. 306–315, San Diego, CA, September 1988. ACM. (revised version in [130]).

[126] Takuo Watanabe and Akinori Yonezawa. An actor-based metalevel architecture for group-wide reflection. In *REX School/Workshop on Foundations of Object-Oriented Languages 1990*, Vol. 489 of *Lecture Notes in Computer Science*, pp. 405–425. Springer-Verlag, 1991.

[127] Daniel Weise, Roland Conybeare, Erik Ruf, and Scott Seligman. Automatic online partial evaluation. In J. Hughes, editor, *FPCA'91 Proceedings of Functional Programming Languages and Computer Architecture (LNCS 523)*, pp. 165–191. Springer-Verlag, August 1991.

[128] Yasuhiko Yokote. The Apertos reflective operating system: The concept and its implementation. In *Proceedings of OOPSLA'92 (SIGPLAN Notices Vol.27, No.10)*, pp. 414–434, Vancouver, B.C., October 1992.

[129] Yasuhiko Yokote, Fumio Teraoka, and Mario Tokoro. A reflective architecture for an object-oriented distributed operating system. In Stephen Cook, editor, *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, pp. 89–106. Cambridge University Press, 1989.

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

[131] Akinori Yonezawa and Brian C. Smith, editors. *Proceedings of International Workshop on New Models for Software Architecture (IMSA92): Reflection and Meta-Level Architecture*, Tama City, Tokyo, November 1992.

Appendix A

Example Compilation of Reflective Programs in ABCL/R3

Here we give a meta-level programming example—the n-Queens problem described in Section 3.2.3.2, and the excerpt of the actual compiled result.

A.1 Base-level Program

The base-level program is an n-Queens parallel search problem. A search node in the search tree is represented by a concurrent object.

```
(define-class queen ())

(define-method queen (do-search self size col places)
  (if (= size col) ; do we reach an answer?
    ;; yes, report the answer
    (print-answer *printer* places)
    ;; no, check if we can place at i'th row of the next column
    (let loop ((i 0))
      (if (< i size)
```

```
(begin
  (if (not (checked? col i places))
    ;; create a new object and have it search in parallel
    (future (do-search (queen) size
              (+ col 1) (cons i places))))
  (loop (+ i 1)))))))
```

A.2 Meta-interpreter for Locality Control

The meta-level programs are divided into two modules; the locality control module and the weighted termination detection module. Firstly, locality control is achieved by the evaluator object `locality-eval`, which specifies the processor numbers of newly created objects. A meta-level argument `depth` is transparently added to inter-object message passing.

```
;;; Class definition.
(define-delegation-class locality-eval ())

;;; The method that gives the processor number for
;;; object creation is overridden.
(define-method locality-eval
               (get-object-creation-node self class arg-vals env)
  ;; look up the meta-level variable depth
  (let ((depth (lookup-meta env 'depth)))
    (if (< *threshold* depth)    ;; compare with the constant threshold
      (this-node-id)             ;; create on the local processor
      (random-node-id))))        ;; create on a remote processor


;;; A hidden parameter depth is passed to a newly created object.  The
;;; following method returns an association list of parameter names and values.
(define-method locality-eval (get-method-invocation-meta-arg
                              self class arg-vals env)
```

```
  ;; look up the meta-level variable depth
  (let ((depth (lookup-meta env 'depth)))
    ;; add a depth parameter to the other meta-level arguments
    ;; generated by the delegate evaluators.
    (cons (cons 'depth (+ depth 1)) ;; (current depth)+1
          (get-method-invocation-meta-arg super class arg-vals env))))
```

A.3 Meta-interpreter for Weighted Termination Detection

The module for the weighted termination detection modules manager objects, evaluator object `WTD-eval`, and several meta-object methods. Here, we only show the evaluator, which (1) calls an initialization method at the beginning of the base-level method, (2) distributes weight to child objects, and (3) calls a finalization method (to return weight) at the termination of the base-level method.

```
;;; Class definition.
(define-class WTD-eval ())

;;; Invoke method init-weight at the beginning of a method.
(define-method WTD-eval (eval-entry self exp env)
  ;; variable ID refers the meta-object
  (init-weight ID)
  ;; body of method execution (by delegation)
  (let ((result (eval-entry super exp env)))
    ;; Invoke the method to return weight at termination of a method.
    (return-remaining-weight ID)
    result))

;;; Parameter weight is passed on to child objects.
(define-method WTD-eval
               (get-method-invocation-meta-arg self class arg-vals env)
```

```
  (cons (cons 'weight (get-weight-for-child ID))
        (get-method-invocation-meta-arg
         super class arg-vals env)))
```

A.4 Specialized Program

We show the resulting compiled code below before being passed into the back-end compiler. Our compilation framework creates a new class `locality-eval*WTD-eval**queen` to hold specialized definitions. Some arguments have been omitted, and some variables have been renamed for readability. Although program size has become slightly larger, interpretation is 'compiled away.' The reasons for increase in program size are: (1) code after a conditional expression has been duplicated, (2) the first iteration of the loop has been unfolded, and (3) different specialized function is constructed for each branch of conditionals, although they have the same definitions. (Note that the compilation is performed by the older version of our system, which based on ABCL/*f*'s syntax. The result of compilation was also generated in the ABCL/*f*'s syntax, but manually changed for uniformity. However, this is not a substantial change to run-time performance.)

```
(define (queen)
  ;; creates an instance of locality-eval*WTD-eval**queen with appropriate
  ;; initial values for its instance variables.
  )

(define-class locality-eval*WTD-eval**queen ()
  class methods state-vars state-values lock evaluator)

(define-method locality-eval*WTD-eval**queen
                        (do-search self size col places)
  (init-weight self)                              ; from WTD-eval
  (if (= size col)
    (let ((result (print-answer *printer* places)))
      (return-weight self)                         ; from WTD-eval
```

```
      result)
    (if (< 0 size)
      (if (not (checked? col 0 places))
        (if (< *threshold* depth)            ; from locality-eval
          (begin
            (future (do-search
                      (now (queen) :on (this-node-id)); local creation
                      size (1+ col) (cons 0 places)
                      (list (cons 'depth (+ depth 1))
                            (cons 'weight
                                  (meta-weight-for-child self)))))
            (eval-apply818 depth col size 1 places)); next step of the loop
          (begin
            (future (do-search
                     (now (queen) :on (random-node-id)); random creation
                     size (1+ col) (cons 0 places)
                     (list (cons 'depth (+ depth 1))
                           (cons 'weight
                                 (meta-weight-for-child self)))))
            (eval-apply819
             depth col size 1 places)))          ; next step of the loop
        (eval-apply820 depth col size 1 places))  ; next step of the loop
      (begin (return-weight self)                 ; from WTD-eval
             #f))))

(define-method locality-eval*WTD-eval**queen
               (eval-apply818 self depth col size row places)
    (if (< row size)
      (if (not (checked? col row places))
        (if (< *threshold* depth)                 ; from locality-eval
          (begin
            (future (do-search
```

```
                    (now (queen) :on (this-node-id)); local creation
                    size (+ col 1) (cons row places)
                    (list (cons 'depth (+ depth 1))
                          (cons 'weight
                                (meta-weight-for-child self)))))
          (eval-apply818
           depth col size (+ row 1) places))    ; next step of the loop
        (begin
          (future (do-search
                  (now (queen) :on (random-node-id)); random creation
                  size (+ col 1) (cons row places)
                  (list (cons 'depth (+ depth 1))
                        (cons 'weight
                              (meta-weight-for-child self)))))
          (eval-apply819
           depth col size (+ row 1) places)))    ; next step of the loop
      (eval-apply820
       depth col size (+ row 1) places))         ; next step of the loop
    (begin (return-weight self)
           0)))
```

;;; Methods `eval-apply819`, `eval-apply820` *have the same definition to*
;;; `eval-apply818`.

Appendix B

Programs Using Guarded Methods

B.1 Base-level Program

A base-level object that uses the guarded method mechanism has an optional form "`(:metaclass ...)`" in the class declaration, and has an expression "`(:guard ...)` " in each guarded method. The following program is the definition of the bounded buffer used in Section 4.2.2.2:

```
(define-class bb () size elements
  (:metaclass guard-meta))

(define-method! bb (put! self item)
  (:guard (< (length elements) size)) ; guard expression
  (become self :elements (append elements (list item))))
```

B.2 Meta-level Program

We define the class `guard-meta`, as a subclass of metaobject, at the meta-level.

```
(define-class guard-meta (metaobject) ; a subclass of metaobject
  (guard (make-guard)))               ; scheduler
```

In the additional instance variable `guard`, each instance of `guard-meta` has a scheduler, which is a user-defined meta-level object. We also override the following two methods of `guard-meta`:

```
(define-method guard-meta (receive self mes &reply-to mresult)
  (let* ((selector (message-selector mes))
         (method (find-method methods selector))
         (guard-exp (cdr (method-find-option method ':guard))))
    (register guard
              (lambda ()
                (let* ((env (make-env self (formals method) mes))
                       (result (eval evaluator guard-exp env)))
                  (if result
                      (reply (accept-W self mes) mresult))
                  result))))) ; result of guard expression

(define-method guard-meta (accept-W self mes)
  (let ((r (make-channel)))
    (let ((result (accept self mes r)))
      (update self (touch r))
      (notify guard)
      result))) ; result of method body
```

The method `receive` registers a closure to `guard`. The closure, when activated by the scheduler, evaluates a guard expression and then invokes `accept-W` if the guard expression returns true. The method `accept-W`, evaluates the method body, as `accept-W` of the class `metaobject` does, and also notifies `guard` at the end of the evaluation.

B.3 Optimized Program

From the base-level and the meta-level programs, our optimization framework generates the following combined program. The meta-level operations for guarded meth-

ods, which are defined in the methods `receive` and `accept-W` of `guard-meta`, are embedded in the method `put!` of the optimized class.

```
(define-class guard-meta**bb ()
  class methods state-vars state-values lock evaluator
  (guard (make-guard)))

(define-method guard-meta**bb (put! self item &reply-to mresult0)
  (let ((c0 (lambda ()
              ;; evaluation of guard expression
              (let* ((values0   (read-cell state-values))
                     (size0     (vector-ref values0 0))
                     (elements0 (vector-ref values0 1))
                     (result0   (< (length elements0) size0)))
                (if result0
                  ;; execution of method body
                  (let* ((state-update-ch0 (make-channel))
                         (values1   (read-cell state-values))
                         (size1     (vector-ref values1 0))
                         (elements1 (vector-ref values1 1)))
                    (reply (vector size1
                                   (append elements1 (list item)))
                           state-update-ch0)
                    (let ((new-state0 (touch state-update-ch0)))
                      (update-cell! state-values new-state0)
                      (notify guard)
                      (reply self mresult0))) ; result of method body
                  #f)
                result0)))) ; result of guard expression
    (register guard c0)))
```

B.4 Nonreflective Program

Instead of using customized meta-objects, we can manually rewrite programs that have the same functionality to the ones using guarded methods. One of the simplest approach is to split each guarded into three actual methods: an entry method, a guard method, and a body method. The following definitions are a manually rewritten bounded buffer:

```
(define-class bb ()     ; nonreflective version
  size elements (guard (make-guard))

(define-method bb (put! self item &reply-to r)
  (let ((c (lambda ()
             (let ((guard-result (put!-guard self item)))
               (if guard-result
                   (reply (put!-body self item) r))
               guard-result))))
    (register guard c)))

(define-method bb (put!-guard self item)
  (< (length elements) size))           ; guard expression

(define-method! bb (put!-body self item)
  (become (begin (notify guard)         ; notification
                 self)
          :elements (append elements (list item))))
```

The class definition has an additional instance variable `guard` for the scheduler. The method `put!` is an entry method that creates and registers a closure to the scheduler. The method `put!-guard` is the guard method, and `put!-body` is the body method. They are invoked from the closure created in `put!`.