TECHNICAL REPORT 94-7e

A Concurrent Object-Oriented Programming Language System for Highly Parallel Data-Driven Computers and its Applications

Masahiro Yasugi

April 5, 1994

Department of Information Science
Faculty of Science, University of Tokyo

7–3–1 Hongo, Bunkyo-Ku Tokyo, 113 Japan

TECHNICAL REPORT 94-7e

TITLE

A Concurrent Object-Oriented Programming Language System for Highly Parallel Data-Driven Computers and its Applications

AUTHORS

Masahiro Yasugi

KEY WORDS AND PHRASES

ABCL, object-oriented, concurrent object, language implementation, look-up table, subtyping, intermediate code, N-body problem, EM-4, highly-parallel, data-driven.

ABSTRACT

With recent developments of highly parallel computers, practical methods for developing programs on such machines become urgent. In this study, we regard our practical concurrent object-oriented programming language as a basic language for highly parallel machines and pursue necessary software approaches to achieving high performance by implementing the language system and developing application programs/parallel algorithms. As for the language system, we propose a language design suitable for parallel execution and implementation schemes that achieve high performance. Our implementation target is a fine-grained data-driven parallel computer, EM-4, developed at Electrotechnical Laboratories. We present and discuss the design of our runtime architecture and optimizing compiler, including reactive representations of objects, efficient management of message buffers with freelists, interruptible pipelined message sends, plan-do style intermediate code, an efficient implementation scheme for table look-up, and advanced register/storage allocation. As for application programs/parallel algorithms, we present concurrent object-oriented algorithms for highly parallel computers, (e.g., N-body problem) and evaluate them. We have developed a prototype compiler and a runtime system which run the N-body problem program on the real machine, and measured its performance. Performance measurements on EM-4 show that the cost of each remote object-creation and remote message-passing can be, in order, comparable to that of a sequential procedure call and the N-body algorithm based on the concurrent object-oriented models can be executed efficiently for a wide range of N. Our results indicate that the object-oriented concurrent computation models and languages are extremely suitable for realizing practical parallel systems.

REPORT DATE	WRITTEN LANGUAGE
April 5, 1994	English
TOTAL NO. OF PAGES	**NO. OF REFERENCES**
124	37

ANY OTHER IDENTIFYING INFORMATION OF THIS REPORT

This report is the author's thesis submitted to the graduate school of the University of Tokyo in partial fulfillment of the requirements for the degree of doctor of science in information science.

DISTRIBUTION STATEMENT

This technical report is available ONLY via anonymous FTP from `ftp.is.s.u-tokyo.ac.jp` (directory `/pub/tech-reports`).

SUPPLEMENTARY NOTES

DEPARTMENT OF INFORMATION SCIENCE
Faculty of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku Tokyo, 113 Japan

A CONCURRENT OBJECT-ORIENTED PROGRAMMING LANGUAGE SYSTEM FOR HIGHLY PARALLEL DATA-DRIVEN COMPUTERS AND ITS APPLICATIONS

高並列データ駆動計算機における
並列オブジェクト指向プログラミング言語処理系とその応用プログラム

by
Masahiro Yasugi
八杉 昌宏

A Thesis

Submitted to

The Graduate School of
The University of Tokyo
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Science
in Information Science

March 1994

ABSTRACT

With recent developments of highly parallel computers, practical methods for developing programs on such machines become urgent. In this study, we regard our practical concurrent object-oriented programming language as a basic language for highly parallel machines and pursue necessary software approaches to achieving high performance by implementing the language system and developing application programs/parallel algorithms. As for the language system, we propose a language design suitable for parallel execution and implementation schemes that achieve high performance. Our implementation target is a fine-grained data-driven parallel computer, EM-4, developed at Electrotechnical Laboratories. We present and discuss the design of our runtime architecture and optimizing compiler, including reactive representations of objects, efficient management of message buffers with freelists, interruptible pipelined message sends, plan-do style intermediate code, an efficient implementation scheme for table look-up, and advanced register/storage allocation. As for application programs/parallel algorithms, we present concurrent object-oriented algorithms for highly parallel computers, (e.g., N-body problem) and evaluate them. We have developed a prototype compiler and a runtime system which run the N-body problem program on the real machine, and measured its performance. Performance measurements on EM-4 show that the cost of each remote object-creation and remote message-passing can be, in order, comparable to that of a sequential procedure call and the N-body algorithm based on the concurrent object-oriented models can be executed efficiently for a wide range of N. Our results indicate that the object-oriented concurrent computation models and languages are extremely suitable for realizing practical parallel systems.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my greatest supervisor, Professor Akinori Yonezawa, for his invaluable supports and directing me to this interesting research area. I would like to thank Dr. Satoshi Matsuoka for his many insightful comments and supports, which really help me.

I am grateful to Drs. Yoshinori Yamaguchi, Mitsuhisa Sato, Yuetsu Kodama at ETL and Dr. Shuichi Sakai at RWCP for allowing us to use EM-4 and sparing time for technical discussions.

I am also thankful to Dr. Rishiyur S. Nikhil at DEC CRL and Professor Kei Hiraki at the Univ. of Tokyo; the ideas of using ideal gains on message passing implementation schemes are inspired by discussions with them.

Finally, I thank all the former/current members of Yonezawa Laboratory and the programming language group: Kenjiro Taura for daily heated and joyful discussions on a wide range of research topics, Dr. Yuuji Ichisugi at ETL for his advice and repeatedly answering my questions, Ken Wakita at Tokyo Institute of Technology for his open seminars on programming languages and implementations, Hidehiko Masuhara for useful discussions and comments, and Tomio Kamada for his research contributions on concurrent garbage collection. I also thank Naoki Kobayashi, Jacques Garrigue, Jeff McAffer, Kenichi Asai, Masaaki Nagatsuka, Tatsurou Sekiguchi, Kazuhiro Konno and the other members for their helpful comments.

The author was supported in part by a Fellowship of the Japan Society for the Promotion of Science for Japanese Junior Scientists.

Chapter 1

Introduction

1.1 Motivations

With recent developments of highly parallel computers, practical methods for developing programs on such machines become urgent. In this study, we regard our Concurrent Object-Oriented Programming (COOP) language as a basic language for highly parallel machines and pursue necessary software approaches to achieving high performance by implementing the language system and developing application programs/parallel algorithms.

Our implementation target is a fine-grained data-driven highly-parallel distributed-memory computer, EM-4, developed by Electrotechnical Laboratories in Tsukuba, Japan. This machine has many attractive features for implementing our concurrent Object-Orient (OO) computing system.

1.2 Contributions

The contributions of this thesis research are as follows:

Language and Type for Efficient Parallel Execution
A novel language design for efficient parallel execution is presented, which introduces classes and static types with subtype relations. In this language, a message is a first class datum (which has a type), and an object has a type as the message target. Consequently, request messages and reply messages are well integrated, and types are well supplied to both past type (namely, only request) and now type (namely, request followed by reply) message passing forms. Increasing static entities (e.g., classes and types) improves performance of parallel execution especially in distributed memory environments.

Small Look-up Table
An efficient implementation scheme of look-up table is developed. The look-up tables are used for pattern matching (by case expression) of data with injection

tags. The size of the table can be bounded by a number slightly more than the number of injection tags which the tested value may have, and the look-up can be performed as direct indexing of the table. This efficient implementation is realized by using explicit subtype relations between disjoint-union types in the type system.

Design Principle for Robust Runtime Architecture

On selecting implementation schemes, we introduce a notion of *ideal gain* as a design criterion, which is an improvement gain on the total execution time in an ideal environment where the system has a sufficient number of tasks for latency hiding and a local improvement on system resource consumption appears as an improvement on the execution time. Implementation schemes following this criterion do not degenerate its performance under high system load, and on average they show good performance even under low system load.

Pipelined Sends — Improving Throughput and Latency by Overlapping Computation and Communication

A (remote) message passing implementation scheme which overlaps computation and communication, a pipelined send, is proposed. This scheme overlaps the evaluation of message arguments and their send in a pipelined way; it improves both throughput and latency, by eliminating local buffering of messages on both sender and receiver nodes. This scheme is originally introduced by the ideal gain notion, but this scheme also works to reduce the latency of message communication.

Plan-Do Style Code for Advanced Compilation

Novel and simple plan-do style intermediate code in which an action (e.g., a request message send) is represented by two separated parts, namely, the plan declaration and the plan execution is proposed. Using the plan-do style code is an essential compilation technique to realize the advanced implementation schemes (e.g., the pipelined sends); future information provided by the plan declaration enables the early execution of the planned action (e.g., eager data transfer).

Advanced Register/Storage Allocation

Compilation techniques for efficient register/storage allocation are proposed:

1. A *least immediately used* algorithm is used for register allocation. This algorithm is not new, but we found that this algorithm is especially suitable for volatile registers of EM-4.
2. Registers are allocated not only to temporary variables but also to state (instance) variables of objects and pattern variables of messages to speed up the access to those variables. This allocation is fully benefited by a primary object-oriented concept, *encapsulation*.

A Fast Parallel N-body Algorithm

A fast parallel object-oriented numerical algorithm which computes the positions and velocities of a large number (N) of particles (masses) interacting via a long range force is developed. The time-complexity of the N-body algorithm is $O(\log N)$ using $O(N)$ concurrent objects. The algorithm is a parallel extension of the Barnes and Hut algorithm[4]. The performance measurement on the real parallel computer EM-4 shows that this algorithm can be executed efficiently for a wide range of N by exploiting the concurrent OO models.

These technical contributions do not work alone; the effects are multiplied. The combination of all these has demonstrated to achieve the high performance we desired.

1.3 Overview of Thesis Research

This section describes the overview of this thesis.

Chapter 2 describes the background of this thesis research. Concurrent object-oriented computation models, such as Actors[1] and ABCM[22], provide formal foundations of parallel computing systems, where the unit of concurrency is a concurrent object, and the interaction between concurrent objects is performed solely via message passings, resulting in a strong expressive power and natural derivation of MIMD concurrency. The target machine of our implementation is a fine-grained data-driven highly-parallel computer, EM-4, designed and built at Electrotechnical Laboratories, whose hardware is a hybrid architecture of a reactive packet(data)-driven architecture and conventional control-flow architecture, and it performs RISC-like execution in the control-flow mode. As for related work, there have been a series of implementations of concurrent OO system; however, most previous implementations (with few exceptions) are inefficient because, for example, they use processes or threads provided by operation systems as concurrent objects.

Chapter 3 outlines our language system ABCL/EM-4. Our approach to the language system development is similar to that of the current RISC technology — the runtime architecture (the runtime software/hardware combination) for the simple and fast hardware is designed using (1) design criteria (e.g., ideal gain) and (2) the result of preliminary measurements, then the language and the compiler are developed to support the runtime architecture. We reduced the cost of remote object-creation and remote message-passing to the degree comparable to that of sequential procedure invocation.

In Chapter 4, we propose a concurrent object-oriented programming (COOP) language, ABCL/ST, which is the source language of ABCL/EM-4. This language is designed suitable for parallel execution; it introduces classes as templates of objects and static typing with subtype relations for the compiler to generate concurrently executable efficient code.

Chapter 5 presents the type system for ABCL/ST, where a message has a type as a first class datum and the type of (a reference to) an object accepting a message is represented with the type of the message. The type system incorporates explicit specifications of subtype relations between disjoint-union types, enabling an efficient implementation scheme for table look-up.

In Chapter 6, we describe our runtime architecture for EM-4, where a concurrent object is represented as a reactive entity with multiple entry points for packets, exploiting the reactive nature of EM-4. The (remote) message passings are *pipelined sends*, which overlap the evaluation of message arguments and their send in a pipelined manner: (1) The sender node reserves a message box on the receiver node, by invoking a proper code block to return the allocated message box address at the receiver node. (2) The sender evaluates the arguments of the message. As each argument is evaluated, a packet containing the result is sent to the proper (address) location in the message box on the receiver node. The evaluation and the message send are pipelined. (3) The message box address is sent to the receiver object. The receiver may enqueue the message until it can process the message.

Chapter 7 presents the performance measurement and discussions on the basic operations such as remote message sends. Performance measurements on EM-4 show that the latency for a remote object-creation followed by a request message send and the corresponding reply reception (called now type message passing) is approximately 10.75 μs (for a 12.5 MHz clock speed), and the throughput of the entire system (80 PEs) for tasks of a remote object-creation followed by a now-type message passing is more than 3000 tasks per 1 ms. Although the packet transmissions to reserve a message box may result in a round-trip latency and some overhead, multi-threading at the sender node could hide the latency. Moreover, the reservation provides us the following significant advantages: (1) It allows us to manage the message queue efficiently at the receiver node, just by pointer manipulation rather than copying all contents of the message. (2) The knowledge of the target message box address makes a pipelined send possible, which reduces (2a) the communication latency by eager data transfer before the completing the evaluation of the message and (2b) the local memory access for local buffering at the sender node.

In Chapter 8, the compiler for EM-4, which realizes the pipelined sends described in Chapter 6, is described. A (high-level) plan-do style intermediate language is introduced, where a plan such as a request message send is declared before the actual execution; the lower level part of the compiler interprets the plan-do style intermediate code to generate the lower-level intermediate code, where the future information (e.g., which values will be used for the plan execution) provided by plan declarations are used, for example, to allocate the message boxes prior to the execution. Other compiler techniques include a register allocation scheme which allocates registers to encapsulated state variables and pattern variables as well as temporary variables.

In Chapter 9, we present a new fast concurrent OO algorithm for N-body problem. The time-complexity of the algorithm is $O(\log N)$ using $O(N)$ concurrent objects. The algorithm is a parallel extension of the Barnes and Hut algorithm[4]. We propose several techniques to avoid bottleneck around the root of the tree structure representing spatial distribution of particles.

Chapter 10 presents the results of performance measurements for the N-body algorithm described in Chapter 9 on the ABCL/EM-4 system. The program uses the whole 80 PEs. It can be executed efficiently for a wide range of N; when N=1040, it creates more than 60,000 objects and exchanges more than 420,000 messages, and the elapsed time is less than 0.5s.

Finally, we will conclude in Chapter 11.

Chapter 2

Background

2.1 Concurrent Object-Oriented Computation Models

The trend toward object-oriented (OO) software construction is becoming more and more prevalent. Important software concepts such as *encapsulation* promote a high degree of code re-use and clean architectural structuring of large software. High-performance parallel programming, previously performed in the context of more conventional programming languages, would also be able to enjoy the benefit from the OO technology with appropriate OO languages and systems.

Although many OO languages currently in use today (such as C++ and Smalltalk) are sequential, it is more natural to consider objects as being a basic unit of concurrency in the context of parallel computation. There, the computation is modeled as a parallel execution of autonomous, concurrently active objects and message passings between them. A recent breed of COOP (*Concurrent Object-Oriented Programming*) languages attempt to provide maximum computational and modeling power through concurrency of objects. Recent work has also been successful in establishing strong, theoretical foundations for concurrent objects; they include series of work in Actors[1], ABCM[22], POOL[2], concurrent rewriting logic[18], and calculi of asynchronous objects[10].

2.1.1 Computation Models for ABCL

In our computation/programming model[33, 32], computation is performed by a collection of autonomous, concurrently active software modules called *concurrent objects*, and the interaction between concurrent objects is performed solely via message passings. More than one concurrent object can become active simultaneously, and more than one message transmission may take place in parallel.

A concurrent object is the basic unit of concurrency. Each has its own (autonomous) single thread of control, and it may have its own local persistent memory, the contents of which can be accessed only by itself[1] (*encapsulation*). Concurrent objects can be dynamically created at the runtime (*dynamic creation*).

There are two kinds of messages, namely, *request messages* and *reply messages*. The transmissions of request messages are asynchronous in the sense that the sender can send a request message to a receiver (both are, of course, concurrent objects) at any time it wishes to. The request messages arriving at a concurrent object are buffered in a message queue associated with the concurrent object until they are processed (*asynchronous buffered communication*). When two request messages are sent from the same sender to the same receiver, they arrive at the receiver in the same order as they were sent (*preservation of transmission order*). On the other hand, a reply message is received by a synchronizer, *reply destination*, which waits for both a reply message and a continuation of a concurrent object and makes the concurrent object resume its execution from the continuation with the reply.

A sender can send a message to any (other) concurrent object as long as the sender knows its name. Since a message may contain names (IDs, or global addresses) of concurrent objects as well as basic values such as numbers and booleans, and concurrent objects can be dynamically created, the connection topology of concurrent objects is dynamic.

The definition of concurrent object's behavior must specify how its local persistent memory is represented, and how it executes its code with its own thread of control. In our programming language, the local persistent memory is represented by a collection of variables, called *state variables*, and the *modes* of its execution are categorized into three modes, namely, *active*, *waiting*, and *dormant*. In the active mode, a concurrent object can execute a sequence of the following four kinds of basic actions:

1. *Message sends* to (other) concurrent objects or reply destinations.

2. *Creation* of concurrent objects.

3. *Referencing and Updating* of the contents of its local memory (*imperative computation*).

4. *Various operations* (such as arithmetic operations) on values that are stored in its local memory and passed around by message transmissions.

Message sends could be *past type* (namely, "asynchronously send and no-wait", syntactically denoted by:

$$[\,Target \ \ \texttt{<=}\ \ Msg\,])$$

or *now type* (namely, "asynchronously send and wait for a reply", syntactically denoted by:

$$[\,Target \ \ \texttt{<==}\ \ Msg\,])$$

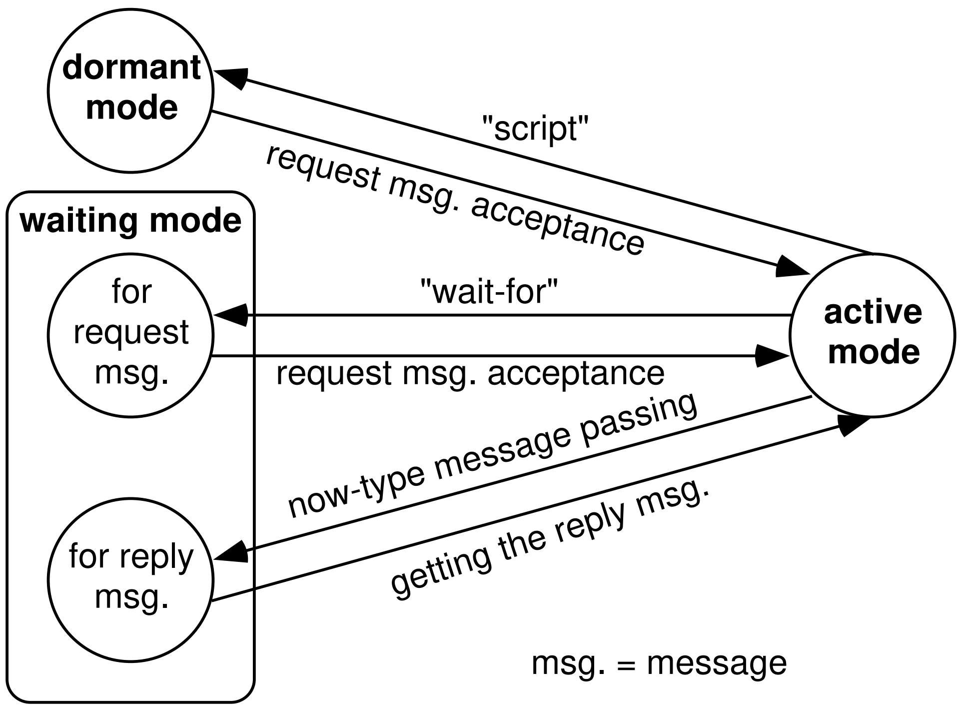


Figure 2.1: Execution Mode Transition

A now type message passing attaches a new reply destination to the message, and changes the execution mode from *active* to *waiting* until a reply message is sent back to the reply destination.

In order to accept a request message, a concurrent object changes its execution mode to *dormant* or *waiting* (See Figure 2.1). The description/code of each sequence of actions to be performed when a message is accepted is called a *method*. Determining which method is executed is called *method look-up*.

In the dormant mode, a concurrent object checks all the messages in its message queue from the head and deletes all the preceding messages until it finds an acceptable message, and it becomes *active* and processes the acceptable message (namely, deleting the message and executing the corresponding method). In the waiting mode, a concurrent object also checks its message queue from the head and skips all the preceding messages (without deleting them) until it finds an acceptable message, and it processes the acceptable message. Whether or not a message is acceptable is determined by matching the message with *message patterns*, and checking the condition on the contents of the message and the current state of the concurrent object.

A concurrent object changes its mode from active to waiting when it performs a special *wait-for* command ("wait-for" transition in Figure 2.1). In the original ABCL[33, 32], (1) when a concurrent object is created, or (2) immediately after every execution of one of its method, it becomes dormant ("script" transition in Figure 2.1). (At this point, the language ABCL/ST proposed in this thesis adopts a slightly different model. The details will be described in Chapter 4.)

Our family of languages, ABCL, is based on this computation model, and have been well studied from theoretical aspects to applicational ones[33, 32]. Applications include parallel discrete event simulation[23], symbolic computation[36], and numerical computation[29, 30].

2.2 A Highly Parallel Data-Driven Computer EM-4

Our base hardware architecture is EM-4, which was designed and built by Electro-Technical Laboratory (ETL) in Tsukuba, Japan. The EM-4 prototype machine consists of 80 (expandable to 1024) packet-driven processing elements (PEs). The PEs are connected with each other via a fast Omega network, and run at 12.5MHz clock speed. This section summarizes EM-4's features essential to our software/hardware architecture for a COOP language. More details of EM-4 can be found in [19, 16].

EM-4 was originally designed to support the *strongly connected arc dataflow model*, which extends the dataflow model by introducing a new type of arcs called *strongly connected arcs* into dataflow graphs. A dataflow subgraph whose nodes are connected by strongly connected arcs is called the *strongly connected block (SCB)*. The nodes in an SCB are executed prior to the nodes not in the SCB. The dataflow synchronization/matching is only applied to the initial node in an SCB.

[1] It is possible for a concurrent object to be *pure*, i.e., not to have such local memory if not needed.

Architecturally, EM-4 is a combination of a reactive, packet-driven dataflow architecture and a conventional register-based control-flow architecture. A packet received by a PE (processing element) is first stored in a hardware queue buffer. The PE takes a single packet from the queue, and performs a simple but fast dataflow matching by the use of the *direct matching scheme (see below)*. If the match occurs, it triggers the execution of an SCB on the PE. The execution of SCB is performed with efficient register-based pipeline execution, and is non-interruptible by subsequent packets until the execution is completed. Imperative operations such as memory writes are also possible, although not explicitly exploited in the strongly connected arc dataflow model. From a different perspective, one could regard a PE in EM-4 as a conventional RISC augmented with packet-driven and packet-output features.

The instruction sequence corresponding to an SCB is stored in a segment of memory called a *template segment*. (A single template segment can contain multiple instruction sequences.) In order to execute the instruction sequence corresponding to an SCB, a segment of memory called the *operand segment* is allocated, and the address of the template segment containing the instruction sequence is stored in the first word of the operand segment.

As mentioned above, matching is performed with a new scheme called the *direct matching scheme*, which is the most characteristic feature of EM-4. we briefly introduce the structure of packets for its explanation. A (normal) packet contains the following data (Figure 2.2(a)):

1. an operand segment number (*OSN*)

2. a displacement (*DPL*) from the top of an operand segment,

3. an operand data for the first instruction of an SCB.

Suppose that the first instruction of an SCB requires two operand data. When the first packet containing an operand data arrives, the operand data is stored in the matching memory whose address is obtained by adding DPL to the address of the top of operand segment obtained from OSN. When another packet containing the second operand data with the same OSN and DPL arrives subsequently, the instruction address to begin the execution of SCB is obtained by adding the DPL value to the address of the top of the template segment, which is pointed to the top word of the operand segment. Then the execution of the SCB is triggered from that address, with the two data operands of the two packets received (Figure 2.2(a)). If the first instruction of the SCB requires only one operand data, the execution of the SCB starts upon the arrival of the first packet without any matching. In both cases, the address of the operand segment is set into a special register called *operand segment register*.

In addition to the normal type of packets described above, there are more than ten types of special packets. The purpose of special packets is to invoke SCBs which do not need dynamic linkage of operand segments to template segments. A special packet specifies a pre-defined SCB, used for common system tasks such as allocating memory

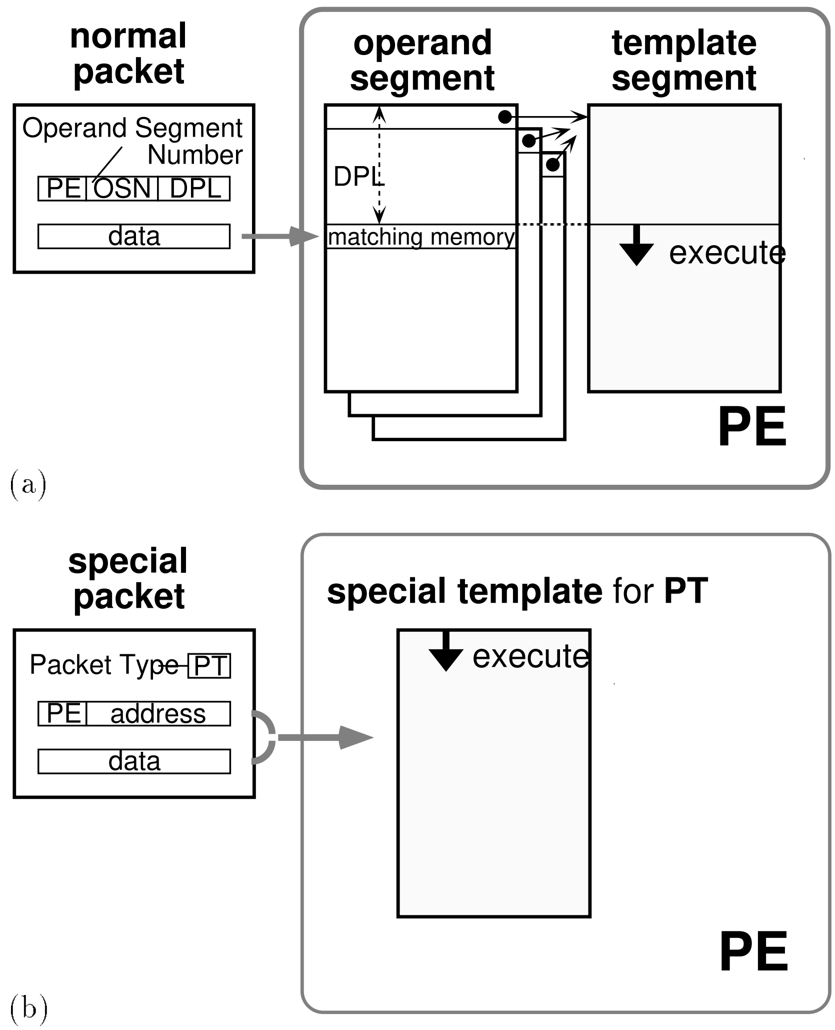


Figure 2.2: Handling a Packet

blocks in a remote PE (Figure 2.2(b)). In addition, a special packet contains two data items, an address data and an operand data, whereas a normal packet can only contain a single operand data. Thus special packets are also used for remote-read/remote-write.

The instruction set of EM-4 consists of arithmetic and logical instructions, test and branch instructions, data transfer instructions and communication instructions. Arithmetic, logical and communication instructions can output a packet into the network directly from the processor.

EM-4 also features a very fast Omega network with support for deadlock avoidance, load balancing, and preservation of transmission ordering. Packet transmission to the adjacent node takes only two clock cycles, and each additional hop takes one clock cycle.

We summarize the architectural features for the later discussion:

- The packet size is fixed to 2 words.
- An execution of an SCB is triggered by a single packet or two packets matching.
- In order to receive a packet, the PE must exit from the current SCB, which causes context switch in the sense of sequential execution.
- Upon context switch, live values on registers should be saved by software.
- Special packets are provided for remote-read/remote-write.
- The network guarantees the point-to-point FIFO-ness.

2.3 Related Work

There is a series of previous work on implementation of concurrent object-oriented languages on parallel computers; however, the most of them use processes or threads provided by operating systems to implement concurrent object, resulting in a serious overhead, which almost made people to believe that concurrent object-orient computing was quite slow.

There are few exceptions which are recently developed sharing the same goal to achieve realistic performance on several parallel architectures:

- Concurrent Smalltalk on J-Machine[12, 11, 6],
- ABCL/onAP1000[24, 35].

The detailed comparisons with these implementations are described in Section 7.4.

Chapter 3

Overview of Concurrent Object-Oriented Programming Language System ABCL/EM-4

This chapter outlines our concurrent object-oriented language system, ABCL/EM-4, details of which are described in the subsequent Chapters 4, 5, 6, 7, and 8, which include the language design, the type system for the language, the runtime architecture for EM-4 and the compiler for EM-4.

3.1 Goals

We pursue various software approaches to achieve very high performance by implementing the language system.

The serious problems of conventional approaches to concurrent OO language system implementation on distributed memory machines are:

- High cost of inter-note communication, and
- High cost of intra-node scheduling.

Thus our implementation goals include:

- High-speed remote message passings comparable to sequential procedure calls, and
- Low-overhead intra-node scheduling.

The following section describes our approach to these implementation goals.

3.2 Our Approach

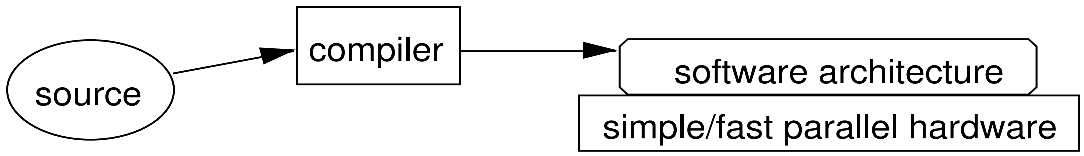


Figure 3.1: Our Approach to Language System Implementation

Our approach to the language system implementation is the use of the combination of sophisticated software (i.e., the efficient runtime software architecture generated by the optimizing compiler) and a simple and fast hardware architecture (Figure 3.1). We do not employ OS's supervisor calls or library routine calls to realize the inter-node communication and the scheduling of concurrent objects; in our ABCL/EM-4 language system, they are implemented by the compiler-generated runtime software architecture. The runtime software architecture is NOT the so-called "runtime system". (This word is sometimes used to represent the fixed part of runtime software components.) "Runtime software architecture" represents all of runtime software components; in ABCL/EM-4, the runtime software architecture changes its form in response to the source program, in order to dramatically improve its performance.

Our approach to the language system development is similar to that of the current RISC technology — the runtime architecture (the runtime software/hardware combination) for the simple and fast hardware is designed using (1) design criteria (e.g., ideal gain) and (2) the result of preliminary measurements, then the language and the optimizing compiler are developed to support the runtime architecture.

We also designed the programming language for ABCL/EM-4, which has high-level features and clear semantics, and also employs *static typing* for the compiler to generate efficient code.

3.3 The Programming Language

A language design suitable for parallel execution is presented in Chapter 4, and the type system is presented in Chapter 5.

Our programming language is a concurrent object-oriented programming language, ABCL/ST, which is a descendant of ABCL/1[33, 32]. ABCL/ST introduces classes and static types with subtype relations. In this language, a message is a first class datum (which has a type), and an object has a type as the message target. Consequently, request messages and reply messages are well integrated, and types are well supplied to both past type (namely, only request) and now type (namely, request followed by reply) message passing forms in ABCL. Increasing static entities (e.g., classes and types) improves performance of parallel execution especially in distributed memory environments.

The type system incorporates explicit specifications of subtype relations between disjoint-union types, enabling an efficient implementation scheme for table look-up.

3.4 The Runtime Architecture

The runtime architecture is a combination of runtime software/hardware architectures. The design of runtime architecture which is the essential part of our high performance language system is presented in Chapters 6,7.

In our runtime architecture for EM-4, a concurrent object is represented as a reactive entity with multiple entry points for packets, exploiting the reactive nature of EM-4. The (remote) message passings are *pipelined sends*, which overlap the evaluation of the message arguments and their send in a pipelined manner: (1) The sender node reserves a message box on the receiver node, by invoking a proper code block to return the allocated message box address at the receiver node. (2) The sender evaluates the arguments of the message. As each argument is evaluated, its packet is sent to the proper (address) location in the message box on the receiver node. The evaluation and the message send are pipelined. (3) Finally, the message box address is sent to the receiver object to indicate the completion of the message send. The receiver may enqueue the message until it can process the message.

Performance measurements on EM-4 show that the latency for a remote object-creation followed by a request message send and the corresponding reply reception (called now type message passing) is approximately 10.75 μs (for a 12.5 MHz clock speed), and the throughput of the entire system (80 PEs) for tasks of a remote object-creation followed by a now-type message passing is more than 3000 tasks per 1 ms. We reduce the cost of remote object-creation and remote message-passing to the degree comparable to that of sequential procedure invocation, and the cost of intra-node scheduling is quite low compared to the essential part of the computation.

3.5 The Compiler

The compilation techniques realizing our high performance runtime architecture are presented in Chapter 8.

The compiler introduces several language layers to overcome the significant difference of the abstraction levels between its input (a concurrent OO program) and output (assembly code); each lower level language implements the functionalities of its higher level language.

In order to realizes the pipelined sends, a (high-level) plan-do style intermediate language is introduced, where a plan such as a request message send is declared before the actual execution; the lower level part of the compiler interprets the plan-do style intermediate code to generate the lower-level intermediate code, where the future information (e.g., which values will be used for the plan execution) provided by plan declarations are used, for example, to allocate the message boxes prior to the execution.

Our compiler provides an efficient implementation scheme of look-up table for pattern matching of data with injection tags. The size of the table can be bounded by a number slightly more than the number of injection tags which the tested value may have, and the look-up can be performed as direct indexing of the table. This efficient implementation is enabled by using explicit subtype relations between disjoint-union types in the type system.

The storage/register allocation schemes include: (1) the read-only pattern variables use their original positions as the home positions (the memory locations for register spilling) to minimize context switch overhead, (2) a least immediately used algorithm is used to allocate registers, and (3) registers are allocated to encapsulated state (instance) variables as well as temporary variables.

3.6 The Current Status of the Language System

The prototype compiler has been developed, which is approximately 10,000 lines Common Lisp code. The runtime system for the compiler consists of approximately 500 lines EM-4 assembly code for packet handlers and approximately 40 lines EM-C code[21] for start up. The language system can run the N-body program (Appendix A) based on a fast parallel object-oriented algorithm proposed in Chapter 9. The results of the evaluation are described in Chapter 10.

Chapter 4

Programming Language ABCL/ST

ABCL/ST (ABCL/Statically Typed) is a concurrent object-oriented programming language for the ABCL/EM-4 language system. ABCL/ST is designed for EM-4 as its primary target machine, but the language itself is machine independent. ABCL/ST is based on the concurrent OO computation model described in Section 2.1.1.

The ancestor of ABCL/ST is ABCL/1[33, 32]. ABCL/ST introduces the following new features:

- static typing
- subtyping
- classes

In this chapter, we present the design principles and a brief introduction of ABCL/ST. The type system for ABCL/ST is described in Chapter 5.

4.1 Design Principles

The design principles of ABCL/ST are summarized as follows:

High efficiency is the primary goal of parallel computing. This is the reason why we want to develop parallel computers. The EM-4 is a distributed-memory machine. Thus high efficiency must be achieved without assuming the presence of shared memories. Based on this design criterion, we chose the following features:

compilation approach : We prefer compiler-based execution to achieve high performance. The language should be designed as input to the compiler rather than an interpreter.

introduction of class : Classes as templates of objects can be used for efficient object creation on distributed memory environments.

static typing : Elimination of runtime type checking is crucial for efficiency. The type system also enables the use of small look-up tables to implement pattern matchings; the details are described in Section 8.2.

Safety is a desirable property to protect the programmer from incomprehensible errors. Safety includes the following meanings:

no system crash : No dangerous memory operations such as `free` are permitted; Garbage Collection (GC) is automatically performed by the system.

static typing with subtyping : No runtime type errors such as `message not understood` occur.

More concrete design principles of ABCL/ST are summarized as follows:

- ABCL/ST is an imperative language, but *no aliasing of variables* is permitted. This is to make the language semantics clear and to increase opportunities to allocate registers to the variables.
- At the source language level, ABCL/ST provides a remote/local transparent view of data location to the language semantics and the programmer:
  - Sharing of mutable objects is realized only with *names (IDs, or global addresses) of concurrent objects.*
  - A structural datum, such as a list, is just a single value not the list of mutable cells with pointers. At the implementation level, it would be represented in a more low-level manner.
  - Parameter passing, such as message passing, is "pass-by-value". At the implementation level, more elaborated schemes would be used to realize the semantics.
  - Load balancing is performed by the system at compile-time or runtime. The compiler should automatically extract compile-time information for load balancing from the source programs; using classes for object creations would help it by providing characteristics of objects.

4.2 Classes and Objects

A *class* is a template of *objects*. A *class* is used (1) as a type of objects, and (2) for dynamic creation of objects, as usual.

4.2.1 Class Definition

A *class definition* specifies behavior of objects of the class. In ABCL/ST, a *class definition* is written in the following form:

```
[class ⟨class name⟩ ⟨interface type⟩ (⟨formal parameter list for creation⟩)
 (state ⟨variable declaration list⟩)
 ⟨expression⟩]
```

The parts of `(state ...)` may be omitted. The following is a simple example of *class definition*:

```
[class counter counter-o ((int c0))
 (state (int (c c0)))
 (script
  (=>  [:add i] [c := (+ c i)])
  (==> [:get] !c))]
```

The above program fragment defines a class named `counter`, which supports an *interface type* `counter-o` (details of *interface types* will be described later). `c0` is a formal parameter of type `int` for object creation, and `c` is a state variable of type `int` whose initial value is `c0`. An object of this class accepts a message `:add` to add the value of the argument to `c`, and also accepts a message `:get` to reply the current value of `c`. The form `!` returns a value to the *standard reply destination* included in `:get` message.

4.2.2 Object Creation

In ABCL/ST, an *object creation* is written in the following form:

```
(new-⟨class name⟩ ⟨actual parameter list for creation⟩)
```

For example, an object of class `counter` can be created by the following form:

```
(new-counter 0)
```

The form returns the name (or ID, or mail address) of the newly created object of class `counter`.

4.3 Messages

In ABCL/ST, messages are first class data. Message values could be copied, and assigned to variables in the same way as other data such as integers or real numbers. This property provides us a strong power of expression to manage messages (e.g., messages in a message, message sorting, delegation), and integrates request messages and reply messages.

4.3.1 Message Passing

In ABCL/ST, an object could perform a message passing, if it knows the target. The target is either an object name or a reply destination. A reply destination can be generated by the following *now type* message passing. Message passing is either *past type* or *now type*:

Past type (send and no wait)

A *past type* message passing is written in the following manner:

[⟨*target expression*⟩ `<=` ⟨*message expression*⟩]

For example, a past type message passing:

```
[x <= 10]
```

sends a message `10` to the target that is the value of variable `x`. The message is asynchronously sent and the sender object is not blocked.

Now type (send and wait)

In ABCL/ST, a now type message passing is written in the following form:

[⟨*target expression*⟩ `<==` [⟨*keyword*⟩ ⟨*actual parameter list*⟩]]

For example, a now type message passing to objects of `counter` class can be described as:

```
[o <== [:get]]
```

The message is asynchronously sent and the sender object is blocked until it receives a reply. The reply value becomes the value of the message passing form itself. This message passing is similar to that of sequential OO languages, but the communication of the reply is also performed via message passing, whose target is the *standard reply destination* included in message `:get`. The *standard reply destination* is supplied to the message by the now type message passing.

The right hand side of now type message passing is not an ⟨*expression*⟩, while that of past type message passing is an ⟨*expression*⟩. This is because, in ABCL/ST, a message is always sent as a first class datum, and the *standard reply destination* is a part of the message, whereas, in ABCL/1, the *standard reply destination* is located on the outside of the message. On past type message passing, the value of ⟨*message expression*⟩ is directly sent as a message. On the other hand, on now type message passing, the form `[:get]` is transformed into `[:get` ⟨*standard reply destination*⟩`]` before its sending. The reply destination is a synchronizer for the reply message and the *continuation* of the now type message passing form.

4.3.2 Message Acceptance

Messages sent to an object are first stored in the message queue of the object.

The `wait-for` form:

```
(wait-for
 (=> ⟨matching pattern⟩ ⟨expression list⟩)
       ...
 (=> ⟨matching pattern⟩ ⟨expression list⟩))
```

accepts the first message that satisfies a ⟨*matching pattern*⟩ from the message queue and executes the corresponding ⟨*expression list*⟩. The value of the last ⟨*expression*⟩ of ⟨*expression list*⟩ becomes the value of the `wait-for` form. Unmatched messages are remained in the message queue.

The `script` form:

```
(script
 (=> ⟨matching pattern⟩ ⟨expression list⟩)
       ...
 (=> ⟨matching pattern⟩ ⟨expression list⟩))
```

is almost the same as `wait-for` form except that `script` form deletes unmatched message from its message queue and repeatedly waits for the next message after execution of an ⟨*expression list*⟩. At this point, the computation model for ABCL/ST is different from the model described in Section 2.1.1. In ABCL/ST, `script` is a command which changes the object's execution mode from active to dormant to accept a message.

In the `wait-for` and `script` forms, `(==> [:get] !c)` is an abbreviation of:

```
(=> [:get std-rply-dst] [std-rply-dst <= c])
```

4.3.3 Interface of Object

The next example tells us how to specify the *interface type*:

```
[class bias (obj real) ((real v) ((obj real) out))
 (script
  (=>inp [out <= (+ inp v)]))]
```

In this example, the *interface type* is `(obj real)`, which means that objects of this class accept messages of type `real`.

An object of class `bias` receives a message `inp` of type `real`, adds the bias value `v` to it, then sends the value to another object `out` of type `(obj real)` .

As we can see in this example, in ABCL/ST, messages for an object are of only a single type, and an interface type of an object is syntactically denoted by:

```
(obj ⟨type⟩)
```

Here, ⟨*type*⟩ is the type of messages which the object can accept.

In order to define *interface type* `counter-o` for the previous `counter` example, the following *type definition* can be used:

```
(deftype counter-o (obj obj-msg-counter-o)
         obj-msg-counter-o (union [:add int] [:get (@ int)]))
```

In this definition, `counter-o` is defined as a type of objects whose messages are of a *disjoint-union type* `obj-msg-counter-o`, which is defined as a union of *keyword types* (a keyword type is a disjoint-union type with a single injection tag) `[:add int]` and `[:get (@ int)]`. The type of the *standard reply destination* of `:get` message is `(@ int)`, which means that the reply destination accepts a value of type `int`.

4.4 Tuples and Tagged Values

In ABCL/ST, a tuple is constructed by `[`⟨*expression*⟩···⟨*expression*⟩`]` and a tagged value is constructed by `(`⟨*keyword*⟩ ⟨*expression*⟩`)`.

There is a syntax sugar concerning tuple and tagged values; for example, `[:tag1 10 20 30]` is an abbreviation of `(:tag1-*-*-* [10 20 30])` (# of `*` denotes the arity of tuple).

4.5 Types

The types in ABCL/ST include:

- Basic types such as `int`, `real`, `bool`,
- Tuple types such as `[int real]`,
- Disjoint-union types, which include keyword types such as `[:tag1 int int int]` and disjoint-union names specified by type definitions (described below),
- Class types, which are specified by class definitions,
- Object types (interface types), such as `(obj [:tag1 int int int])`, which indicates that the object can accept messages of type `[:tag1 int int int]`,
- Reply destination types, such as `(@ int)`, which indicates that the reply destination can receive a reply message of type `int`.

4.5.1 Defining Subtype Relations

A `union` operation on disjoint-union types is used to specify the subtype relation. Keyword types, such as `[:tag1 int int int]`, are used to attach tags to values. A keyword type is a disjoint-union type with a single injection tag. The subtype relations between disjoint-union types are specified by the following type definitions[1]:

[1]More mathematical treatment of these definitions is discussed in Chapter 5

```
(deftype weekday (union [:mon] [:tue] [:wed] [:thu] [:fri]))
(deftype weekend (union [:sat] [:sun]))
(deftype week (union weekday weekend))
```

Here, `week` is a supertype of `[:mon]`, `weekday` and `weekend`.

Subtype relations between object types whose messages are of disjoint-union types are specified by the following type definitions:

```
(deftype counter-o (obj obj-msg-counter-o)
         obj-msg-counter-o (union [:add int] [:get (@ int)]))

(deftype counter-with-reset-o (obj obj-msg-counter-with-reset-o)
         obj-msg-counter-with-reset-o (union [:reset] obj-msg-counter-o))
```

Here, `obj-msg-counter-with-reset-o` is a supertype of `obj-msg-counter-o`, and `counter-with-reset-o` is a subtype of `counter-o`. There is an abbreviation for object type definition; for example, the above definition can be written as:

```
(deftype (obj-msg counter-o) (union [:add int] [:get (@ int)]))
(deftype (obj-msg counter-with-reset-o) (union [:reset] obj-msg-counter-o))
```

The following class definition gives a class whose interface type is a subtype of that of `counter-o`:

```
[class counter-with-reset counter-with-reset-o ((int c0))
 (state (int (c c0)))
 (script
  (=>  [:add i] [c := (+ c i)])
  (=>  [:reset] [c := c0])
  (==> [:get] !c))]
```

The `union` operation on disjoint-union types is also used to define recursive types such as lists and trees:

```
(deftype list-of-int (union [:nil] [:cons int list-of-int]))
(deftype tree-of-int (union [:empty] [:node tree-of-int int tree-of-int]))
```

4.5.2 Type Definitions

The following type definition defines an abbreviation of ⟨*type*⟩:

`(deftype` ⟨*type name*⟩ ⟨*type*⟩`)`

After this definition, the use of ⟨*type name*⟩ always causes structural expansion of ⟨*type*⟩ at the program position of ⟨*type name*⟩.

The following type definition defines a disjoint-union name ⟨*disjoint-union name*⟩:

(deftype ⟨*disjoint-union name*⟩ (union $\{\{$⟨*disjoint-union type*⟩$\}\}$))

Here, $\{\{\}\}$ denotes more than zero repetition. After this definition, ⟨*disjoint-union name*⟩ becomes a supertype of all listed ⟨*disjoint-union type*⟩. Here, ⟨*disjoint-union type*⟩ is either ⟨*disjoint-union name*⟩ or ⟨*keyword type*⟩. `union` is not a part of type expressions; ⟨*disjoint-union name*⟩ is defined by `union` operation is not expanded upon its use.

For these reasons, we can not write as follows:

(deftype ⟨*object type name*⟩ (obj (union $\{\{$⟨*disjoint-union type*⟩$\}\}$)))

Instead of this, the following abbreviation is provided:

(deftype (obj-msg ⟨*object type name*⟩) (union $\{\{$⟨*disjoint-union type*⟩$\}\}$))

Here, a disjoint-union name is implicitly supplied (by the compiler) for the message type of ⟨*object type name*⟩.

4.6 Discussions

4.6.1 Arrays

Currently, ABCL/ST is not augmented with *arrays*.

A possible language design for arrays that follows the design principles described in Section 4.1 would be as follows: (1) an imperative operation, selective update of arrays, is provided, (2) arrays are not shared, (3) array elements should be supplied on its construction to avoid uninitialized values (for safety). Examples are as follows:

```
(let* (((array-2D real)
         (mat1 (for (i j) [10 10] (if (= i j) 1.0 0.0)))))
         ;; comprehensive construction of an array
   (dotimes (i 9)
     (dotimes (j 9)
       ;; imperative update of arrays
       [(aref mat1 [i j]) := (aref mat1 [(+ i 1) (+ j 1)])])))

;; function definition for matrix multiplication
(defun matrix-mul (array-2D real) (((array-2D real) a1 a2))
  (let* (([int  int]
            ([r1 c1] (boundaries a1))
            ([r2 c2] (boundaries a2))))
    (if (= c1 r2)
        (for (i j) [r1 c2]
             (reduce-+ (k) [c1]
                (* (aref a1 [i k]) (aref a2 [k j]))))
        (for (i j) [0 0] 0.0))))
```

A problem in the language semantics described above (namely, "pass-by-value") is a large amount of potential copy of array at the implementation level. One implementation-level solution is the use of "copy-on-write" coupled with "pass-by-pointer"; Moreover, "copy-on-write" can be enhanced to "copy for write" (where the copy is performed as early as possible if the "write" can not be avoided) in order to exploit other characteristics for optimization such as (1) no aliasing and (2) no sharing.

4.6.2 Inheritance

ABCL/ST provides the inheritance of interface types of objects by `union` operation on disjoint-union types of their messages. On the other hand, the class inheritance is not provided yet. In the case where a class definition contains a single `script` command, the conventional class inheritance mechanism could be used (except for the problem with inheritance anomaly[17]). For multiple `script` and `wait-for` commands, the class inheritance can be realized by supplying unique names to those commands.

Chapter 5

Type System for ABCL/ST

This chapter presents the type system for ABCL/ST, including types, subtype relation and typing rules for a simplified ABCL/ST.

First, in Section 5.1, for preparation, we will briefly examine two simple type systems to look at the duality between labeled record and disjoint union with labels, and the duality between values and continuations. Then we will regard messages as values and objects as continuations and present the reason why we have chosen disjoint union with labels to generate subtype relation.

Next, in Section 5.2, we will point out difficulties on efficient implementation of type system with subtyping and present two restrictions to provide efficiency, namely explicit subtype relation with names and restricted coercing. Then types and subtype relation are presented. Finally, in Section 5.3, the typing rules are presented after defining a simplified ABCL/ST, followed by some comments for our type checking algorithm.

5.1 Preliminaries

5.1.1 Two Simple Type Systems

Subtype relations are usually discussed on labeled record types. Other types generating subtype relations are disjoint-union types using labels as injection tags.

Let us look at a type system $T_l(ord_N)$, which is given by the syntax:

$$\begin{array}{rcl} \tau & ::= & b \,|\, \neg\tau \,|\, \{l:\tau,\ldots,l:\tau\} \,|\, [l:\tau,\ldots,l:\tau] \\ b & ::= & \texttt{bool} \,|\, \texttt{int} \,|\, \texttt{real} \end{array}$$

where l ranges over a finite (N) set of label names, $Dom(ord_N)$ is equal to the set of label names. $\{l:\tau,\ldots,l:\tau\}$ are labeled record types, whereas $[l:\tau,\ldots,l:\tau]$ are disjoint-union types with injection labels.

Types of the form $\neg\tau$ are types for *continuations*. A τ-accepting continuation (which accepts a τ type value and executes the rest of the computation) has a type $\neg\tau$; this notation appears in [8], where $\neg\tau$ is defined as $\tau \to \bot$; Intuitively, this means that the function can be applied to the input value of type τ, but the result is never returned to the calling point.

$T_l(ord_N)$ does not have function types like: $\tau_1 \rightarrow \tau_2$; however a continuation passing style (written as *cps*) function can be represented as a continuation which accepts a pair of the argument and a continuation for the reply:

$$\tau_1 \rightarrow \tau_2 \text{ is roughly equal to } \neg\{argument : \tau_1, reply\text{-}to : \neg\tau_2\}$$

$T_l(ord_N)$ does not have recursive types, but they are automatically introduced later by a naming mechanism which is required for efficient implementation.

Instead of giving subtype rules, let us think about embedding of the types of $T_l(ord_N)$ into the simple type system without labels, namely $T_d(N)$. $ord_N : \{l_i\} \rightarrow \{1, 2, \ldots, N\}$ is a one-to-one morphism and can be used for this embedding.

$T_d(N)$ is given by the following syntax:

$$\begin{array}{rcl} \tau & ::= & \top \,|\, \bot \,|\, b \,|\, \neg\tau \,|\, (\tau_1 \times \tau_2 \times \cdots \times \tau_N) \,|\, (\tau_1 + \tau_2 + \cdots + \tau_N) \\ b & ::= & \texttt{bool} \,|\, \texttt{int} \,|\, \texttt{real} \end{array}$$

where $(\tau_1 \times \tau_2 \times \cdots \times \tau_N)$ is an N-tuple type, and $(\tau_1 + \tau_2 + \cdots + \tau_N)$ is a disjoint sum type with N tags.

The subtype relation of $T_d(N)$ is easily described as:

$$\vdash_{T_d(N)} \bot \leq \tau \qquad\qquad \vdash_{T_d(N)} \tau \leq \top$$

$$\vdash_{T_d(N)} b \leq b$$

$$\frac{\vdash_{T_d(N)} \tau_1 \leq \tau_1' \quad \ldots \quad \vdash_{T_d(N)} \tau_N \leq \tau_N'}{\vdash_{T_d(N)} (\tau_1 \times \cdots \times \tau_N) \leq (\tau_1' \times \cdots \times \tau_N')} \qquad \frac{\vdash_{T_d(N)} \tau_1 \leq \tau_1' \quad \ldots \quad \vdash_{T_d(N)} \tau_N \leq \tau_N'}{\vdash_{T_d(N)} (\tau_1 + \cdots + \tau_N) \leq (\tau_1' + \cdots + \tau_N')}$$

$$\frac{\vdash_{T_d(N)} \tau_1 \leq \tau_2}{\vdash_{T_d(N)} \neg\tau_2 \leq \neg\tau_1} \qquad \frac{\vdash_{T_d(N)} \tau_1 \leq \tau_2 \quad \vdash_{T_d(N)} \tau_2 \leq \tau_3}{\vdash_{T_d(N)} \tau_1 \leq \tau_3}$$

The equality between types of $T_d(N)$ is the usual structural equality between types, and the subtype relation satisfies:

$$\tau_1 \leq \tau_2, \tau_1 \geq \tau_2 \Longleftrightarrow \tau_1 = \tau_2$$

Now, *embedding function* from types of $T_l(ord_N)$ into types of $T_d(N)$ can be defined as:

$$\begin{array}{rcl} \psi_{ord_N}(b) & = & b \\ \psi_{ord_N}(\neg\tau) & = & \neg\,\psi_{ord_N}(\tau) \\ \psi_{ord_N}(\{l_1 : \tau_{11}, \ldots, l_n : \tau_{1n}\}) & = & (\tau_{21} \times \cdots \times \tau_{2N})\text{where } \tau_{2j} = \begin{cases} \psi_{ord_N}(\tau_{1i}) & \text{if } j = ord_N(l_i) \\ \top & \text{otherwise} \end{cases} \\ \psi_{ord_N}([l_1 : \tau_{11}, \ldots, l_n : \tau_{1n}]) & = & (\tau_{21} + \cdots + \tau_{2N})\text{where } \tau_{2j} = \begin{cases} \psi_{ord_N}(\tau_{1i}) & \text{if } j = ord_N(l_i) \\ \bot & \text{otherwise} \end{cases} \end{array}$$

The subtype relation between types of $T_l(ord_N)$ is defined by that of $T_d(N)$ in the following way:

$$\vdash_{T_l(ord_N)} \tau_1 \leq \tau_2 \overset{def}{\equiv} \vdash_{T_d(N)} \psi_{ord_N}(\tau_1) \leq \psi_{ord_N}(\tau_2)$$

This definition leads to the usual subtype relation:

$$\frac{\vdash_{T_l(ord_N)} \tau_1 \leq \tau_1' \quad \ldots \quad \vdash_{T_l(ord_N)} \tau_n \leq \tau_n'}{\vdash_{T_l(ord_N)} \{l_1 : \tau_1, l_2 : \tau_2, l_n : \tau_n, \ldots, l_{n+m} : \tau_{n+m}\} \leq \{l_1 : \tau_1', l_2 : \tau_2', l_n : \tau_n'\}}$$

$$\frac{\vdash_{T_l(ord_N)} \tau_1 \leq \tau_1' \quad \ldots \quad \vdash_{T_l(ord_N)} \tau_n \leq \tau_n'}{\vdash_{T_l(ord_N)} [l_1 : \tau_1, l_2 : \tau_2, l_n : \tau_n] \leq [l_1 : \tau_1', l_2 : \tau_2', l_n : \tau_n', \ldots, l_{n+m} : \tau_{n+m}']}$$

5.1.2 Objects as Continuations

Labeled records are often used as objects, but here we will present another representation of objects. Labeled records have their dual representation of disjoint-union types together with the duality between values and continuations:

$$\{l_1 : \tau_1, l_2 : \tau_2, \ldots, l_n : \tau_n\} \leftrightarrow \neg[l_1 : \neg\tau_1, l_2 : \neg\tau_2, \ldots, l_n : \neg\tau_n]$$

where a record type corresponds to a continuation which accepts either of $l_1, \ldots, l_n$ and returns a reply to the corresponding continuation of either of types $\neg\tau_1, \ldots, \neg\tau_n$.

Let us consider a record type whose l_1-field is a cps function. Then its dual representation would be:

$$\{l_1 : \neg\{argument : \tau_1, reply\text{-}to : \neg\tau_2\}\} \leftrightarrow \neg[l_1 : \neg\neg\{argument : \tau_1, reply\text{-}to : \neg\tau_2\}]$$

The dual representation reveals a problem of "objects as records", that is, there are two function-style transactions to dispatch an object's method: the caller should (1) pass l_1 and get $\neg\{argument : \tau_1, reply\text{-}to : \neg\tau_2\}$, (2) pass the argument of type τ_1 and get a reply of type τ_2. This is not problematic for sequential objects, but is for concurrent objects where we would not like to incur double of message traveling time. We rather prefer a single transaction:

$$\neg[l_1 : \{argument : \tau_1, reply\text{-}to : \neg\tau_2\}]$$

where l_1 is sent together with the argument.

For the reasons above, we choose continuations accepting disjoint unions for representation of objects. Regarding objects as continuations and regarding messages as values give us (1) an object-oriented view where a message is sent to its target, and (2) a uniform view on request messages and reply messages, namely both are value passing. Consequently, it is more natural using disjoint-union types for messages to incorporate subtype relation in object types.

As we can see, a subtype relation has the dual representation both on labeled records and disjoint unions with labels. To avoid confusion with mixing these two, we substitute tuples, denoted by $(\tau, \tau, \ldots, \tau)$, for labeled records; as a result, the above example is represented as follows:

$$\neg[l_1 : (\tau_1, \neg\tau_2)]$$

We should note that the type of objects proposed above is based on the external view of objects, namely, the viewpoint of the message sender to the objects. We think that the record types are more appropriate as types of objects from the internal view of objects, namely, from the viewpoint of the implementor of objects; there, the object's own properties should not be concealed from the implementor and may be manipulated for inheritance.

5.2 The Types and Subtype Relation for ABCL/ST

5.2.1 Restrictions for Efficiency

Embedding $T_l(ord_N)$ into $T_d(N)$ gives us an implementation scheme of subtyped languages, where time-complexity of a field selection or a case branch is $O(1)$ (just a direct table indexing operation), but space-complexity of $O(N)$ is required.

Another naive implementation may use an associative list or a binary search algorithm; they would reduce the space complexity to $O(M)$ (where M be the number of injection tags which the tested value may have), but their time complexity would increase to $O(N)$ or $O(\log N)$. Hash search algorithms may improve the time complexity to $O(1)$; but its coefficient is still large.

We will propose a novel implementation where the time complexity of a look-up is $O(1)$ and its space complexity is slightly more than $O(M)$, where the look-up is performed by direct indexing. The implementation and its precise space complexity are described in Section 8.2.

For this complexity requirement, we use the type system where the subtype relation between disjoint-union types is explicitly given by the programmer. The system reduces overhead by reflecting the programmer's mind via naming.

The explicit declaration of subtype relation is performed with *union* operations on disjoint-union types. The resulting type of a union operation is a supertype of operands of the union operation:

```
(deftype weekday (union [:mon] [:tue] [:wed] [:thu] [:fri]))
(deftype weekend (union [:sat] [:sun]))
(deftype week (union weekday weekend))
```

where, for example,

$$\begin{array}{l} \texttt{[:mon]} \leq \texttt{weekday}, \\ \texttt{weekday} \leq \texttt{week}, \quad \text{and} \\ \texttt{weekend} \leq \texttt{week} \quad \text{hold.} \end{array}$$

By the definitions above, *explicit disjoint-union names* `weekday`, `weekend`, and `week` are introduced to give their names to the corresponding disjoint-union types.

The type definition operator, `union`, is not the disjoint sum. For example, someone may want `[:fri]` in `weekend`:

```
(deftype weekday (union [:mon] [:tue] [:wed] [:thu] [:fri]))
(deftype weekend (union [:fri] [:sat] [:sun]))
(deftype week (union weekday weekend))
```

where `week` has 7 elements rather than 8 elements, because the union operation maps two `[:fri]` into one.

Another restriction for the sake of high efficiency is that: for high-speed type conversion at the implementation level, the application of coercing rules, such as the one between **int** and **real**, (i.e., inferring subtype relations without using a mapping provided by the union operation) is restricted as is describe in the subsequent sections.

5.2.2 The Types and Subtype Relation

This section presents a type system T_r for ABCL/ST. In the type system for ABCL/ST, **obj** τ and @ τ corresponds to $\neg\tau$ in $T_l(ord_N)$ and $T_d(N)$. The set of types (ranged over by τ) of T_r is given by the syntax:

$$\begin{array}{lcl} \tau & ::= & \top \mid \bot \mid b \mid c \mid u \mid (\tau, \tau, \ldots, \tau) \mid \textbf{obj}\ \tau \mid @\ \tau \mid \tau \rightarrow \tau \\ b & ::= & \textbf{int} \mid \textbf{real} \mid \textbf{bool} \mid \textbf{nullobj} \\ u & ::= & k\ \tau \mid s \end{array}$$
$$F_s = \{s_1 \mapsto \{u_{11}, \ldots, u_{1m_1}\}, \ldots, s_n \mapsto \{u_{n1}, \ldots, u_{nm_n}\}\}$$
$$F_c = \{c_1 \mapsto \textbf{obj}\ \tau_1,\ c_2 \mapsto \textbf{obj}\ \tau_2, \ldots,\ c_n \mapsto \textbf{obj}\ \tau_n\}$$

The type equality is just the structural equality.

The keyword k ranges over injection tags. **obj** τ is a type for objects which receive messages of type τ. @ τ is the type for reply destinations each of which receives a (single) message of type τ. c is a class name. F_s is a function which maps a disjoint-union name into a set of disjoint-union types (either disjoint-union name or keyword type). F_s is specified by the programmer. One constraint on F_s is that the following function $Kset(F_s)$ must be well-defined:

$$\begin{array}{l} \mathrm{Dom}(Kset(F_s)) = \{u_j\}, \\ Kset(F_s)(s) = \displaystyle\bigcup_{u' \in F_s(s)} Kset(F_s)(u') \\ Kset(F_s)(k\ \tau) = \{k\ \tau\} \end{array}$$

$Kset(F_s)$ recursively expands disjoint-union names until keyword types and finally provides the set of keyword types, which are the original inputs to union operation. For $Kset(F_s)$ to be well-defined, the recursive expansion must be terminated.

The subtype relation $\leq_u$ of T_r based on the explicit subtype relation on disjoint-union types is defined as follows:

$$C \vdash b \leq_u b \qquad\qquad C \vdash c \leq_u c$$

$$C \vdash s \leq_u s$$

$$C \vdash k\ \tau \leq_u s \quad \text{if } k\ \tau \in F_s(s) \qquad C \vdash s' \leq_u s \quad \text{if } s' \in F_s(s)$$

$$\frac{C \vdash \tau_1 \leq_u \tau_2}{C \vdash k\ \tau_1 \leq_u k\ \tau_2} \qquad \frac{C \vdash \tau_1 \leq_u \tau_1' \quad \ldots \quad C \vdash \tau_n \leq_u \tau_n'}{C \vdash (\tau_1, \ldots, \tau_n) \leq_u (\tau_1', \ldots, \tau_n')}$$

$$\frac{C \vdash \tau_1 \leq_u \tau_2}{C \vdash \mathbf{obj}\ \tau_2 \leq_u \mathbf{obj}\ \tau_1} \qquad \frac{C \vdash \tau_1 \leq_u \tau_2}{C \vdash \mathbf{@}\ \tau_2 \leq_u \mathbf{@}\ \tau_1}$$

$$\frac{C \vdash \tau_1' \leq_u \tau_1 \quad C \vdash \tau_2 \leq_u \tau_2'}{C \vdash \tau_1 \rightarrow \tau_2 \leq_u \tau_1' \rightarrow \tau_2'} \qquad \frac{C \vdash \tau_1 \leq_u \tau_2 \quad C \vdash \tau_2 \leq_u \tau_3}{C \vdash \tau_1 \leq_u \tau_3}$$

where $C = (F_s, F_c)$. Of course $\leq_u$ is a partial order on types of T_r.

The subtype relation $\leq$ is defined by adding the restricted coercing rules in the following way:

$$\frac{C \vdash \tau_1 \leq_u \tau_2}{C \vdash \tau_1 \leq \tau_2} \qquad \frac{C \vdash \tau_1 \leq \tau_2 \quad C \vdash \tau_2 \leq \tau_3}{C \vdash \tau_1 \leq \tau_3}$$

$$\frac{C \vdash \tau_1 \leq \tau_2}{C \vdash k\ \tau_1 \leq k\ \tau_2} \qquad \frac{C \vdash \tau_1 \leq \tau_1' \quad \ldots \quad C \vdash \tau_n \leq \tau_n'}{C \vdash (\tau_1, \ldots, \tau_n) \leq (\tau_1', \ldots, \tau_n')}$$

$$C \vdash \bot \leq \tau \qquad C \vdash \tau \leq \top$$

$$C \vdash \mathbf{int} \leq \mathbf{real} \qquad C \vdash \mathbf{int} \leq \mathbf{bool}$$

$$C \vdash \mathbf{nullobj} \leq c \qquad C \vdash \mathbf{nullobj} \leq \mathbf{obj}\ \tau$$

$$C \vdash \mathbf{nullobj} \leq \mathbf{@}\ \tau \qquad C \vdash c \leq \mathbf{bool}$$

$$C \vdash \mathbf{obj}\ \tau \leq \mathbf{bool} \qquad C \vdash \mathbf{@}\ \tau \leq \mathbf{bool}$$

$$C \vdash \mathbf{obj}\ \tau \leq \mathbf{@}\ \tau \qquad C \vdash c \leq \mathbf{obj}\ \tau \quad \text{if } \mathbf{obj}\ \tau = F_c(c)$$

The **nullobj** type has a single element *nil*; a message passing to *nil* always succeeds but the message is simply discarded. Thus **nullobj** is a subtype of all object types, class types and reply destination types. Whether a message target is *nil* or not can be tested by coercing it into **bool** type. The rule of $\mathbf{obj}\ \tau \leq \mathbf{@}\ \tau$ implies that an object name can be used instead of a reply destination of the same message type.

These coercing rules for $\leq$ can not appear on proof tree nodes which are higher than the rules for $\mathbf{obj}\ \tau_2 \leq_u \mathbf{obj}\ \tau_1$, $\mathbf{@}\ \tau_2 \leq_u \mathbf{@}\ \tau_1$, and $\tau_1 \rightarrow \tau_2 \leq_u \tau_1' \rightarrow \tau_2'$. This is mainly for efficiency; the details of the resulting efficient implementation are described in Section 8.2.2.

In addition to the constraint on $\mathit{Kset}(F_s)$, F_s is constrained to meet the following condition: for all $s \in \mathrm{Dom}(F_s)$ and k, $\{\tau \mid k\ \tau \in \mathit{Kset}(F_s)(s)\}$ is either an empty set or a set which has a unique maximal element in terms of $\leq$. Thus each s can be regarded as a disjoint union of the maximal elements with the keywords as the injection tags. Of course, these constraints would be checked by the compiler.

5.3 Typing for ABCL/ST

This section describes the typing for ABCL/ST. To simplify its description, first, we introduce a simplified ABCL/ST in Section 5.3.1. Then we describe the typing rules for the simplified ABCL/ST in Section 5.3.2.

The goals of the typing are:

1. To eliminate unnecessary casts which can be found in C++,

2. To hardly fail on the subtype relations which form not a lattice but a partial order.

These goals are realized by the use of approximate minimum types and type assignment in a top-down manner on the syntax tree.

5.3.1 A Simplified Language for Typing Rule Description

The simplified ABCL/ST for typing rule description is given by the syntax:

$$
\begin{array}{rcl}
l & ::= & x \,|\, (l, l, \ldots, l) \,|\, \#n\ l \\
m & ::= & c^\tau \,|\, x \,|\, (m, m, \ldots, m) \,|\, k\ m \\
e & ::= & c^\tau \,|\, x \,|\, (e, e, \ldots, e) \,|\, \#n\ e \,|\, k\ e \,|\, \textbf{out}\ e \,|\, \textbf{the}\ \tau\ e \,| \\
& & e\ e \,|\, e + e \,|\, e = e \,|\, l := e \,|\, e \Leftarrow e \,|\, \textbf{sync2}\ x\ e \,|\, @\ x \,| \\
& & \textbf{let}\ \tau\ x = e\ \textbf{in}\ e \,|\, \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \,|\, (e; e; \ldots; e) \,| \\
& & \textbf{case}\ e\ \textbf{of}\ m \Rightarrow e\ \textbf{or}\ m \Rightarrow e\ \textbf{or} \ldots \textbf{or}\ m \Rightarrow e \,| \\
& & \textbf{wait-for}\ m \Rightarrow e\ \textbf{or}\ m \Rightarrow e\ \textbf{or} \ldots \textbf{or}\ m \Rightarrow e \\
d & ::= & \textbf{defclass}\ c\ \tau\ \tau\ x\ \textbf{state}\ \tau\ x = e, \tau\ x = e, \ldots, \tau\ x = e\ \textbf{do}\ e
\end{array}
$$

where τ ranges over the set of types, k ranges over the set of injection tags, l ranges over the set of left expression for assignments, m ranges over the set of pattern expression for matching, e ranges over the set of usual expressions, and d is a class definition. $\#n\ e$ is n-th field selection of tuple expression e.

"$\textbf{defclass}\ c\ \tau\ \tau_0\ x_0\ \textbf{state}\ \tau_1\ x_1 = e_1, \tau_2\ x_2 = e_2\ \textbf{do}\ e_b$" defines a class named c. c is also used as a type. τ is an interface type, which is represented by $\textbf{obj}\ \tau_m$. τ_m is the type of messages which objects of class c can receive. τ_m can be referred to by **mymsgtype** within the class definition. $\tau_0\ x_0$ is the parameter (and its type) for object creation. $\tau_i\ x_i = e_i (i = 1, \ldots)$ are declaration of state variables and expressions for their initial values. The created object executes e_b. The class definition updates F_c to be $F_c(c) = \tau$.

"$l := e$" denotes an assignment. "$e_t \Leftarrow e_2$" sends message e_m to the target e_t. "$e_1 = e_2$" is an equality check. Note that, in the equality check, we do not have to be worried about the lost labels by subtyping, since we employ disjoint-union types rather than record types for the subtype relations; a supertype of a disjoint-union type never lose the labels in contrast to the behavior of record types.[1]

[1] Despite of this good property, we still use the notion of minimum types in our type checking algorithm for the equality check expressions described in Section 5.3.3, because of the presence of coercing rules.

$$A \vdash_l x : \tau \quad \text{if } \tau \in \mathrm{Dom}(A), A(x) = \tau$$

$$\frac{A \vdash_l l_i : \tau_i \ (\forall i = 1, \ldots, n)}{A \vdash_l (l_1, l_2, \ldots, l_n) : (\tau_1, \tau_2, \ldots, \tau_n)}$$

$$\frac{A \vdash_l l : (\tau_1, \ldots, \tau_n, \ldots, \tau_{n+m})}{A \vdash_l \#n\ l : \tau_n}$$

Figure 5.1: Typing Rules for Left Hand Side Expressions

"**sync2** x e" synchronizes a reply to x and the local evaluation of e and makes their pair as the result. This is used to implement now-type message passing; x is like a block name in Common Lisp (i.e., `(block` ⟨*block name*⟩ `... (return-from` ⟨*block name*⟩ `1) ...))`, but instead of using `return-from x`, @ x produces the reply destination. The now-type message passing form `[x2 <== [:msg 1]]` is implemented as #1 **sync2** x_1 $x_2 \Leftarrow$ *:msg* (@ $x_1, 1$), where the reply destination produced by @ x_1 is passed to object x_2 as the first message argument and the reply from x_2 is extracted as the first element of the resulting pair of synchronization.

5.3.2 The Typing Rules

The type judgment on expressions (e) is represented by 4-tuple $A, B \vdash_C e : \tau$. A is a type environment of variable identifiers, B is an environment of the synchronizing identifiers described above. τ is the assigned type. The typing rules are shown in Figure 5.3. The first rule in Figure 5.3 is the so-called *subsumption* rule, which permits an expression to have a type if the expression has its subtype.

A_1A_2 and $A_1 + A_2$ which are used in the typing rule description are defined as:

$$A = A_1A_2 \iff$$
$$\mathrm{Dom}(A) = \mathrm{Dom}(A_1) \cup \mathrm{Dom}(A_2),$$
$$A(x) = \begin{cases} A_1(x) & \text{if } x \in \mathrm{Dom}(A_1) - \mathrm{Dom}(A_2) \\ A_2(x) & \text{if } x \in \mathrm{Dom}(A_2) \end{cases}$$

$$A = A_1 + A_2 \iff$$
$$\mathrm{Dom}(A) = \mathrm{Dom}(A_1) \cup \mathrm{Dom}(A_2),$$
$$\mathrm{Dom}(A_1) \cap \mathrm{Dom}(A_2) = \emptyset,$$
$$A(x) = \begin{cases} A_1(x) & \text{if } x \in \mathrm{Dom}(A_1) \\ A_2(x) & \text{if } x \in \mathrm{Dom}(A_2) \end{cases}$$

The type judgment on left hand side expressions (l) is represented by 3-tuple $A \vdash_l l : \tau$. The typing rules are in Figure 5.1.

The type judgment on pattern expressions (m) is represented by 3-tuple $A \vdash_{C,m} m : \tau$. The typing rules are in Figure 5.2. In the rule for constant (the first rule in Figure 5.2), c^τ is compared with the value of τ's supertype. In the rule for keyword (the fourth rule

$$\frac{\vdash_C \tau \leq \tau' \quad \vdash_C \tau' \neq \top}{\{\} \vdash_{C,m} c^\tau : \tau'}$$

$$\{x \mapsto \tau\} \vdash_{C,m} x : \tau$$

$$\frac{A_i \vdash_{C,m} m_i : \tau_i \ (\forall i = 1, \ldots, n)}{A_1 + \cdots + A_n \vdash_{C,m} (m_1, m_2, \ldots, m_n) : (\tau_1, \tau_2, \ldots, \tau_n)}$$

$$\frac{A \vdash_{C,m} m : \tau_2 \quad \vdash_C \tau \leq \tau_2 \ (\forall \tau \in \{\tau' \mid k\ \tau' \in Kset(F_s)(\tau_1)\})}{A \vdash_{C,m} k\ m : \tau_1}$$

Figure 5.2: Typing Rules for Pattern Expressions

in Figure 5.2), m is used to receive the value in the tagged value; τ_2 must be a supertype of all possible types of the carried value.

5.3.3 The Type Checking

There are a lot of solutions of type assignments to a given (sub)expression which follow the typing rules described in the previous section. To perform type checking for subtyped languages, usually, a type checking algorithm, which assigns a minimum type to each (sub)expression, would be employed for sound and complete type checking. In the proposed type system, however, such a minimum type does not always exist even when a solution which satisfies the typing rules exists. This is because of the restriction of our type system which is required for efficiency; the restriction increases the cases in which *least upper bound* of two types does not exist.

By all means, even under such a restriction, there would be a complete algorithm which computes all possible type assignments. However, ambiguity on selecting a single type from the solutions still remains. We would rather employ a type checking algorithm which eliminates such ambiguity and provides a single type for each (sub)expression as a result, because this makes it easier for the user to understand what the result of the algorithm is, and for the implementor to develop the optimizing compiler using the provided type information.

Our type checking algorithm proposed in the following is sound but not complete with respect to the typing rules in the previous section, but it is not ambiguous for selecting a single type for each (sub)expression. Furthermore, it hardly fails in spite of the restriction on the type system, because it basically assigns maximum types rather than minimum types. The minimum type approximation rules are only applied for six kinds of subexpression, namely, functions of function applications, targets of message passings, the operands of numerical operations, the both sides of equality checks, examined parts of case expressions, and inner parts of reply-synchronization expressions. For those programs that these approximations succeed in isolating the minimum types, the present type checking algorithm is complete; and most programs written by the user seem to belong

$$\frac{A, B \vdash_C e : \tau \quad \vdash_C \tau \leq \tau'}{A, B \vdash_C e : \tau'}$$

$$A, B \vdash_C c^\tau : \tau$$

$$A, B \vdash_C x : \tau \qquad \text{if } x \in \mathrm{Dom}(A), A(x) = \tau$$

$$\frac{A, B \vdash_C e_i : \tau_i \; (\forall i = 1, \ldots, n)}{A, B \vdash_C (e_1, e_2, \ldots, e_n) : (\tau_1, \tau_2, \ldots, \tau_n)}$$

$$\frac{A, B \vdash_C e : (\tau_1, \ldots, \tau_n, \ldots, \tau_{n+m})}{A, B \vdash_C \#n \; e : \tau_n}$$

$$\frac{A, B \vdash_C e : \tau}{A, B \vdash_C k \; e : k \; \tau}$$

$$\frac{A, B \vdash_C e : k \; \tau}{A, B \vdash_C \mathbf{out} \; e : \tau}$$

$$\frac{A, B \vdash_C e : \tau}{A, B \vdash_C \mathbf{the} \; \tau \; e : \tau}$$

$$\frac{A, B \vdash_C e_1 : \tau_1 \to \tau \quad A, B \vdash_C e_2 : \tau_1}{A, B \vdash_C e_1 \; e_2 : \tau}$$

$$\frac{A, B \vdash_C e_1 : \tau_1 \quad A, B \vdash_C e_2 : \tau_2 \quad \vdash_C \tau = \text{num-lub}(\tau_1, \tau_2)}{A, B \vdash_C e_1 + e_2 : \tau}$$

$$\frac{A, B \vdash_C e_1 : \tau \quad A, B \vdash_C e_2 : \tau \quad \vdash_C \tau \neq \top}{A, B \vdash_C e_1 = e_2 : \mathbf{bool}}$$

$$\frac{A \vdash_l l : \tau \quad A, B \vdash_C e : \tau}{A, B \vdash_C l := e : \tau}$$

$$\frac{A, B \vdash_C e_1 : \tau_1 \quad A, B \vdash_C e_2 : \tau_2 \quad \vdash_C \tau_2 = \mathrm{Msg}(\tau_1)}{A, B \vdash_C e_1 \Leftarrow e_2 : \top}$$

$$\frac{A, B\{x \mapsto \{\tau_1', \ldots, \tau_n'\}\} \vdash_C e : \tau_2 \quad \vdash_C \tau \leq \tau_1 \; (\forall \tau \in \{\tau_1', \ldots, \tau_n'\})}{A, B \vdash_C \mathbf{sync2} \; x \; e : (\tau_1, \tau_2)}$$

$$A, B \vdash_C \mathbf{@} \; x : \mathbf{@} \; \tau \qquad \text{if } x \in \mathrm{Dom}(B), \tau \in B(x)$$

$$\frac{A, B \vdash_C e_1 : \tau_1 \quad A\{x \mapsto \tau_1\}, B \vdash_C e_2 : \tau}{A, B \vdash_C \mathbf{let} \; \tau_1 \; x = e_1 \; \mathbf{in} \; e_2 : \tau}$$

$$\frac{A, B \vdash_C e_1 : \mathbf{bool} \quad A, B \vdash_C e_2 : \tau \quad A, B \vdash_C e_3 : \tau}{A, B \vdash_C \mathbf{if} \; e_1 \; \mathbf{then} \; e_2 \; \mathbf{else} \; e_3 : \tau}$$

$$\frac{A, B \vdash_C e_i : \tau_i \; (\forall i = 1, \ldots, n-1) \quad A, B \vdash_C e_n : \tau}{A, B \vdash_C (e_1; e_2; \ldots; e_n) : \tau}$$

$$\frac{A, B \vdash_C e_c : \tau' \quad A_i \vdash_{C,m} m_i : \tau' \quad AA_i, B \vdash_C e_i : \tau \; (\forall i = 1, \ldots, n)}{A, B \vdash_C \mathbf{case} \; e_c \; \mathbf{of} \; m_1 \Rightarrow e_1 \; \mathbf{or} \; m_2 \Rightarrow e_2 \; \mathbf{or} \ldots \mathbf{or} \; m_n \Rightarrow e_n : \tau}$$

$$\frac{A_i \vdash_{C,m} m_i : \mathbf{mymsgtype} \quad AA_i, B \vdash_C e_i : \tau \; (\forall i = 1, \ldots, n)}{A, B \vdash_C \mathbf{wait\text{-}for} \; m_1 \Rightarrow e_1 \; \mathbf{or} \; m_2 \Rightarrow e_2 \; \mathbf{or} \ldots \mathbf{or} \; m_n \Rightarrow e_n : \tau}$$

Figure 5.3: Typing Rules for Expressions

to this category. However, to further prevent the isolating processes from failing, the user can supply a proper supertype for an expression explicitly by '**the** τ e'.

The single type assignment on e is represented by 4-tuple $A, B \vdash_{C,S} e : \tau$. A is a type environment of variable identifiers, B is an environment of the synchronizing identifiers. τ is the assigned type. The single type assignment rules are shown in Figure 5.5.

The minimum type approximation on e is represented by 3-tuple $A \vdash_{C,M} e :: T$. A is a type environment of variable identifiers, T stands for approximate minimum type. (Such approximation is not required when the subtype relations form a lattice.) T is a set of types, and represents another set of types $UB(T) = \{\tau \mid (\forall \tau' \in T).\tau' \leq \tau\}$, where T is always normalized not to contain obviously redundant elements to represent the same set of types as $UB(T)$, s.t. $(\forall \tau \in T)(\forall \tau' \in T - \{\tau\}).\tau' \not\leq \tau$; this normalization is performed by $maximal()$. The minimum type approximation rules are shown in Figure 5.6.

The single type assignment on m is represented by 3-tuple $A \vdash_{C,m,S} m : \tau$. The single type assignment rules are in Figure 5.4.

To perform type checking, elaborate unification algorithm is not necessary but both inheriting (namely, in a top-down manner on the syntax tree) of types and synthesizing (namely, in a bottom-up manner on the syntax tree) of approximate minimum types are required. The type checking algorithm is simple and almost identical to the single type assignment rules and the minimum type approximation rules. The type checking can be performed by using 4 kinds of procedures (when using side effects to assign types to (sub)expressions) on the syntax tree:

$TC_b(A, e) = T$ synthesizes the approximate minimum type (T) according to the minimum type approximation rules.

$TC_t(A, e, \sigma) = (B, S)$ inherits a request type (σ) and synthesizes an environment of the synchronizing identifiers (B) and a substitution (S) according to the single type assignment rules. If not fail, it assigns $S(\sigma)$ to e, where σ, S and $S(\sigma)$ are defined as follows:

$$\sigma ::= \tau \mid \textbf{size-unknown}(n, \sigma, svar) \mid \textbf{key-unknown}(\sigma, kvar)$$

$$S = \{svar_1 \mapsto n_1, \ldots, svar_{n_s} \mapsto n_{n_s}, kvar_1 \mapsto k_1, \ldots, kvar_{n_k} \mapsto k_{n_k}\}$$

$$\begin{array}{rcl} S(\textbf{size-unknown}(n, \sigma, svar)) & = & (\top, \ldots, \top, \overset{n}{\check{S}(\sigma)}\ \top, \ldots, \overset{m}{\check{\top}}) \quad \text{if } S = \{svar \mapsto m, \ldots\} \\ S(\textbf{key-unknown}(\sigma, kvar)) & = & k\ S(\sigma) \quad \text{if } S = \{kvar \mapsto k, \ldots\} \end{array}$$

$TC_l(l) = \tau$ synthesizes the type (τ) of a left hand expression (l) according to the left hand side typing rules. If not fail, it assigns τ to l.

$TC_m(m, \tau) = A$ inherits a type (τ) and synthesizes a type environment of variable identifiers (A) according to the single type assignment rules for pattern expressions. If not fail, it assigns τ to m.

$$\frac{\vdash_C \tau \leq \tau' \quad \vdash_C \tau' \neq \top}{\{\} \vdash_{C,m,S} c^{\tau} : \tau'}$$

$$\{x \mapsto \tau\} \vdash_{C,m,S} x : \tau$$

$$\frac{A_i \vdash_{C,m,S} m_i : \tau_i \ (\forall i = 1, \ldots, n)}{A_1 + \cdots + A_n \vdash_{C,m,S} (m_1, m_2, \ldots, m_n) : (\tau_1, \tau_2, \ldots, \tau_n)}$$

$$\frac{A \vdash_{C,m,S} m : \tau_2 \quad \vdash_C \tau_2 = \max\{\tau' \,|\, k\ \tau' \in Kset(F_s)(\tau_1)\} \quad \vdash_C \tau_1 \neq \top}{A \vdash_{C,m,S} k\ m : \tau_1}$$

Figure 5.4: Single Type Assignment Rules for Pattern Expressions

$$\frac{\vdash_C \tau \leq \tau'}{A, B \vdash_{C,S} c^\tau : \tau'}$$

$$\frac{\vdash_C \tau \leq \tau'}{A, B \vdash_{C,S} x : \tau'} \qquad \text{if } x \in \mathrm{Dom}(A), A(x) = \tau$$

$$\frac{A, B \vdash_{C,S} e_i : \tau_i \ (\forall i = 1, \ldots, n)}{A, B \vdash_{C,S} (e_1, e_2, \ldots, e_n) : (\tau_1, \tau_2, \ldots, \tau_n)}$$

$$\frac{A, B \vdash_{C,S} e : (\top, \ldots, \top, \tau_n, \top, \ldots, \top)}{A, B \vdash_{C,S} \#n\ e : \tau_n}$$

$$\frac{A, B \vdash_{C,S} e : \tau_2 \quad \vdash_C \tau_2 = \max\{\tau \mid k\ \tau \leq \tau_1\}}{A, B \vdash_{C,S} k\ e : \tau_1}$$

$$\frac{A, B \vdash_{C,S} e : k\ \tau}{A, B \vdash_{C,S} \mathbf{out}\ e : \tau}$$

$$\frac{A, B \vdash_{C,S} e : \tau \quad \vdash_C \tau \leq \tau'}{A, B \vdash_{C,S} \mathbf{the}\ \tau\ e : \tau'}$$

$$\frac{A \vdash_{C,M} e_1 :: \{\tau_2 \to \tau\} \quad A, B \vdash_{C,S} e_1 : \tau_2 \to \tau \quad A, B \vdash_{C,S} e_2 : \tau_2 \quad \vdash_C \tau \leq \tau'}{A, B \vdash_{C,S} e_1\ e_2 : \tau'}$$

$$\frac{A \vdash_{C,M} e_i :: \{\tau_i\} \quad A, B \vdash_{C,S} e_i : \tau_i \ (\forall i = 1, 2) \quad \vdash_C \tau = \text{num-lub}(\tau_1, \tau_2) \quad \vdash_C \tau \leq \tau'}{A, B \vdash_{C,S} e_1 + e_2 : \tau'}$$

$$\frac{A \vdash_{C,M} e_i :: T_i \quad A, B \vdash_{C,S} e_i : \tau \ (\forall i = 1, 2) \quad \vdash_C \{\tau\} = \mathit{maximal}(T_1 \cup T_2) \quad \vdash_C \mathbf{bool} \leq \tau'}{A, B \vdash_{C,S} e_1 = e_2 : \tau'}$$

$$\frac{A \vdash_l l : \tau \quad A, B \vdash_{C,S} e : \tau \quad \vdash_C \tau \leq \tau'}{A, B \vdash_{C,S} l := e : \tau'}$$

$$\frac{A \vdash_{C,M} e_1 :: \{\tau_1\} \quad A, B \vdash_{C,S} e_1 : \tau_1 \quad A, B \vdash_{C,S} e_2 : \tau_2 \quad \vdash_C \tau_2 = \mathrm{Msg}(\tau_1)}{A, B \vdash_{C,S} e_1 \Leftarrow e_2 : \top}$$

$$\frac{A \vdash_{C,M} e :: \{\tau_2\} \quad A, B\{x \mapsto T_1\} \vdash_{C,S} e : \tau_2 \quad \vdash_C \tau \leq \tau_1 \ (\forall \tau \in T_1) \quad \vdash_C \tau_2 \leq \tau_2'}{A, B \vdash_{C,S} \mathbf{sync2}\ x\ e : (\tau_1, \tau_2')}$$

$$A, B \vdash_{C,S} \mathbf{@}\ x : \mathbf{@}\ \tau \qquad \text{if } x \in \mathrm{Dom}(B), \tau \in B(x)$$

$$\frac{A, B \vdash_{C,S} e_1 : \tau_1 \quad A\{x \mapsto \tau_1\}, B \vdash_{C,S} e_2 : \tau}{A, B \vdash_{C,S} \mathbf{let}\ \tau_1\ x = e_1\ \mathbf{in}\ e_2 : \tau}$$

$$\frac{A, B \vdash_{C,S} e_1 : \mathbf{bool} \quad A, B \vdash_{C,S} e_2 : \tau \quad A, B \vdash_{C,S} e_3 : \tau}{A, B \vdash_{C,S} \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau}$$

$$\frac{A, B \vdash_{C,S} e_i : \top \ (\forall i = 1, \ldots, n-1) \quad A, B \vdash_{C,S} e_n : \tau}{A, B \vdash_{C,S} (e_1; e_2; \ldots; e_n) : \tau}$$

$$\frac{A \vdash_{C,M} e_c :: \{\tau'\} \quad A, B \vdash_{C,S} e_c : \tau' \quad A_i \vdash_{C,m,S} m_i : \tau' \quad AA_i, B \vdash_{C,S} e_i : \tau (\forall i = 1, \ldots, n)}{A, B \vdash_{C,S} \mathbf{case}\ e_c\ \mathbf{of}\ m_1 \Rightarrow e_1\ \mathbf{or}\ m_2 \Rightarrow e_2\ \mathbf{or} \ldots \mathbf{or}\ m_n \Rightarrow e_n : \tau}$$

$$\frac{A_i \vdash_{C,m,S} m_i : \mathbf{mymsgtype} \quad AA_i, B \vdash_{C,S} e_i : \tau (\forall i = 1, \ldots, n)}{A, B \vdash_{C,S} \mathbf{wait\text{-}for}\ m_1 \Rightarrow e_1\ \mathbf{or}\ m_2 \Rightarrow e_2\ \mathbf{or} \ldots \mathbf{or}\ m_n \Rightarrow e_n : \tau}$$

Figure 5.5: Single Type Assignment Rules for Expressions

$$A \vdash_{C,M} c^\tau :: \{\tau\}$$

$$A \vdash_{C,M} x :: \{\tau\} \qquad \text{if } x \in \mathrm{Dom}(A), A(x) = \tau$$

$$\frac{A \vdash_{C,M} e_i :: T_i \ (\forall i = 1, \ldots, n)}{A \vdash_{C,M} (e_1, e_2, \ldots, e_n) :: \{(\tau'_1, \tau'_2, \ldots, \tau'_n) \mid \tau'_i \in T_i \ (i = 1, \ldots, n)\}}$$

$$\frac{A \vdash_{C,M} e :: T \quad \vdash_C T' = \mathit{maximal}(\{\tau_n \mid (\tau_1, \ldots, \tau_n, \ldots, \tau_{n+m}) \in T\})}{A \vdash_{C,M} \#n \ e :: T'}$$

where $T = \{(\tau_{11}, \ldots, \tau_{1n}, \ldots, \tau_{1n+m}), \ldots, (\tau_{l1}, \ldots, \tau_{ln}, \ldots, \tau_{ln+m})\}$

$$\frac{A \vdash_{C,M} e :: T}{A \vdash_{C,M} k \ e :: \{k \ \tau' \mid \tau' \in T\}}$$

$$\frac{A \vdash_{C,M} e :: \{k \ \tau_1, \ldots, k \ \tau_n\}}{A \vdash_{C,M} \textbf{out} \ e :: \{\tau_1, \ldots, \tau_n\}}$$

$$A \vdash_{C,M} \textbf{the} \ \tau \ e :: \{\tau\}$$

$$\frac{A \vdash_{C,M} e_1 :: \{\tau_2 \rightarrow \tau\}}{A \vdash_{C,M} e_1 \ e_2 :: \{\tau\}}$$

$$\frac{A \vdash_{C,M} e_1 :: \{\tau_1\} \quad A \vdash_{C,M} e_2 :: \{\tau_2\} \quad \vdash_C \tau = \text{num-lub}(\tau_1, \tau_2)}{A \vdash_{C,M} e_1 + e_2 :: \{\tau\}}$$

$$A \vdash_{C,M} e_1 = e_2 :: \{\textbf{bool}\}$$

$$\frac{A \vdash_l l : \tau}{A \vdash_{C,M} l := e :: \{\tau\}}$$

$$A \vdash_{C,M} e_1 \Leftarrow e_2 :: \{\top\}$$

$$\frac{A \vdash_{C,M} e :: \{\tau_2\} \quad A, B\{x \mapsto T'\} \vdash_{C,S} e : \tau_2 \quad \vdash_C T_1 = \mathit{maximal}(T')}{A \vdash_{C,M} \textbf{sync2} \ x \ e :: \{(\tau'_1, \tau_2) \mid \tau'_1 \in T_1\}}$$

$$\frac{A\{x \mapsto \tau_1\} \vdash_{C,M} e_2 :: T}{A \vdash_{C,M} \textbf{let} \ \tau_1 \ x = e_1 \ \textbf{in} \ e_2 :: T}$$

$$\frac{A \vdash_{C,M} e_2 :: T_2 \quad A \vdash_{C,M} e_3 :: T_3 \quad \vdash_C T = \mathit{maximal}(T_2 \cup T_3)}{A \vdash_{C,M} \textbf{if} \ e_1 \ \textbf{then} \ e_2 \ \textbf{else} \ e_3 :: T}$$

$$\frac{A \vdash_{C,M} e_n :: T}{A \vdash_{C,M} (e_1; e_2; \ldots; e_n) :: T}$$

$$\frac{A \vdash_{C,M} e_c :: \{\tau'\} \quad A_i \vdash_{C,m,S} m_i : \tau' \quad AA_i \vdash_{C,M} e_i :: T_i \ (\forall i = 1, \ldots, n) \quad \vdash_C T = \mathit{maximal}(T_1 \cup \cdots \cup T_n)}{A \vdash_{C,M} \textbf{case} \ e_c \ \textbf{of} \ m_1 \Rightarrow e_1 \ \textbf{or} \ m_2 \Rightarrow e_2 \ \textbf{or} \ldots \textbf{or} \ m_n \Rightarrow e_n :: T}$$

$$\frac{A_i \vdash_{C,m,S} m_i : \textbf{mymsgtype} \quad AA_i \vdash_{C,M} e_i :: T_i \ (\forall i = 1, \ldots, n) \quad \vdash_C T = \mathit{maximal}(T_1 \cup \cdots \cup T_n)}{A \vdash_{C,M} \textbf{wait-for} \ m_1 \Rightarrow e_1 \ \textbf{or} \ m_2 \Rightarrow e_2 \ \textbf{or} \ldots \textbf{or} \ m_n \Rightarrow e_n :: T}$$

Figure 5.6: Minimum Type Approximation Rules for Expressions

Chapter 6

Runtime Architecture for EM-4

As is summarized in Section 2.2, EM-4 was originally designed and built to support the dataflow model in which function invocations are side-effect free. We found, however, that EM-4 had far greater potential; since it also supports RISC style imperative features such as memory writes, it is possible to realize side-effectual/state-transitional computation easily and efficiently. We decided to exploit this capability in order to realize concurrent objects which usually have local states.

Since EM-4 is a packet-driven, reactive architecture, it might seem that the reception of a message and the activation of the corresponding method could be carried out by using the direct-matching scheme, and the imperative execution of the method could be done using the instruction sequencing within the SCB. However, these features of the EM-4 hardware are too low level to be employed directly, and an appropriate software architecture must be provided in order to fully realize the concurrent OO computational model in an efficient way. For example, EM-4 requires the exit from SCB when a packet is to be received — as a result, software architecture must guarantee that the object code be restarted subsequently at a proper place after the packet reception.

The basic functionalities that the software architecture must provide are (1) (Remote) Object Creation, (2) Asynchronous Message Passing (messages larger than one packet) and Method Invocation over the network, (3) Intra-node Object Scheduling, and (4) System-wide Garbage Collection. In this section, we describe our proposed software architecture of ABCL/EM-4[26, 27, 35, 28, 34], which provides their efficient implementation, fully exploiting the hardware speed; we discuss the rationales and the advantages of the architecture.

6.1 Representation of Concurrent Objects

In order to implement the concurrent OO-model efficiently, we assume that (1) an object does not span multiple nodes, since the unit of concurrency is each individual object[1],

[1]Some formal theory of concurrent objects, notably [18], allows for internal concurrency; in this case, an object may span multiple nodes. Whether or not this is useful in practice is the subject of future research.

and that (2) there would be multiple objects per each physical node.

In ABCL/EM-4, each concurrent object is composed of several components. They are: the *object segment*, *reactive text*, and *class-specific text*. The illustrative overview of an object is depicted in Figure 6.1. We briefly describe the role of each:

- **The Object Segment:** The object segment basically holds the state information of an object, such as the *state variables*, the *temporary variables*, and the pointers for managing its *message queue*. The operand segment described in Section 2.2 is used for its representation. (When the size of an object becomes larger than the operand segment size (a fixed size, 128 words), the excessive state part of the object is allocated, and a forwarding pointer is placed in the object segment.) The head portion of the object segment are *entries* for realizing the message passing protocols. They are common to all the objects of all classes — the details will be described in the subsequent sections.

  The message queue is a queue of *message boxes* or *mboxes*, which are also fixed in size (16 to 32 words) and allocated from a freelist. An object also maintains a list of *temporary boxes*, which are used to hold the context when its internal routine (i.e., function) is invoked (they are not used for message reception itself). The temporary boxes are allocated from the shared freelist from which message boxes are allocated as well.

  The object segment has a unique address within a single node. Coupled with the unique node ID, the pair, which is encoded in one word, constitutes a system-wide unique object ID which serves as a reference to the object.

- **The Reactive Text:** The reactive text is the portion of the object code that appropriately handles each incoming packet to the object entries, and is common to all the objects of all classes. It resides in the template segment, and is pointed to by all the objects in the node. (Recall that the first word of the operand segment points to the template segment where the code to be invoked when the packet arrives.)

- **The Class-Specific Text:** The class-specific text is the portion of the object code that realizes the single thread of control of the object. The sequential execution is internally imperative. As the name indicates, there is one class-specific text per each class. It generally performs the followings: (1) initialization, (2) dequeuing of a message from the message queue, (3) method look-up, and (4) method execution. A method is usually executed as a sequence of SCBs, rather than as a single SCB so that packets may be received at the beginning of the SCBs as well as for doing fine-grained scheduling so that hardware packet buffer does not overflow.

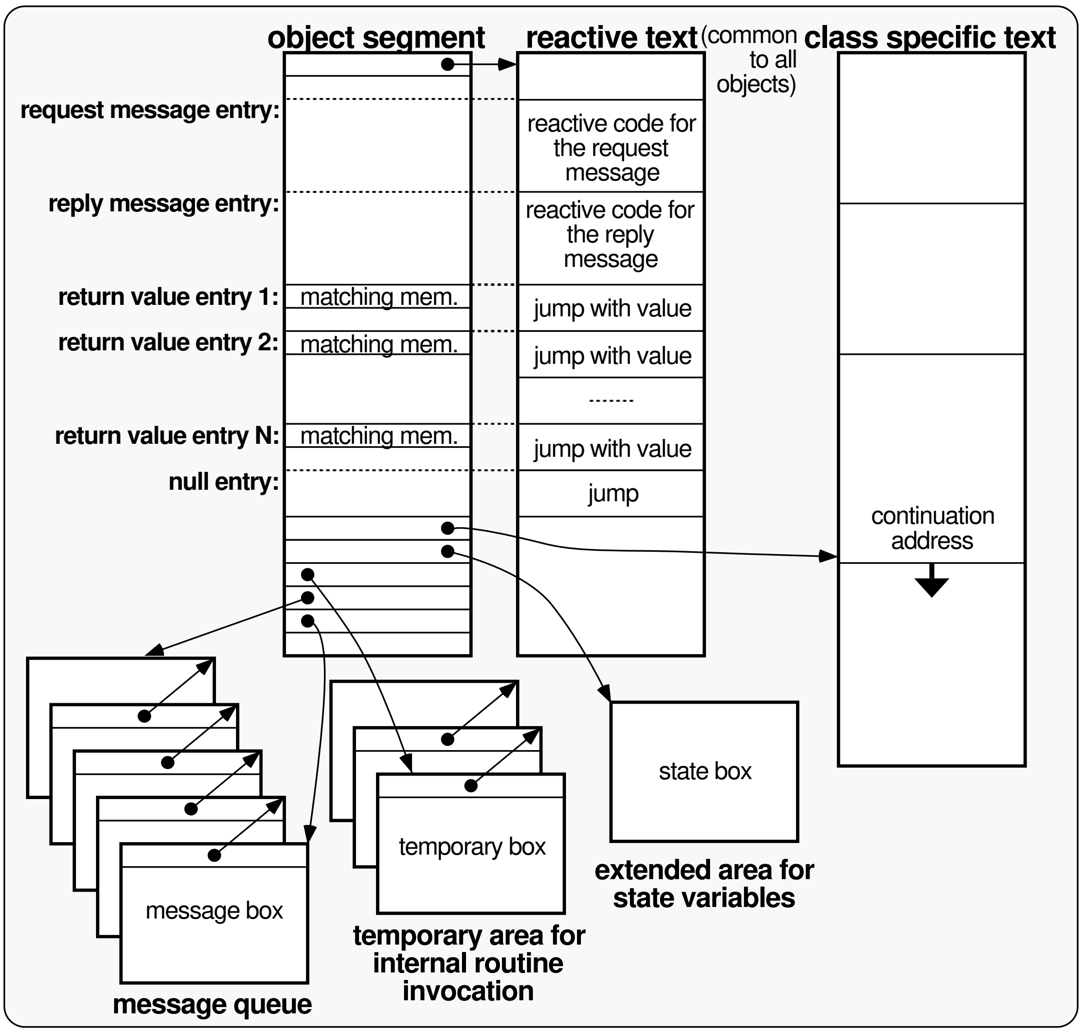


Figure 6.1: Representation of an Object

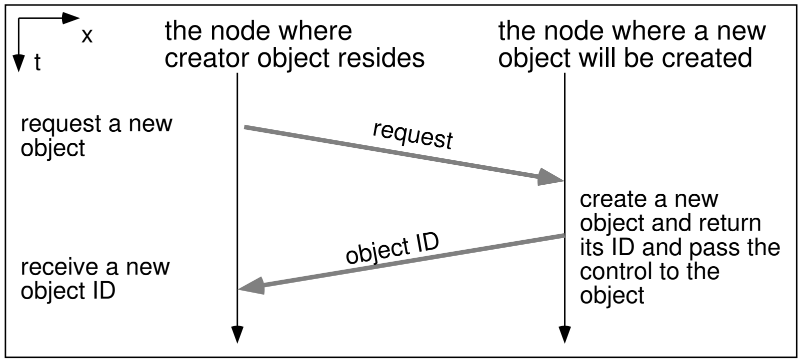


Figure 6.2: Packet Exchange for Object Creation

6.2 Outline of Object Creation and Message Passing in ABCL/EM-4

In this section, we outline the software architecture of the packet exchange protocols for implementing fast object creation and message sends. The details of the reactive invocation of the necessary code are described in Section 6.3.

- **Object Creation:** An object is created in a remote node by sending an object creation request (special) packet to the *object creator* in the remote node. The packet contains the class ID and the reply address. The object creator (1) allocates a single object segment, (2) sends the ID of the allocated object to the reply address, and then (3) passes the thread of control to the newly created object, thereby executing the necessary initialization code within the class-specific text. Figure 6.2 illustrates the packet exchanging protocol of object creation.

- **Request Message Send:**

  Sending an asynchronous request message to a concurrent object involves much more than just sending multiple packets, because the receiver node must construct a complete message data structure from multiple packets, which are separately sent and are merged with other packets to the receiver node.

  The packet exchange protocol for the message passing in ABCL/EM-4 proceeds as follows (Figure 6.3):

  Step 1: A request (special) packet is sent to the receiver node to allocate a message box of a fixed size for storing the message elements.

  Step 2: The message box is allocated in the receiver node, and a (normal) packet containing the message box address is returned to the sender node.

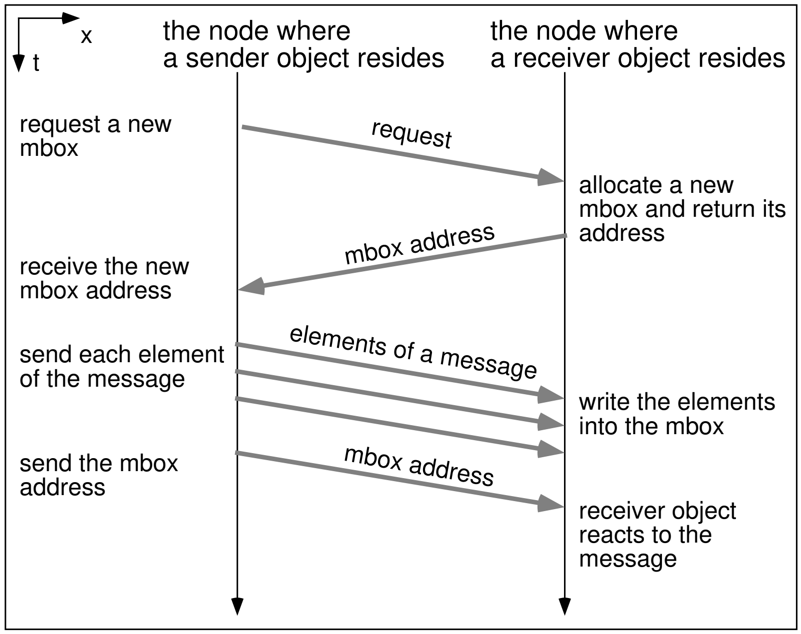


Figure 6.3: Packet Exchange for a Request Message Send

Step 3: A sequence of (special) packets containing elements of message data are transmitted from the sender to the receiver node. The data are written directly into the message box, as is described in the following sections.

Step 4: Finally, a (normal) packet containing the address of the message box is sent to the *receiver object* in the receiver node. The packet invokes a proper code block so that the receiver object can receive the message and a proper method be invoked if appropriate.

- **Reply Message Send:**

  On a now type request message send, the sender specifies a reply-destination of the request message as a pair consisting of the sender object ID and a new reply message box. The reply to a request message is treated analogously to message sends. Since the reply address is already known, and the reply message box will have been preallocated, steps 1 and 2 above are omitted in the usual case. (Figure 6.4).

6.3 Implementation of Message Passing

As mentioned earlier, the packet exchange protocols are not implemented in a traditional, procedural sense, but rather are implemented using the reactive invocation of appropriate code that govern its own part of the protocol. For example, during the execution of a method, an object has to exit from an SCB when it receives a packet for the above

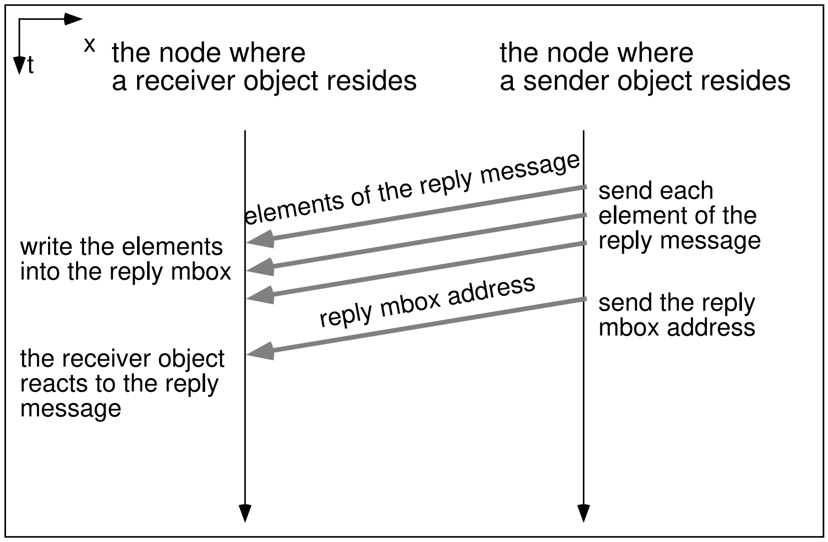


Figure 6.4: Packet Exchange for a Reply Message Send

protocols, such as message sending (Recall that a node cannot receive packets while executing an SCB.). Therefore, in order to guarantee the execution of the rest of the method, the packet exchange protocol for message send guarantees that the object code be restarted subsequently after the packet reception. We refer to proper addresses within the method code where the execution is to be resumed as the *continuation addresses* of the method.

ABCL/EM-4 employs both the normal packet and the special packet on EM-4 for realizing various protocols described above.

Recall that a normal packet invokes the code (with appropriate matchings) with the same displacement in a template segment as pointed to by an operand segment. Since every object segment (which is implemented with an operand segment) points to the reactive text common to all objects, the head portion of an object segment is used as reactive *entries* for receiving packets and invoking various code blocks within the reactive text (Figure 6.1). They are (1) the *Request Message Entry*, (2) the *Reply Message Entry*, (3) the *Return Value Entries*, and (4) the *Null Entry*:

- *Request Message Entry* — This is the entry where a packet containing the address of a message box is sent on Step 4 of the message sending described above. The corresponding code in the reactive text executes the following: If the sequential execution of the object is blocked and it is waiting for a new request message, it resumes the execution from the *continuation address* stored in a certain location in the object segment (Figure 6.1). If not, the message is simply appended to the message queue.

- *Reply Message Entry* — This is the entry where a packet for a reply message for a now type message is sent. The packet should contain the address of the reply message box in the original sender node. If the sequential execution of the object is blocked and it is waiting for the reply message, it resumes the execution from the continuation address stored in the head of the reply message box. If not, a flag indicating the presence of the reply is set.

- *Return Value Entries* — This is a set of two input entries, each of which waits for a pair of packets, one of which contains the returned value from a remote node for some previous request packet (e.g., address of newly allocated objects and/or message boxes), and the other contains the continuation address. The corresponding reactive text contains a 'jump with value' instruction, which jumps to the continuation address specified by the latter packet with the value of former in a register. Note that there are multiple Return Value Entries for requesting multiple resources, e.g., multicasting requires allocation of multiple message boxes on multiple nodes.

- *Null Entry* — This entry is similar to the return value entry, except that only the continuation address is waited for. It is employed for the packet-level, *hardware scheduling* only.

The reaction to a request message arrival at Request Message Entry depends on the *reactive states* of the object, namely (RS1) Enqueue, (RS2) Dispatch, and (RS3) Enqueue–Dispatch, as is briefly described above. When an object needs a new request message in *dormant mode*[2] (It has no messages in the message queue), the object changes its reactive state from RS1 to RS2 then it blocks its thread execution. When an object needs a new request message in *waiting mode* (No messages in the message queue are acceptable), the object changes its reactive state from RS1 to RS3 then it blocks its thread execution. When the execution is resumed by the message arrival, the object changes its reactive state to RS1. In the actual implementation, we prepare multiple template segments, each of which contains a reactive text specialized for each reactive state, and switch them dynamically by changing the linkage from the object segment to the template segments.

In waiting for a reply packet of a resource request on Return Value Entries, the continuation addresses are supplied by the original sender of the remote request packet by self-packet transmissions. The reader may wonder why this is done so, rather than employ the naive approach of storing the continuation address in some local memory (as is the case for the Request Message Entry). The rational is as follows: For waiting one reply, there is actually no overhead incurred by comparison to the latter, more naive approach, because of the hardware characteristics of EM-4. When waiting for multiple return values of multiple resource requests, the code that is resumed for the kth entry merely self-transmits the packet containing the continuation address for the $k+1$st entry, except for the next to the last one, which self-transmits a packet containing the continuation address of the code to be executed after the values are received. This has the advantage

[2] See Section 2.1.1, if necessary.

of eliminating the need for maintaining the counter on how many packets have been returned, achieving faster execution speed.

Special packets, on the other hand, are employed to invoke special-purpose *managers* at each node. Recall that a special packet invokes code at some fixed location, and allows the packet to carry two user-defined values instead of one for the normal packet. For ABCL/EM-4, the manager code fragments are invoked with special packets. They are: the *object creator*, the *message box allocator*, and the *message writer* (The dark-grey boxes in Figure 6.5). The former two has the obvious meanings; the message writer is actually a single instruction, which writes an element of a message into a message box.

Given the above entries, here is the more detailed description of the packet exchange protocol for a message send (Figure 6.5):

Step 1: A request packet is sent to the message box allocator. The packet contains the address of the receiver object (only the destination node information is used here), and the address of the Return Value Entry of the sender object (Figure 6.5(1)). Before exiting from the SCB the sender self-sends a packet containing the continuation address within the method where the evaluation of the arguments of the message is performed.

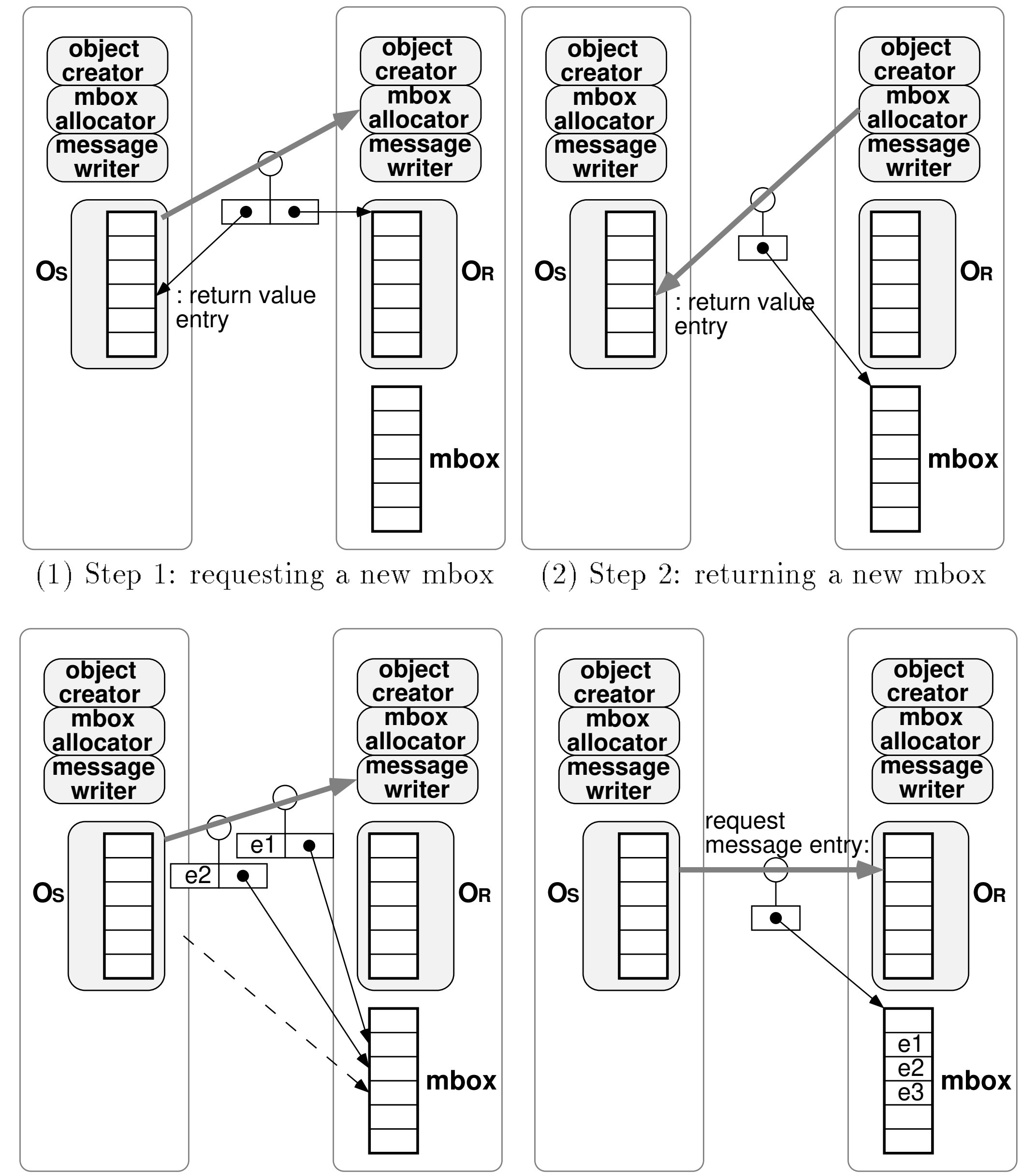
Step 2: A message box is allocated in the receiver node from the freelist by the mbox allocator, and its address is sent to the Return Value Entry of the sender object (Figure 6.5(2)). This packet is matched with the self-sent packet of Step 1, invoking the argument evaluation/sending from the continuation address.

Step 3: As each argument is evaluated, a sequence of special packets containing both the elements of message data and the address within the message box where the data is to be stored, are transmitted from the sender object to the message writer of the receiver node (Figure 6.5(3)). The evaluation of message arguments and their send are pipelined. This process may be interrupted by nested message sends, as is described in Chapter 7. The message writer writes the data directly into the specified address in the message box.

Step 4: Finally, a packet containing the address of the message box is sent to the Request Message Entry of the receiver object (Figure 6.5(4)). As described above, this triggers the execution of appropriate code so that the receiver object can either invoke a specified method or just enqueue the message depending on the object state.

We call the message sends in ABCL/EM-4 the *pipelined sends*, because the evaluation of message arguments and their send are pipelined. The advantages of the pipelined send over other message passing implementation schemes are described in Chapter 7.

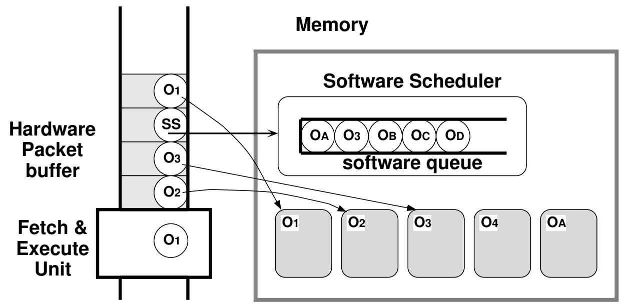
6.4 Scheduling of Objects



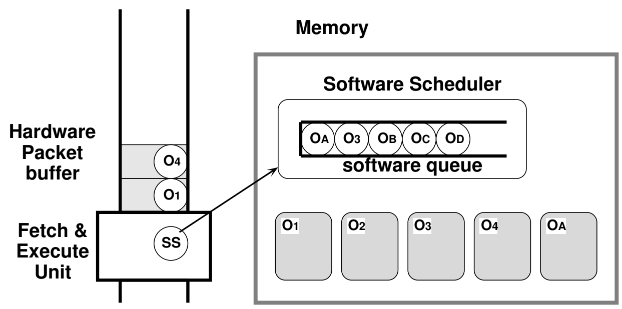
(1) Step 1: requesting a new mbox (2) Step 2: returning a new mbox

(3) Step 3: writing elements into mbox (4) Step 4: sending the mbox address

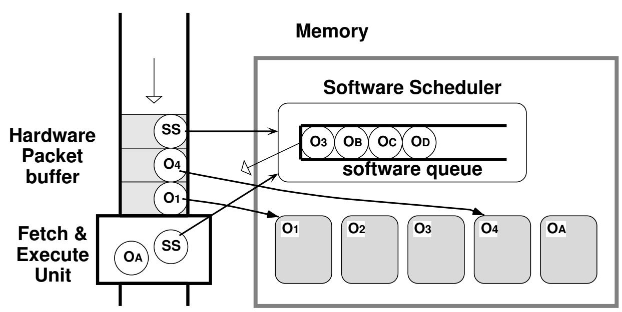
Figure 6.5: The Details of Packet Exchanges for a Message Send



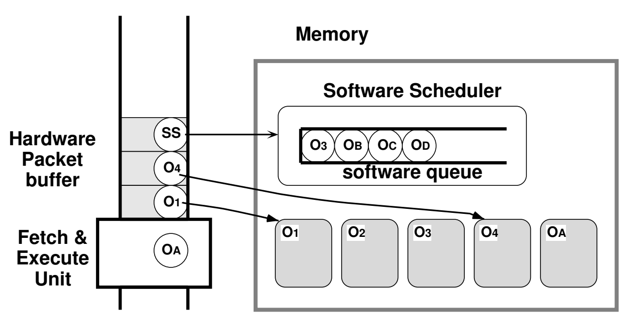
(1) Packets containing the names of Objects and Software Scheduler are in the hardware packet buffer.



(2a) When Software Scheduler gets control,



(2b) Software Scheduler dequeues O_A from the software queue and enqueues itself in the hardware packet buffer,



(2c) and Software Scheduler transfers the control to O_A.

Figure 6.6: The Combination of Hardware Scheduler and Software Scheduler

The scheduling of objects within a node is performed using two types of schedulers. One is the *hardware scheduler*, and the other is the *software scheduler*. Both employ the packet-driven architecture of EM-4.

The hardware scheduler directly utilizes the hardware packet buffer as the scheduling queue. When scheduling of an execution of code is needed, a packet containing a continuation address is sent to one of the object entries, typically a Return Value Entry or the Null Entry. This packet is buffered in the hardware packet buffer, which will eventually be scheduled for execution.

The software scheduler is similar to more conventional schedulers but nevertheless is packet-driven. That is to say, since it too is a code block that must be invoked via the packet-driven mechanism, its address must be enqueued in the hardware packet buffer. When it is scheduled, it re-enqueues itself in the hardware packet buffer for subsequent rescheduling of itself and then dequeues one of the objects it holds in its software scheduling queue and transfers control to the object (Figure 6.6). When an object is to be a subject of software scheduling, the software scheduler enqueues the object in the software scheduling queue and then makes an exit from the SCB, thereby causing a context switching in the hardware scheduler.

The combination of hardware and software schedulers has the following advantages. High priority tasks, such as reception of a reply message, are scheduled by the hardware scheduler for fast dispatching. Lower priority tasks are managed and scheduled by the software scheduler, which (1) prevents the hardware buffer overflow[3], and (2) allows for flexible scheduling policies other than the round-robin.

Note that the software scheduler should be optional, because it consumes the resource of the processing power. (It is small but not zero.) For the program which does not need an elaborated scheduling and hardware queue buffer is enough long for the execution of the program, the only hardware scheduling will work well.

6.5 Storage Management

Garbage collection of objects is performed based on the marking algorithm using virtual global snapshot and object coloring. Garbage collection is under development; but Kamada's GC scheme[15] is the likeliest one to be employed, which reduces the number of marking packets, by using *weights*, to half of the scheme using ACK that I first considered. I have another plan in which the host node that appears in Kamada's GC is eliminated by using *precise weights*.

[3]Although hardware buffer overflow is automatically handled by the EM-4 hardware, it incurs significant overhead because the overflowed elements must be written to/read from the main memory.

Chapter 7

Performance Measurements and Discussions of Basic Operations

7.1 Latency Evaluation – Counters

In order to measure the latency of remote object creation and remote now-type message send, our benchmark program (Figure 7.1) creates objects along a closed Hamilton path among the 80 nodes, and sends messages through the path. When an object O_i accepts a `:create-count` now-type message M_i from O_{i-1}, O_i (1) creates a new object O_{i+1} at the adjacent node along the path, (2) sends a `:create-count` now-type message M_{i+1} to O_{i+1}, and (3) forwards the reply value from O_{i+1} to O_{i-1} in reply to M_i. On the other hand, the `:count` now-type message utilizes the existing objects created by `:create-count`. Since all the messages are sent using now type message passing in this program, they are all serialized. (I.e., only one message at a time is being sent throughout the entire system.)

Table 7.1 shows the results of the benchmark. When the objects are created dynamically using `:create-count` messages, the total execution time is 0.860 ms (10750 clock cycles), or approximately 10.75 μs (134 clock cycles) per object creation followed by now-type message send. When the objects already exist, the total execution time is 0.743 ms (9287 clock cycles), or 9.29 μs (116 clock cycles) per now-type message send.

	creation-request-reply cycles	request-reply cycles
Average execution time per one cycle	10.75 μs (134 clocks)	9.29 μs (116 clocks)
Total execution time for 80 objects	0.860 ms (10750 clocks)	0.743 ms (9287 clocks)

Table 7.1: Execution Time for Remote-Creation/Send/Reply Cycles

```
(deftype (obj-msg Counter-Obj)
         (union [:create-count int (@ int)]
                [:count int (@ int)]))
[class Counter Counter-Obj ()
  (state (Counter-Obj next))
  (script
    (==> [:create-count i]
         (if (= i 0)
             !1      ; returns 1 ('!' is the reply construct)
             (progn
               [next := (new-Counter)] ; A newly created counter object
                                       ; is set to state variable  next
               ![next <== [:create-count (- i 1)]]))) ; Now type message sending
    (==> [:count i]
         (if (= i 0)
             !1
             ![next <== [:create-count (- i 1)]])))] ; employ existing path
[class main (obj []) (int i)
  (let ((Counter-Obj (next (new-Counter))))
    [next <== [:create-count i]]
    [next <== [:count i]]))]
```

Figure 7.1: Message Loops through a Hamilton Path

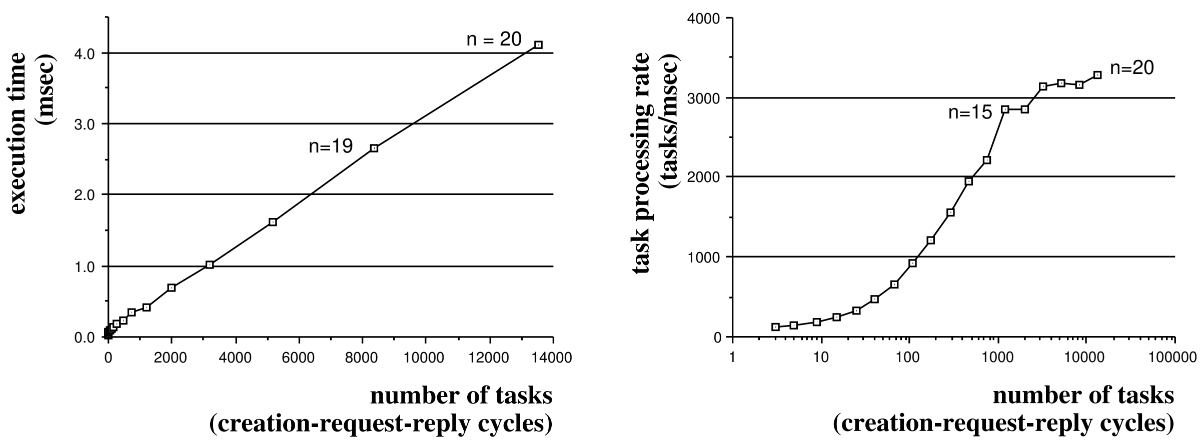
7.2 Throughput Evaluation – Fibonacci Numbers

The previous program only measures the latency of message processing; as a result, there was little concurrency involved. We next measure the number of method invocations possible throughout the entire system per each time unit. The example is a tree computation induced by the calculation of Fibonacci numbers (Figure 7.2). The objects of class `fib` comprise the tree, either becoming an interior node or a leaf node depending on the argument of the message. The leaf node immediately returns a reply message (the value 1). The interior node (1) creates two sibling objects, (2) multicasts the `:calc` message to the siblings using the now type message send, and waits for their reply, then (3) computes the new value returns the result to its parent node. Thus each object is first (a) created, (b) receives a request to calculate and (c) answers a reply. Here, for convenience, we count the above process (a) – (c) as one *task*. (We also make a note that this program is solely for performance measurements only, and by all means NOT the fastest way to compute the Fibonacci number with a COOP language.)

We computed the Fibonacci number for $\mathtt{n} = 3 \ldots 20$ using this program. Figure 7.3(1) shows the execution time of the program versus the total number of tasks processed. The graph indicates that over 13000 tasks are performed in 4.1 ms, or approximately 3 tasks per 1 μs. In order to see whether this is the maximum, we consider the second graph (Figure 7.3(2)) which shows the task processing rate (tasks/ms) against the total number of tasks. As we can see, the processing rate saturates between 1000 to 10000 tasks, or $\mathtt{n} \geq 15$ at the above rate.

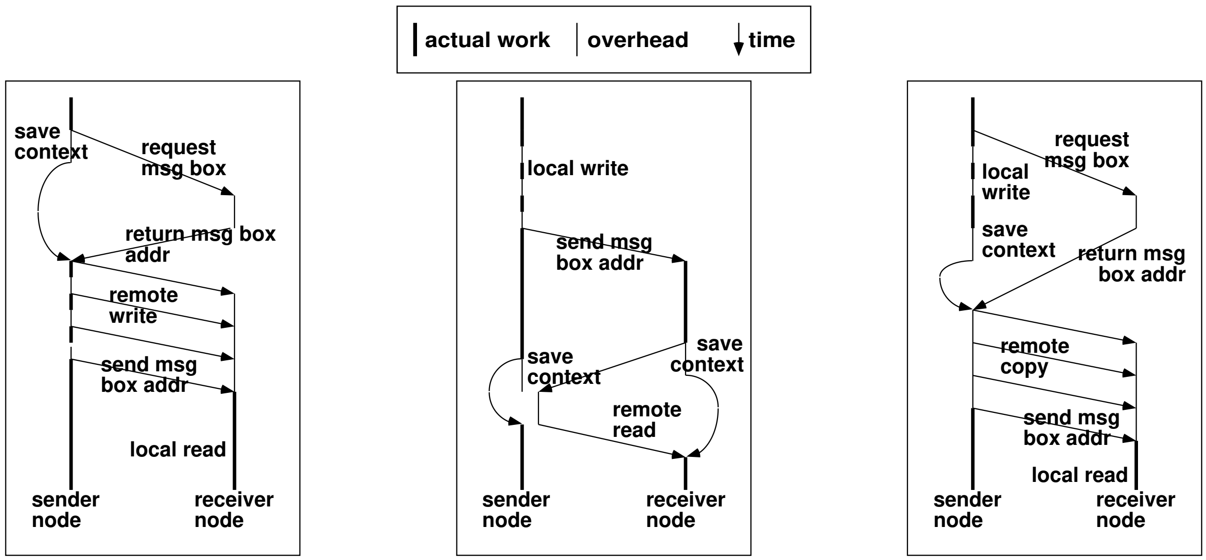
```
(deftype fib-o (obj [:calc n (@ int)]))
[class fib fib-o ()
  (script
   (==> [:calc n]
        (if (<= n 2)
            !1
            !(+ [[(new-fib) (new-fib)]   ; creations of two  fib class objects
                 <==* [[:calc (- n 1)] [:calc (- n 2)]]] ; multicast now type message sends
                ))))]
[class main (obj []) (int n)
  [(new-fib) <== [:calc n]]]
```

Figure 7.2: Simple-Minded Fibonacci Numbers with Object Creation



(1) Execution time (2) Throughput

Figure 7.3: The Result of the Throughput Measuring Program for ABCL/EM-4



(a) Remote-Write Local-Read Scheme

(b) Local-Write Remote-Read Scheme

(c) Local-Write Remote-Copy Local-Read Scheme

Figure 7.4: Typical Message Passing Implementation Schemes

7.3 Discussions on Message Passing Implementation Schemes

There are various ways which implement messages of arbitrary length and they have their own pros and cons. We pick up three typical implementation schemes for message passing protocol in the ABCL/EM-4:

- Remote-write local-read scheme,
- Local-write remote-read scheme,
- Local-write remote-copy local-read scheme.

Figure 7.4 illustrates these schemes. The arrows represent packet communication and thick and thin vertical lines represent actual work and overhead, respectively. First, we examine the Remote-Write Local-Read (RWLR) scheme, which we adopt in the ABCL/EM-4. Then we discuss the Local-Write Remote-Read (LWRR) scheme and the Local-Write Remote-Copy Local-Read (LWRCLR) scheme, respectively comparing them with the RWLR scheme in terms of its IG (ideal gain) to be defined.

7.3.1 Remote-Write Local-Read (RWLR) Scheme

The RWLR scheme seems to be simpler than the other two schemes. On a message send,

(S1) The sender requests a message box on receiver's node;

(S2) The sender saves the context and sends its continuation address to itself, then exits from the current SCB;

(R1) In response to (S1), the receiver allocates a message box on that node and returns the address to the sender;

(S3) The sender resumes the execution by matching of (S2) and (R1) packets, then performs a series of argument evaluations and remote-writes to fill up the message box;

(S4) The sender sends the message box address to the receiver;

(R2) The receiver reads the message contents locally.

The RWLR scheme relies on the FIFO-ness of the network.

7.3.2 Local-Write Remote-Read (LWRR) Scheme

The LWRR scheme is the dual of the RWLR Scheme. On a message send,

(S1) The sender allocates a message box locally;

(S2) The sender performs a series of argument evaluations and local-writes to fill up the message box;

(S3) The sender sends the message box address to the receiver;

Then, with the message box address of (S3), the receiver performs remote-reads to access the message contents. On a remote-read,

(R1) The receiver requests a remote-read to the sender;

(R2) The receiver saves the context and sends continuation address to itself, then exits from the current SCB;

(S4) In response to (R2), the sender returns the message element;

(R3) The receiver resumes the execution by matching of (R2) and (S4) packets.

In Table 7.2(a), advantages and disadvantages of the LWRR scheme are listed compared with the RWLR scheme. The network/buffer costs and PE costs are clock cycles consumed by packet traveling/buffering/handling and instruction execution, respectively. An IG (ideal gain) value is the total clock cycles multiplied by the number of repetitions. We assume that a typical message size (N_M) is 4 and a typical number of remote-reads (N_R) is 3. IG values may not directly appear in the total execution time as real gain, but they can be used, for example, to indicate that advantage 1 and 2 are almost always overwhelmed by disadvantages 4 and 5. Because of this reason, we did not choose the LWRR scheme.

(a) Advantages/Disadvantages of Local-Write Remote-Read Scheme (compared to Remote-Write Local-Read Scheme)					
No.	Advantages(+)/ Disadvantages(−)	Network/Buffer Cost (clocks)	PE Cost (clocks)	multiplied by	Ideal Gain
1	No round-trip latency for allocating a remote message box	$+(12+\alpha)$	$+2$	1	$+14$
2	The sender does not have to exit from the SCB in order to receive the message box address from the receiver node (Upon an exit from an SCB, live values on registers must be saved into memory)	$+(3+\alpha)$	$+(6+\beta)$	1	$+9$
3	Local-write is inexpensive than remote-writes.	$+(3+\alpha)$	$+1$	$N_M(=4)$	$+16$
4	Round trip latency is necessary to remote-read.	$-(12+\alpha)$	-2	$N_R(=3)$	-42
5	At the receiver's node, an exit from an SCB is required to get the result packet of remote-read.	$-(3+\alpha)$	$-(6+\beta)$	$N_R(=3)$	-27

(b) Advantages/Disadvantages of Local-Write Remote-Copy Local-Read Scheme (compared to Remote-Write Local-Read Scheme)					
No.	Advantages(+)/ Disadvantages(−)	Network/Buffer Cost (clocks)	PE Cost (clocks)	multiplied by	Ideal Gain
1	The round trip latency for allocating a remote message box is hidden not only by context saving but also by local-writes.	$+0$	$+0$	1	$+0$
2	The sender has to write all the contents of message into local memory and read from local memory for remote-copy.	-0	-4	$N_M(=4)$	-16

α is the value influenced by system loads.
β is the value determined by the size of context on registers.
N_M is the number of message words (typically 4).
N_R is the number of remote-reads (typically 3), which may be fewer than N_M (I.e., $0 \leq N_R \leq N_M$).

Table 7.2: Estimation of Ideal Gain

7.3.3 Local-Write Remote-Copy Local-Read (LWRCLR) Scheme

The LWRCLR scheme is similar to the RWLR scheme. The only difference is that the sender evaluates its message arguments before getting the address of the message box allocated on the receiver's node. When the sender gets the message box address, it already has all the message contents and simply copies the message to the message box on the receiver's node.

In Table 7.2(b), advantages and disadvantages of the LWRCLR scheme are listed compared with the RWLR scheme. The IG by the latency hiding is zero because it only changes the scheduling order and it does not change the cost. To put it concretely, the latency hiding of LWRCLR is not effective when the other activities on the PE already hide the message box allocating latency. (Moreover, as Hiraki argues in [9], excessive latency hiding may reduce the overall throughput.) On the other hand, the LWRCLR scheme incurs extra costs for local-write/read for all the message elements, consequently reducing the throughput when there are enough activities to prevent PEs from being idle. A finer analysis of this problem should be made using probability/statistics models which reflect parallel features of real machines.

7.4 Discussions and Comparison to Previous Approaches

Our ABCL/EM-4 software/hardware architecture offers substantial performance improvements over the previous implementation architectures of COOP languages, such as the J-Machine[6] and A-NET[3]. Here, we would like to clarify the differences between our approach and the previous approaches, specifically how we reduce the overhead of message sends with appropriate software/hardware architectural combination.

In the previous approaches, the execution time was dominated by the overhead of message processing. To reduce it, we must improve the speed of context switching and message reception/dynamic method invocation.

7.4.1 Overhead Reduction in Packet Dispatch/Context Switching

Much of the low-level provisions for fast packet dispatching and context switching is provided by the EM-4 hardware. In addition, the compiler supports fast context switching by minimizing the context.

Fast Dispatch and (Pipelined) Zero-Overhead Context Switching

The EM-4 hardware features a very low latency from the time the packet is received until the code is actually invoked. As a result, the reactive style of concurrent object-oriented computation can become feasible. A very high performance is realized with the direct matching scheme and the reduction of pipeline stages. Typical latency is approximately

3 clock cycles, or 240 ns. Special packets are processed even faster, with typical pipeline latency of 2 clocks, or 160ns.

EM-4 also features zero-overhead context switching. The context switching occurs when the CPU exits from the SCB and receives a packet. There is no context switching overhead involved owing to *pipeline fusioning* of the *normal dataflow pipeline* and the *strongly connected pipeline*[19].

Flexibility in Amount of the Context to be Preserved

Some machines have (1) provided support for automatic preservation of register contexts, and/or (2) decreased the number of general registers in order to reduce the overhead of context switching. For example, in J-Machine, there are only 4 general registers, which are automatically preserved on certain traps.

By contrast, EM-4 has 10 general registers (16 registers total), allowing most of the computation to be performed in a RISC-like manner. EM-4 does not provide automatic register context preservation; thus, we must resort to software for register preservation. In ABCL/EM-4, however, we can reduce the amount of necessary context information to be preserved to the minimum, since we know when the context switch will occur due to the mutual exclusiveness of the strongly connected mode. The minimum context information needed to be preserved is the address of the object segment and the continuation address within the class-specific text.[1] Whether other registers are saved or not upon an exit from the SCB is decided by the flow analysis performed by the compiler.

7.4.2 Overhead Reduction in Remote Message Passing

As stated earlier, the hardware overhead of packet communication is extremely small in EM-4. However, such low communication latency alone does not reduce the overhead involved in message passings amongst concurrent objects in the remote nodes.

Another major difference between ABCL/EM-4 and previously proposed architecture-supported OO-language implementations such as Concurrent Smalltalk on the J-Machine[12, 11] and A-NET[3] is that we attempt to eliminate the memory read/write/copy overhead of the outgoing and incoming messages as much as possible. That is to say, whether messages will require buffering or not could have significant impact on performance due to overheads such as copying and management of the buffers.

We briefly review the implementations of inter-object message passing described in the previous sections, and explain the rationales behind the design choices made, specifically why our design ideas incur less overhead compared to the previous implementations.

Here is the summary of the message sending activity, the *pipelined send*, of ABCL/EM-4:

- The sender node reserves the message box on the receiver node.

[1] In fact, as we have seen in Chapter 6, these are not only stored in the local store, but may also exist within packets being transmit.

- The sender evaluates the arguments of the message. As each argument is evaluated, its packet is sent to the proper (address) location in the message box in the receiver node. The evaluation and message send are pipelined.

- The message box address is sent to the receiver object. The receiver object may enqueue the message until it can process the message.

By contrast, here is the summary of the message sending activity of the J-machine:

- The `SEND` instruction loads message words into the FIFO message buffer. When the message is complete (indicated by the `SENDE` instruction) or the buffer is full, the message is sent to a specified node.

- When the message reaches the receiver node, it is automatically enqueued in one of the two circular buffers (one for each priority level) in that node. These circular buffers are also used as scheduling queues.

Given the above characterizations, we now discuss the advantages of our scheme.

Efficient Management of Message Buffers with Freelists

In our scheme, the message buffers are managed with the freelist. The message boxes are efficiently allocated from/deallocated to the freelist. Recall that the size of message boxes is fixed. This of course incurs storage overhead — a space/speed tradeoff. In practice, all is not wasted, since the unused portion of a message box may be used for temporary variables.

Compared to using the hardware circular buffer as was adopted by the J-machine, our scheme using the freelist has a great advantage that *copying of queued messages is unnecessary.* By contrast, for hardware circular buffer, the message data will have to be copied and removed from the circular buffer into some external buffer in the following cases:

- *When scheduling other than first-come-first-serve is preferable.* In many classes of problems, such as discrete event simulation algorithms[14] and search problems, proper scheduling of message processing is essential for good performance.

- *When an object cannot process the received message immediately.* This happens when the object is waiting for the reply message of the previously sent request message. Consequently, the messages which are scheduled before the reception of the reply message must be copied from the circular buffer. For real complex parallel programs, this is often the case.

Reservation of Message Boxes

In our scheme, the sender reserves the message box prior to the actual sending of the message data. Although the packet transmissions to allocate a message box may result in some overhead, the turnaround overhead is not significant owing to the fast network architecture.

Without such reservation, the receiver node has to allocate the message buffer as fast as the network speed. Usually this requires some special hardware to allocate the message buffer, and it is difficult for such hardware to realize the complex allocation scheme such as one using the freelist. In our scheme with the reservation, allocation can be done in software, which performs the complex allocation without such hardware.

The reservation is useful not only for receiving the message but also for sending the message. That is to say, since the sender node can obtain the address of message box prior to the actual sending of the message data, *the per-packet, pipelined and interruptible sending* of the message data becomes possible as is described next.

Sending of Message Elements with Per-Packet Communication (Pipelined Send)

EM-4 allows per-packet specification of the address to which the packet data is to be written. This property enables the following *pipelined send*: Once the sender gets the address of the message box, it can emit the packets as the message expressions are evaluated. As a result, (1) the latency of sending of actual message data can be reduced because most of the message elements can be sent into the network before the sender node gets the complete elements of the message, and (2) message elements can be sent in an arbitrary order at arbitrary times; in fact, sending of a single message (consisting of multiple packets) could be interrupted in its midst, and interleaved with other message sends.

On the other hand, in some architectures such as the J-Machine, since the location where the message will be stored is not determined upon its sending, once the receiver node is specified, the rest of the message elements must be sent continuously. This will reduce the size of messages, but has the drawback that the message sending cannot be interrupted.[2]

Our claim is that, non-interruptible messages decreases throughput by requiring local memory accesses. Let us consider the following example:

```
[ObjA <= [:MsgA (- n 1) (Fun x y) [ObjB <== [:MsgB]]]]
```

The above message sending expression contains a function call and an embedded message sending expression as arguments. In ABCL/EM-4, as soon as each argument is evaluated, it is immediately sent to the node of the target object (*ObjA*). The evaluation of the second argument (function call) and the third argument (message send to *ObjB*) could or will effectively interrupt the sending of the outer message, but this will not incur any overhead as explained above. By contrast, in the J-machine, the arguments must be saved (or, buffered) into memory as they are evaluated, because of possible interruption.

[2] It may seem that one could easily simulate the former with the latter; however, in the J-machine, a message is received by memory (circular buffer) not registers.

At the end of the evaluations of all the arguments, the results must be re-read from memory when they are actually sent. Altogether, the benefits of direct message sending from the hardware processor would be lost here; and extensive flow analysis performed by the compiler will only serve as a minor solution.

7.4.3 Comparison to Implementation on Stock Multicomputer

There is another implementation work of ABCL, ABCL/onAP1000[24, 35], on a multicomputer based on conventional SPARC chips (Fujitsu AP1000[13]).

In ABCL/onAP1000, an *active message*[25] is used for a remote message send, which is implemented by preparing a collection of message handlers and placing the first instruction address in the message when sending the message. Each handler code is specialized for the individual message format, eliminating the runtime check of the message format.

In contrast to the active message on AP1000, our runtime architecture uses the pipelined send; the advantages of the pipelined send over the active message on AP1000 are: (1) local buffering is not needed at the sender node, (sending memory buffer is not used in ABCL/onAP1000, but the message contents should be kept as local context, usually on stack (some of them might be in registers, but storing into memory is required on spilling), until the message evaluation is completed), (2) copying of all message contents is not necessary when enqueuing the message, (3) the latency is reduced by starting the message send before the completion of evaluation of the message arguments, and (4) structural data can be sent easily by reservation, while the disadvantage is only the pre-allocation cost of message boxes is necessary. Note that the advantage (2) would be eliminated by active message supported by hardware register reception[25].[3]

In ABCL/onAP1000, *stack based scheduling*[24] is introduced to reduce the overhead of scheduling queue manipulation as much as possible. On the other hand, in ABCL/EM-4, the overhead of scheduling queue manipulation is eliminated by using the hardware packet queue buffer (At this point, EM-4 belongs to the group of multithreaded architectures).

The stack based scheduling also increases the register utilization effects in some circumstance because it locates short live-range data (message) on registers by coupling the scheduling with the message send. (This scheme shares the same philosophy as the LIU concept described in Section 8.5.2.) However, it only works in the case where the message passing is local (namely, to the same node) and the receiver object is in the dormant reactive state (namely, blocking and waiting for a request message in dormant mode). On the other hand, the design of ABCL/EM-4 runtime architecture tries to exploit concurrency, where the remote message passing is preferable and, in the situation that the message is considerably large, becomes even faster by using two processors (namely the sender node and the receiver node) in parallel. In ABCL/EM-4, a remote/local check of target node is omitted; the check is costly, compared to the remote message passing speed of ABCL/EM-4, because it needs several logic operations to test the node number (which

[3] RWC-1, which is scheduled to build by Real World Computing Partnership by 1996, supports the message reception with hardware registers.

is encoded in one word with a local address for an object ID) and a conditional branch (totally 4 instructions + 1 delay slot is required).

7.4.4 Comparison to Implementation of Procedural Languages

There is another language implementation work in progress on EM-4[21]; there, a superset of the C language, EM-C, augmented with message passing libraries is being developed. We will do a brief comparison and argue why our approach is faster.

Aside from the fact that procedural languages require extensive flow analysis to extract the inherent parallelism, library-based message sending has a drawback of requiring extraneous memory accessing: according to the language definition of C, a function call is call-by-value-based, i.e., all the arguments must be evaluated before invoking the function. This requires that the evaluated arguments to be stored into memory, then re-read in order to perform a message send. This nullifies the benefit of being able to perform fast message passing directly from the hardware processor.

Recently, EM-C provides more elaborate language features, in which *global distributed address space* is provided[20]. By exploiting this functionality, EM-C can be used as a *back end* of our language to realize the pipelined send. However, the shared memory semantics prevents us from exploiting new register allocation schemes describes in Sections 8.5.1 and 8.5.3.

7.4.5 Summary of Comparison to Previous Approaches

Our results in Sections 7.1 and 7.2 indicate that, concurrent message processing for concurrent objects can be significantly improved over the previous approaches, and as a result, the COOP languages could serve well in a highly parallel environment. We believe that this claim holds even for the small examples in this thesis. For example, by the numbers given in [11], the total number of machine cycles for object creation followed by a message send and its reply is estimated to be well over 200 cycles for the J-Machine, whereas it is approximately 134 cycles in our case. And, for the reasons we gave in this chapter, we further believe that the difference in actual throughput should increase in even larger, real-life applications.

The principle behind our software/hardware architecture is very similar to that of the current RISC technology — rather than to provide very complex hardware for message passing that could take hundreds of machine cycles with microcodes, we base ourselves on a low-level reactive architecture with packet latency and throughput comparable to intra-node processing, and then build an appropriate software architecture with language and compiler support. Architectures such as J-Machine share our views in many ways, but as discussed above, our software/hardware architecture is cleaner/faster by comparison, allowing us to avoid much of memory operations, which results in considerable overhead when the latency of hardware packet architecture is minimized.

Chapter 8

The Compiler for EM-4

8.1 Overview of Compiler

The compiler of ABCL/EM-4 translates a program written in the source language ABCL/ST into the corresponding target code.[1] Since the abstraction levels of the source language and the target code quite differ, a direct compilation is difficult. This difficulty is overcome by introducing several language layers; each lower level language implements the functionalities of its one-level higher language. Using such layered approaches also provides portability of the compiler; changing the lower level part is only necessary to port it on other target machine architectures.

8.1.1 The Language Hierarchy

The compiler adopts the following language hierarchy:

1. Source Language
2. High-Level Language
3. Middle-Level Language
4. Low-Level Language
5. I-Language
6. Assembly Language

The *source language* is ABCL/ST, a statically-typed concurrent object-oriented programming language. All the languages from *high-level* to *low-level* and *I-language* roughly correspond to what are called *intermediate languages* in a traditional simpler compiler (such as C, Pascal), where the number of intermediate languages may be fewer than ours

[1]The target code is the most important component of the runtime architecture for high-performance as is described in Chapter 3.

because the abstraction level of source language is not so far from that of the target language.

The brief descriptions of the rest of the languages are as follows:

- The *high-level language* is at almost the same level as the source language, and has the same subtype relation and recursive types. The program of the *high-level language* is *high-level code*, which embodies the semantics of the source program in terms of sequential execution with a single thread of control. The high-level code is in *plan-do* style; this new feature is essential in realizing our runtime architecture for EM-4 (especially for pipelined sends) without loss of machine independence of high-level code. High-level code is a sequence of instructions, variable declaration statements, and *plan declaration statements*; the lower part of the compiler may generate eager data transfer code by interpreting the plan declaration.

- The *middle-level language* is roughly at the level of C language; but the *middle-level code* is also a sequence of instructions and variable declaration statements. The language has *pointers* which can be used to implement subtype relations and recursive types in the higher level language; i.e., higher level values are also translated into lower level ones including pointers. Pointers are also used to implement message passings. Pointers (and other values) are associated with *middle-level types*, and field selection of boxed value is performed with a field number rather than an offset in memory address space. (They are different because values may vary in size.)

- The *low-level language* introduces the notion of data size, and pointers are used in the sense of memory addresses. The *low-level code* is also a sequence of instructions and variable declaration statements which are similar to the middle-level ones but more low-level. Data in the low-level language are still augmented with *low-level types*, which are quite simple and consist of only basic types and no structural ones. The low-level code is responsible for actual runtime data representation, which accommodates an interface and restrictions of the heap management system which may manage the heap as fixed-size memory boxes, where a large structure may be represented by a tree of memory boxes.

- The *I-language* is almost identical to the low-level language; but opcodes of *I-code* instructions are specialized for the corresponding low-level types of its operands (like an assembly language). Conventional optimization schemes, such as dataflow analysis, are performed on the I-code. The I-code is also used for storage allocation and register allocation in the code generation phases. For this reason, the I-language is slightly influenced by the assembly language; for example, a real-number comparison-branch instruction in low-level language is translated into two instructions in I-language (a real-number comparison and an integer comparison-branch instruction) for the assembly code generator to be able to insert the register save/load/move instructions just before the branch instruction.

Table 8.1: Dependency between Translators and Runtime Architecture

<table>
<tr><th></th><th>Code-Conversion</th><th>(Data-Conversion)</th></tr>
<tr><td>Source Language → High-Level Language</td><td colspan="2">Independent</td></tr>
<tr><td>High-Level Language → Middle-Level Language</td><td rowspan="2">Dependent</td><td>Partially Independent</td></tr>
<tr><td>Middle-Level Language → Low-Level Language</td><td></td></tr>
</table>

- The *assembly language* is the target language of our compiler. The rest of conversion is performed by the *EM-4 assembler* developed at ETL.

All the languages from *high-level* to *low-level* are independent of the assembly language (instruction set of the target machine architecture), while some of the languages and some of the converters (i.e., fragments of the compiler) between them depend on a runtime architecture tuned for the target machine. The dependency is summarized in Table 8.1. The main part of data conversion from high-level to middle-level is independent of the underlying runtime architecture. This part realizes the subtype relation in the high-level language by using middle-level data structures. The details are described in Section 8.2.

The code conversion from high-level to middle-level is not only dependent on the underlying runtime architecture, but also is the essential part of our compiler to realize our high performance runtime architecture for EM-4 (especially for pipelined sends). Details of the conversion are described in Section 8.4.

8.1.2 Compilation Phases

In this section, we outline the compilation phases. Figure 8.1 illustrates the phases:

1. First, a source program is parsed; the *parser* generates a *high-level parse tree* as a result. The parser is NOT the main target of development of our compiler; indeed, we just adopt *S-expression* as a basic syntax of the source language, with which the parsing is quite simple. An additional front end which adopts a more elaborate syntax can easily be realized with familiar tools, such as *lex* and *yacc*.

2. Type checking is performed on the high-level parse tree; as a result, types are assigned to all variables and expressions. The details of our type system and type checking algorithm are described in Chapter 5.

3. The type-checked high-level parse tree is converted into the *high-level code*. As is described in the previous section, the high-level code is *plan-do* style code. The *coder* may introduce an adequate number of plans with destinations, temporary variables, and jump-labels to convert a tree into sequential code. The compilation of *pattern matching* into the *table look-up* code is also performed via this conversion. The details of the high-level code are described in Section 8.3.

4. The high-level code is translated into the *middle-level code*. As is described earlier, the data translation realizes the subtype relation in high-level language using

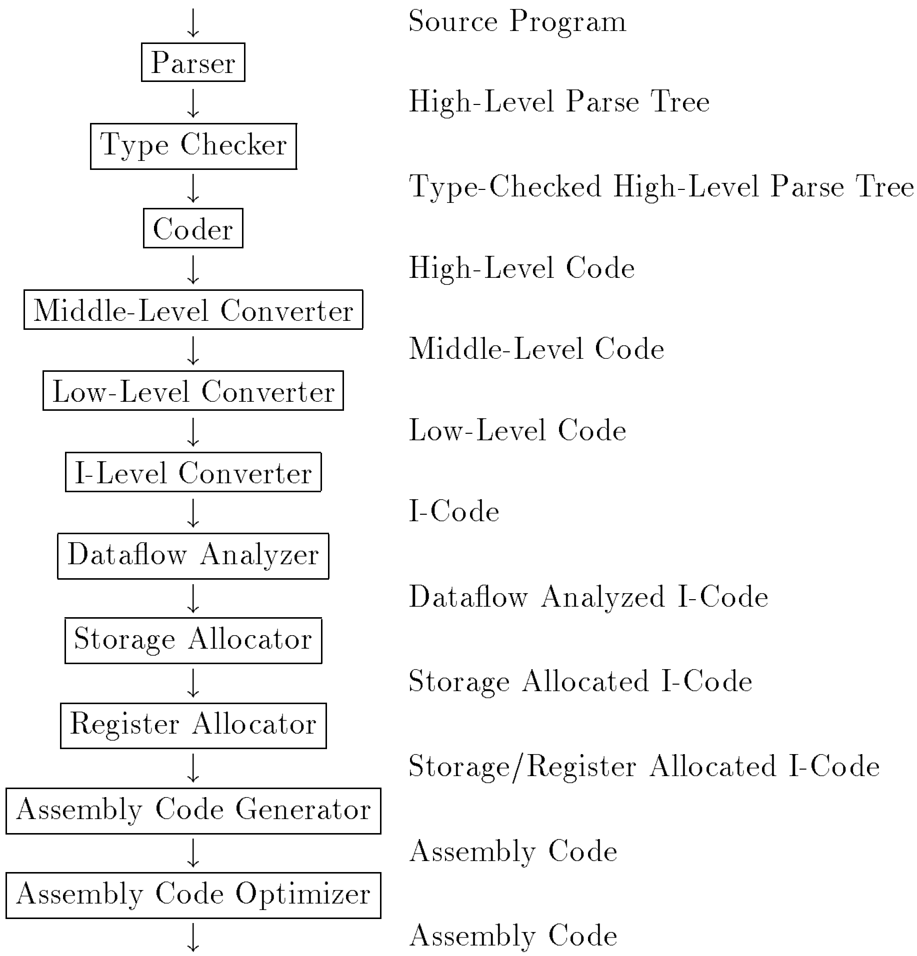


Figure 8.1: Compilation Phases

middle-level data structure, and the *middle-level converter* implements our high performance runtime architecture for EM-4 (especially for pipelined sends) by interpreting the high-level plan-do style code. The principle of plan-do style is that *the declaration of an action (plan) prior to its actual execution (do) provides the advanced (future dataflow) information.* This information helps the middle-level converter choose the middle-level code to implement the high-level semantics. The details of the data translation and code translation are described in Section 8.2 and Section 8.4, respectively.

5. The middle-level code is translated into the *low-level code*. The *low-level converter* maps middle-level data representation onto low-level data representation which is the actual runtime data representation based on the interface with the heap management system.

6. The low-level code is translated into *I-code*.

7. The *dataflow analyzer* examines the I-code and produces lifetime information of the variables.

8. Storages are allocated to variables. The *storage allocator* tries to compact the storage area as much as possible by means of so called *graph coloring* using lifetime information provided by the dataflow analysis. A storage allocated for a variable is used as a *home position* of the variable. Note that some home positions are previously allocated by the low-level converter, in order to minimize the amount of register context (See Section 8.5.1).

9. Registers are allocated to variables. The *register allocator* tries to allocate as many registers to variables as possible, using lifetime information and register constraint. Values spilt from registers are kept by the corresponding home positions. We do not employ graph coloring for register allocation; we rather use a more optimal algorithm, where the spilled value is what is used farthest in the future. Let us call it the *Least Immediately Used* (LIU) algorithm by analogy with *Least Recently Used* (LRU) algorithm. We will present the details in Section 8.5.

10. The *assembly code generator* produces *assembly code* from the storage/register allocated I-code. This translation is easy because the EM-4 processor has RISC-like instruction set, where an I-code instruction simply corresponds to a sequence of assembly code instructions.

11. The *assembly code optimizer* performs peephole optimization on the EM-4 assembly code, which includes: (1) elimination of redundant 'nop' instruction, (2) long/short branch optimization, etc.

8.2 Realizing Subtype Relation with Small Overhead

The high-level datum has a mapping to the corresponding middle-level data structure. In this section, we propose the mapping which realizes the subtype relation between high level data with small overhead.

A naive implementation of subtype relation may use large tables whose size is equal to the number of all sorts of injection (or message or method) which appear in the source program. Let N be the number of sorts. On a look-up, it needs only $O(1)$ time complexity but it needs $O(N)$ space complexity. Another naive implementation may use an associative list or a binary search algorithm; they would reduce the space complexity to $O(M)$ (Let M be the number of injection tags which may appear in the currently examined disjoint-union type value), but their time complexity would increase to $O(N)$ or $O(\log N)$. Hash search algorithms may improve the time complexity to $O(1)$; but its coefficient is still large.

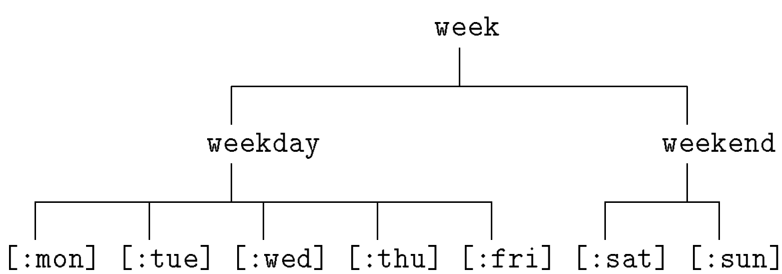
We propose an implementation scheme where the time complexity of a look-up is $O(1)$ and the space complexity is slightly more than $O(M)$ (the precise number is given by function *maxidx* described later), where the look-up is performed as a very fast operation, *direct indexing operation* of table.

8.2.1 Representation of Disjoint-Union Type

First, we examine the following disjoint-union types:

```
(deftype weekday (union [:mon] [:tue] [:wed] [:thu] [:fri]))
(deftype weekend (union [:sat] [:sun]))
(deftype week (union weekday weekend))
```

The subtype relation among them can be represented as the following tree:



In the above figure, each type is a subtype of its connecting upper ones; for example, `[:mon]` is a type which has only one element `[:mon]`, and a subtype of `weekday` and `week`.

Let `xw` be a variable of type `week`, `xwd` be a variable of type `weekday`, and `xwe` be a variable of type `weekend`. `xwd` may be used for *pattern matching* in the following expression:

```
(match xwd
  (=> [:mon] ...)
  (=> [:tue] ...)
  (=> [:wed] ...)
  (=> [:thu] ...)
  (=> [:fri] ...))
```

To realize the pattern matching, the value of `xwd` must be distinguished for each keyword. We use an *index* for this purpose. The domain of an index is a contiguous subset of non-negative integer; it is from 0 to n for a non-negative integer n. For `xwd`, the index ranges from 0 to 4, and '0' represents `[:mon]`, '1' represents `[:tue]` and so on. Similarly, for `xwe`, the index ranges from 0 to 1, and '0' represents `[:sat]`, '1' represents `:sun`. However, for `xw`, it would range from 0 to 6, and '5' would represent `[:sat]`, which is represented by '0' in the case of `xwe`. Thus, we can see a problem that the representation of `[:sat]` differs between `xw` and `xwe`. To solve this problem, when we perform an assignment:

```
[xw := xwe]
```

the index value must be shifted up (i.e., $0 \mapsto 5$, $1 \mapsto 6$). This can be performed by a quite simple operation which just *adds* 5 to the index value. We call this adding value (i.e., 5) *index adjustment value.*

This mechanism can be generalized: a disjoint-union type is either a disjoint-union type symbol (denoted by s) or a keyword type (denoted by $k\ \tau$). The index range of a disjoint-union type symbol s is constructed by concatenating all the ranges of its union elements (used by union operation) $\{u_1, u_2, \ldots, u_n\}(= F_s(s))$, where the *order* of the union elements must be consistent everywhere in the language system; the order for union operation in the source program specified by the programmer is used by our current compiler. On the other hand, the index range of a keyword type is just from 0 to 0. The number of elements in an index domain is given by the following function which maps a disjoint-union type to an integer:

$$\begin{aligned} maxidx(s) &= \sum_{u' \in F_s(s)} maxidx(u') \\ maxidx(k\ \tau) &= 1 \end{aligned}$$

This function gives the size of look-up table, which is the space complexity of the look-up (pattern matching) operation.

Now, the index adjustment value, when we use a value of a disjoint union type (a source type) as a value of its supertype (a target disjoint-union type), is given by the function *index-diff* defined below, which is a multi-valued function (i.e., returns multiple integers) from a pair of $\langle$*source disjoint-union type, target disjoint-union type*$\rangle$. *index-diff* is derived by the following rules:

$$\textit{index-diff}(u, u) = 0.$$

$$\textit{index-diff}(u_s, u_j) = i, F_s(s_d) = \{u_1, u_2, \ldots, u_n\} \Longrightarrow$$
$$\textit{index-diff}(u_s, s_d) = i + \sum_{k=1}^{j-1} maxidx(u_k) \quad (1 \le j \le n).$$

index-diff returns multiple values only when the source disjoint-union type appears more than once during the decomposition of the target disjoint-union type, where decomposition means recursive application of F_s to disjoint-union type symbols started from the target disjoint-union type. Any of the returned multiple values can be used as the index adjustment value. The compiler may choose the smallest one, because some machines (including EM-4) have quick addition instruction for small integers.

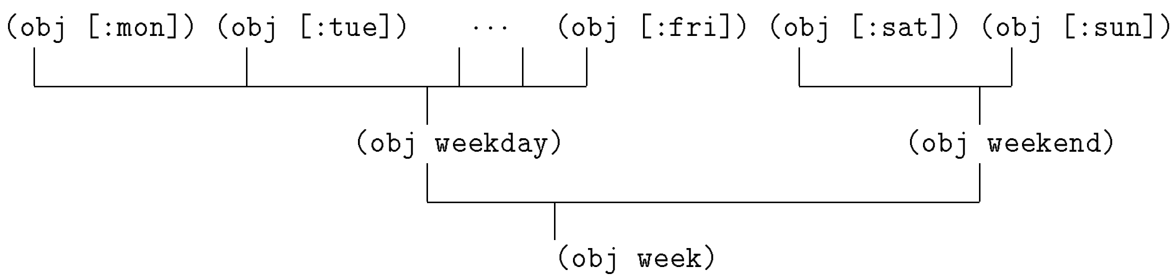
When the compiler generates a look-up table for pattern matching (by case expression) of disjoint-union type data, the multiple integers returned by *index-diff* are also used to determine the entry locations on the table for each pattern. (There might be multiple table entries for a single pattern.)

So far, we have examined indices, but the value of disjoint-union type has a carried value other than the index. We introduce *pointers* to hold the rest of the value; i.e., a disjoint-union type datum is represented by a pair $\langle$*pointer, index*$\rangle$. The indirection by pointer is required for standardizing the representation, because the rest of the data may vary (in the above example, all the carried values are just 0-tuple, but it is not the general case).

8.2.2 Representation of Object Type

In this section, we discuss the representation of object types. An object id, whose type does not have subtype relation, is just represented by a single object id; but an object id, whose type has subtype relation with others, should be represented in a more elaborate way.

The subtype relation among object types is reverse relation of their message types. For example, subtype relation among `(obj [:mon])`, `(obj [:tue])`, `(obj [:wed])`, `(obj [:thu])`, `(obj [:fri])`, `(obj [:sat])`, `(obj [:sun])`, `(obj weekday)`, `(obj weekend)`, and `(obj week)` is as follows:



Let `osun` be a variable of type `(obj [:sun])`, `owd` be a variable of type `(obj weekday)`, `owe` be a variable of type `(obj weekend)`, and `ow` be a variable of type `(obj week)`. The object to which `ow` is bound may perform the following pattern matching:

```
(wait-for
  (=> [:mon] ...)
  (=> [:tue] ...)
  (=> [:wed] ...)
```

```
  (=> [:thu] ...)
  (=> [:fri] ...)
  (=> [:sat] ...)
  (=> [:sun] ...))
```

Since the type of `ow` is a subtype of the type of `owe`, the following assignment is permitted:

```
[owe := ow]
```

After this assignment, the object to which `owe` is bound is equal to the one to which `ow` is bound, and the following message send is also permitted:

```
[owe <= [:sun]]
```

The index of `[:sun]` is 1, because `owe`'s type is `(obj weekend)` and its message type should be `weekend`. However, the required index value is 6, because the actual object receives the message as `week` type.

To solve this problem, the index adjustment is necessary before the message send. Generally, this conversion can be performed with a *type conversion (identity) function*; the function may be denoted by `fn x => x : weekend -> week` in ML.[2] The function is with the object id and is applied to the message before an actual message send. Thus the representation of a high-level object id can be a pair $\langle$*low-level object id, type conversion function*$\rangle$.

Moreover, the type of `owe` is a subtype of the type of `osun`; the following assignments and message send are permitted:

```
[owe := ow]
[osun := owe]
[osun <= [:sun]]
```

In this case, the adjustment should be performed by a composition function of (1)`id: [:sun] -> weekend` and (2) `id :weekend -> week`; (2) is a part of `owe`. Thus, for `[osun := owe]`, the type conversion function for `osun` should be newly composed of (1) and (2).

If the type conversion function is represented by a usual function, such as code block location, each message send will suffer an extra overhead of function invocation. To avoid this overhead, we would like the function to be represented by (a set of) index adjustment value(s); this is possible, if the following condition is satisfied (Let τ_s be the input type of the type conversion function and τ_d be the output type of the type conversion function):

> The source of the subtype relation between τ_s and τ_d includes only the subtype relation between disjoint-union types, where, by "source", we mean the set of applied axioms to prove the subtype relation between τ_s and τ_d (in short, the leaf set of the proof tree).

[2] To put it more precisely, the actual target type of the type conversion function may be a supertype of `week`. But we suppose that the type is `week` in this section for simplicity.

$$\begin{array}{rcl} \langle \mathit{MLrepr} \rangle & ::= & (\langle \mathit{MLtype} \rangle, \ldots, \langle \mathit{MLtype} \rangle) \mid [\langle \mathit{MLrepr} \rangle, \ldots, \langle \mathit{MLrepr} \rangle] \\ \langle \mathit{MLtype} \rangle & ::= & \langle \mathit{MLb} \rangle \mid \mathbf{MLidx} \mid \mathbf{MLuptr} \mid \mathbf{MLkptr}(\langle \mathit{MLrepr} \rangle) \mid \langle \mathit{MLimptype} \rangle \\ \langle \mathit{MLb} \rangle & ::= & \mathbf{MLint} \mid \mathbf{MLreal} \mid \mathbf{MLbool} \end{array}$$

Figure 8.2: Syntax of Middle-Level Type

Examples are `[:sun] -> weekend`, `weekend -> week`, `[[:sun] weekend] -> [week week]`, etc. This representation also improves the speed of composition of functions significantly; for example, the composition of (1)`id: [:sun] -> weekend` and (2) `id :weekend -> week` can be performed just as an integer addition of their index adjustment values. For this reason of efficiency, as is described in Chapter 5, we restrict the application of coercing rules, such as the one between `int` and `real`.

Next, let us consider types `(obj (obj week))` and `(obj (obj weekend))`. `(obj (obj weekend))` is a subtype of `(obj (obj week))`. We need a new adjustment value to adjust index adjustment value; it may be called *"index adjustment value" adjustment value*. But we can see that all these adjustment values and index values are non-negative (small) integers; we use a middle-level data type, **MLidx**, as a type of variables to hold these values.

8.2.3 General Representation

The previous two sections described how to use indices to implement the subtype relation in high-level language. This section gives the general representation of high-level datum by middle-level data structure.

The middle-level type, $\langle \mathit{MLtype} \rangle$, is given by the syntax in Figure 8.2

A middle-level representation, $\langle \mathit{MLrepr} \rangle$, represents a high-level type. The representation uses two kinds of parenthesis [] and (). Using two kinds of parenthesis is just to make it easier for middle-level code to implement high-level field selection. A representation parenthesized by [] denotes a high-level tuple type. On the other hand, a representation parenthesized by () denotes a high-level type other than tuple type, whose value can not be decomposed in terms of high-level field selection.

$\langle \mathit{MLimptype} \rangle$ is not specified above, because it is not essential to implement subtype relation but is essential to implement message passings for the underlying runtime architecture; thus, the complete definition may vary in different implementation schemes for message passings. $\langle \mathit{MLimptype} \rangle$ for our runtime architecture for EM-4 is described in Section 8.4.

The following function *Repr* maps the high-level type onto middle-level representation:

$$
\begin{array}{rcl}
Repr(\top) & = & () \\
Repr(\bot) & = & \text{no such case} \\
Repr(\mathbf{int}) & = & (\mathbf{MLint}) \\
Repr(\mathbf{real}) & = & (\mathbf{MLreal}) \\
Repr(\mathbf{bool}) & = & (\mathbf{MLbool}) \\
Repr(c) & = & \mathit{IMPoptr}(\tau) \\
Repr(s) & = & (\mathbf{MLuptr}, \mathbf{MLidx}) \\
Repr(k\ \tau) & = & (\mathbf{MLkptr}(Repr(\tau)), \mathbf{MLidx}) \\
Repr((\tau_1, \ldots, \tau_n)) & = & [Repr(\tau_1), \ldots, Repr(\tau_n)] \\
Repr(\mathbf{obj}\ \tau) & = & \mathit{IMPoptr}(\tau) +\!\!+ \mathit{Idxlist}(\mathit{Getnum\text{-}idx}(\tau)) \\
Repr(@\ \tau) & = & \mathit{IMPrptr}(\tau) +\!\!+ \mathit{Idxlist}(\mathit{Getnum\text{-}idx}(\tau)) \\
Repr(\mathbf{qpos}\ \tau) & = & \mathit{IMPqpos}(\tau)
\end{array}
$$

where $+\!\!+$ is an infix operator which concatenates two middle-level representations parenthesized by $()$. **qpos** τ is a type of pointers to messages of type τ in the object's message queue. This type is used to implement `wait-for` and `script` commands as is described in Section 8.3.4.

For the same reason as $\langle$*MLimptype*$\rangle$, the external functions, *IMPoptr*, *IMPrptr*, and *IMPqpos*, also depend on the implementation scheme for message passings; their definition is deferred to Section 8.4.

Idxlist(n) returns a middle-level representation parenthesized by $()$ which includes n middle-level types of **MLidx**:

$$
\begin{array}{rcl}
\mathit{Idxlist}(0) & = & () \\
\mathit{Idxlist}(n) & = & (\mathbf{MLidx}) +\!\!+ \mathit{Idxlist}(n-1)
\end{array}
$$

Getnum-idx(τ) returns the number of adjustment values which are necessary to convert τ into its possible supertype. For this reason, the number returned by *Getnum-idx*(τ) is equal to the number of **MLidx**s which appear in *Repr*(τ) but not in the part generated by the external functions *IMPoptr*, *IMPrptr*, and *IMPqpos*. *Getnum-idx* is defined as:

$$
\begin{array}{rcl}
\mathit{Getnum\text{-}idx}(\top) & = & 0 \\
\mathit{Getnum\text{-}idx}(b) & = & 0 \\
\mathit{Getnum\text{-}idx}(c) & = & 0 \\
\mathit{Getnum\text{-}idx}(s) & = & 1 \\
\mathit{Getnum\text{-}idx}(k\ \tau) & = & 1 + \mathit{Getnum\text{-}idx}(\tau) \\
\mathit{Getnum\text{-}idx}((\tau_1, \ldots, \tau_n)) & = & \sum_{i=1}^{n} \mathit{Getnum\text{-}idx}(\tau_i) \\
\mathit{Getnum\text{-}idx}(\mathbf{obj}\ \tau) & = & \mathit{Getnum\text{-}idx}(\tau) \\
\mathit{Getnum\text{-}idx}(@\ \tau) & = & \mathit{Getnum\text{-}idx}(\tau)
\end{array}
$$

The middle-level representation given by *Repr* is sightly redundant because some indices are constant; indices for keyword types are always 0. The following function

Arepr improves *Repr* at this point and can be used instead of *Repr*:

$$
\begin{array}{rcl}
Arepr(\top) & = & () \\
Arepr(\mathbf{int}) & = & (\mathbf{MLint}) \\
Arepr(\mathbf{real}) & = & (\mathbf{MLreal}) \\
Arepr(\mathbf{bool}) & = & (\mathbf{MLbool}) \\
Arepr(c) & = & \mathit{IMPoptr}(\tau) \\
Arepr(s) & = & (\mathbf{MLuptr}, \mathbf{MLidx}) \\
Arepr(k\ \tau) & = & (\mathbf{MLkptr}(Arepr(\tau))) \\
Arepr((\tau_1, \ldots, \tau_n)) & = & [Arepr(\tau_1), \ldots, Arepr(\tau_n)] \\
Arepr(\mathbf{obj}\ \tau) & = & \mathit{IMPoptr}(\tau) + + \mathit{Idxlist}(\mathit{Getnum\text{-}idx}(\tau)) \\
Arepr(@\ \tau) & = & \mathit{IMPrptr}(\tau) + + \mathit{Idxlist}(\mathit{Getnum\text{-}idx}(\tau)) \\
Arepr(\mathbf{qpos}\ \tau) & = & \mathit{IMPqpos}(\tau)
\end{array}
$$

8.3 Converting into Sequential Code

This section briefly describes the high-level code. The high-level code is machine independent high-level sequential code, which gives the high-level semantics to the program in the sense of sequential execution with a single thread of control.

Rather than a direct translation from the high-level parse tree into the middle-level code, we divide the translation by interposing the high-level code. The conversion from the tree into the high-level code is a *common part* of different translations from the tree into the middle-level sequential code, including: (1) introducing labels for conditionals, (2) compacting pattern matching, etc. This separation increases the modularity of the compiler.

8.3.1 Plan-Do Style Code

The high-level code should contain certain amount of dataflow information to realize eager data transfer at the lower-level. The dataflow information is actually needed, rather than for code optimization, for more basic compilation process — code conversion (generating lower-level code from higher-level code). The information includes (when fixing a value v): (1) whether or not v is used eventually, (2) which object will use v, (3) to which location on memory v will be finally stored, etc. Especially (2) and (3) are indispensable to realizing the pipelined sends described in Chapter 6.

The *plan-do style* code satisfies the requirement described above also provides additional benefit for the compilation process itself. This subsection briefly describes the plan-do style code; the important effect of using plan-do style (i.e., realizing eager data transfer, such as pipelined sends) will be demonstrated when the plan-do style code is interpreted by the lower-level part of the compiler, which is demonstrated in Section 8.4.

The basic idea behind plan-do style is that (1) the *plan-part* (i.e., plan declaration part) provides dataflow information, and (2) the *do-part* (i.e., plan execution part) provides control-flow information. The plan-do style code embodies the semantics of the source program in terms of sequential execution with a single thread of control, and also provides dataflow information without loss of machine independence of high-level code. The benefits of the plan-do style are: (1) it takes a stream form instead of an arbitrary graph structure, and (2) interleaving of plan declaration and plan execution is possible. Thus, it allows combining of several compilation phases into one pass (or alternatively, pipelined parallelization of the compilation phases).

There are compilation targets for which the plan-do style code is specially suitable, which are the *expressions* representing natural dataflow information; using the plan-do style code makes it possible to extract and represent the information without costly flow analysis.

Let us start with a brief example. First, consider the following ABCL/ST source program:

```
[obj1 <= [:msg1 x (+ i 1)]]
```

where we assume that `x` is of `real` type, `i` is of `int` type, `obj1` is of `(obj mset1)` type (it means the object can accept a message of type `mset1`.), and `mset1` is a disjoint-union type whose elements contain `(:msg1-*-* [real int])` type (`:msg1-*-*` is an injection tag, and the tuple type `[real int]` is the type of its carried value.). This program fragment in ABCL/ST denotes a request message passing to object `obj1`, whose message is composed by an injection tag `:msg1-*-*` and a real number `x` and an integer `i`+1.

This program would be converted into the following type-checked high-level parse tree:

```
(request-send top
      ((varref (obj mset1) (obj1))
       (inject mset1
               (:msg1-*-* (make-tuple [real int]
                            ((varref real (x))
                             (+ int ((varref int (i)) (const int (1))))))))))
```

Here, the tree is represented in the following grammar[3]:

$$\begin{array}{rcl} \langle \textit{syntax tree} \rangle & ::= & (\langle \textit{construct} \rangle\ \langle \textit{type} \rangle\ (\left\{\!\left\{ \langle \textit{subtree} \rangle \right\}\!\right\})) \\ \langle \textit{subtree} \rangle & ::= & \langle \textit{syntax tree} \rangle \,|\, \langle \textit{variable id} \rangle \,|\, \langle \textit{const} \rangle \,|\, \langle \textit{keyword} \rangle \cdots \end{array}$$

Here, $\left\{\!\left\{\ \right\}\!\right\}$ denotes more than zero repetition.

This tree would be converted into the following high-level code:

```
(newplan (p1 (d1 (obj mset1)) (d2 mset1)) (request-send))
(throw obj1 d1)
(newplan (p2 (d3 [real int])) (inject :msg1-*-* d2))
(newplan (p3 (d4 real) (d5 int)) (make-tuple d3))
```

[3]Here, we use a grammar based on the so-called *syntax tree* rather than *parse tree* to represent the tree for short.

$$
\begin{array}{rcl}
\langle \textit{plan decl} \rangle & ::= & \texttt{(newplan ((}\langle \textit{plan id}\rangle\ \langle \textit{dest decl}\rangle\texttt{) (initialize }\langle\textit{variable id}\rangle\texttt{))}\ | \\
 & & \texttt{(newplan ((}\langle \textit{plan id}\rangle\ \langle \textit{dest decl}\rangle\texttt{) (update }\langle\textit{l-exp}\rangle\texttt{))}\ | \\
 & & \texttt{(newplan ((}\langle \textit{plan id}\rangle\ \langle \textit{dest decl}\rangle\texttt{) (dist }\left\{\!\left\{\langle\textit{dest id}\rangle\right\}\!\right\}\texttt{))}\ | \\
 & & \texttt{(newplan ((}\langle \textit{plan id}\rangle\ \left\{\!\left\{\langle \textit{dest decl}\rangle\right\}\!\right\}\texttt{) (make-tuple }\langle\textit{dest id}\rangle\texttt{))}\ | \\
 & & \texttt{(newplan ((}\langle \textit{plan id}\rangle\ \langle \textit{dest decl}\rangle\texttt{) (inject }\langle\textit{keyword}\rangle\ \langle\textit{dest id}\rangle\texttt{))}\ | \\
 & & \texttt{(newplan ((}\langle \textit{plan id}\rangle\ \langle \textit{dest decl}\rangle\texttt{) (out }\langle\textit{dest id}\rangle\texttt{))}\ | \\
 & & \texttt{(newplan ((}\langle \textit{plan id}\rangle\ \langle \textit{dest decl}\rangle\texttt{) (get-part }\langle\textit{field number}\rangle\ \langle\textit{dest id}\rangle\texttt{))} \\
\langle \textit{dest decl} \rangle & ::= & \texttt{((}\langle\textit{dest id}\rangle\ \langle\textit{type}\rangle\texttt{)} \\
\langle \textit{l-exp} \rangle & ::= & \texttt{(var }\langle\textit{variable id}\rangle\texttt{)}\ |\ \texttt{(part }\langle\textit{field number}\rangle\ \langle\textit{l-exp}\rangle\texttt{)}\ |\ \texttt{(tuple }\left\{\!\left\{\langle\textit{l-exp}\rangle\right\}\!\right\}\texttt{)}
\end{array}
$$

Figure 8.3: The Syntax of Plan Declaration (Data Transfer/Construct/Selection Part Only)

```
(throw x d4)
(newplan (p4 (d6 int) (d7 int)) (arith3 + d5))
(throw i d6)
(throw 1 d7)
(do p4 p3 p2 p1)
```

`(newplan (p1 (d1 (obj mset1)) (d2 mset1)) (request-send))` declares *plan* `p1` with *destinations* `d1` and `d2`. By 'destination', we mean an argument of the plan (which corresponds to a dataflow arc in the terminology of dataflow model). The plan is to send a *request message* to a target object. The target object's name is given by throwing the name to `d1`, whose type is `(obj mset1)`. By 'throwing', we mean that a value is passed as an argument for a plan. The value of request message is given by throwing the value to `d2`, whose type is `mset1`.

`(throw obj1 d1)` throws the value of `obj1` to the destination `d1`. `(do p1)` carries out the plan `p1`.

Similarly, `(newplan (p2 (d3 [real int])) (inject :msg1-*-* d2))` declares plan `p2` to (1) construct a tagged value by injecting the value thrown to `d3` with an injection tag `:msg1-*-*` then (2) throw the tagged value to `d2`. `(newplan (p3 (d4 real) (d5 int)) (make-tuple d3))` declares plan `p3` to (1) make a tuple of values thrown to `d4` and `d5` then (2) throw the tuple to `d3`. `(newplan (p4 (d6 int) (d7 int)) (arith3 + d5))` declares plan `p4` to (1) add values thrown to `d6` and `d7` then (2) throw their sum to `d5`.

Figure 8.3 shows a part of the syntax of plan declaration.

The translation of the parse tree into the plan-do style code is straightforward: when the (recursive) translation function reaches a certain tree node, before the translation of the subtree, it declares the plan for the current node, and after the translation of the subtree, it 'executes' the plan.

8.3.2 Standard Interpretation of Plan-Do Style Code

This section briefly summarizes a standard and trivial interpretation of plan-do style code for later discussion; this interpretation adopts a plenty of temporary variables, and keeps itself at the high-level. However, such an interpretation would not exploit the possible pipelined send capabilities of the underlying architecture. An alternative interpretation developed for eager data transfer is presented in Section 8.4.

The trivial interpretation is as follows:

1. A plan declaration, such as `(newplan (p1 (d1 int) (d2 int)) (arith3 + d0))`, is interpreted just as declaration and no actions are performed. A destination is interpreted as a temporary variable; declaration of destinations is interpreted as declaration of temporary variables.

2. `(throw` ⟨*value*⟩ `d1)` is interpreted as an assignment of the value to the temporary variable that is denoted by `d1`.

3. `(do p1)` is interpreted as an actual execution of the plan `p1`. When the plan was declared by `(newplan (p1 (d1 int) (d2 int)) (arith3 + d0))`, the interpretation of `(do p1)` would assign the sum of values of temporary variables `d1` and `d2` to the temporary variable `d0`.

8.3.3 Imperative Updates

The left hand side of an assignment remains in the plan-do code as it is:

```
[(part1 (part2 x)) := (+ i 1)]
```

The syntax tree would be:

```
(setf int
      ((part int (1 (part [int real] (2 (varref [int [int real]] (x))))))
       (+ int ((varref int (i)) (const int (1))))))
```

The high-level code would be:

```
  (newplan (p1 (d1 int)) (update (part 1 (part 2 (var x)))))
  (newplan (p2 (d2 int)) (dist d1 d0))
  (newplan (p3 (d3 int) (d4 int)) (arith3 + d2))
  (throw i d3)
  (throw 1 d4)
  (do p3 p2 p1)
```

As we can see, for the left hand side of imperative update operation, the plan-do style is not used. This is because updates of writable variables can not usually advance the timing of the update for consistency.

8.3.4 Pattern Matching and Variable Binding

This section describes the capability of high-level code for variable binding and pattern matching compilation. These are common tasks to convert the high-level tree into sequential code. The characteristic of the code described below provides an opportunity of register allocation as is described in Section 8.5.1.

The example is as follows:

```
(wait-for
 (=> [:msg1 r i]
     [s := (+ s r)]))
```

The syntax tree would be:

```
(wait-for real
    (((inject mset1
              (:msg1-*-*
                  (make-tuple [real int]
                      ((varref real (r)) (varref int (i))))))
      (progn
        real
        ((setf real
               ((varref real (s))
                (+ real ((varref real (s)) (varref real (r))))))))))
```

The high-level code is shown in Figure 8.4. `(newtmp t1 (qpos mset1))` declares a temporary variable `t1` of type `(qpos mset1)`; `t1` is a pointer to the current message in the object's message queue. `(newplan (p1 (d1 (qpos mset1))) (initialize t1))` declares plan `p1` to initialize `t1` with the value thrown to `d1`.

The compilation result of a pattern matching consists of the following binding statements:

$$\texttt{(case-bind } \langle \textit{variable id} \rangle \ \texttt{(}\Big\{\Big\{\texttt{((}\langle \textit{keyword} \rangle \ \texttt{(}\langle \textit{variable id} \rangle \ \langle \textit{type} \rangle\texttt{)} \ \langle \textit{label id} \rangle\texttt{)}\Big\}\Big\}\texttt{)} \ \langle \textit{label id} \rangle\texttt{)}$$

$$\texttt{(tuple-bind } \langle \textit{variable id} \rangle \ \texttt{(}\Big\{\Big\{\texttt{((}\langle \textit{variable id} \rangle \ \langle \textit{type} \rangle\texttt{)}\Big\}\Big\}\texttt{))}$$

$$\texttt{(bind } \langle \textit{variable id} \rangle \ \texttt{((}\langle \textit{variable id} \rangle \ \langle \textit{type} \rangle\texttt{))}$$

The `case-bind` statement (1) matches the keywords (listed in the second argument) for the injection tag of its first argument (of course, of which type must be a disjoint-union type) and (2) branches to the corresponding label with the corresponding newly declared read-only variable which is bound to the carried value of the first argument. If there are no keywords for matching, it branches to the label given as the third argument. The `tuple-bind` statement binds the newly declared read-only variables (listed in the second argument) to the corresponding tuple elements of its first argument (of course, of

```
  (newtmp t1 (qpos mset1))
  (newplan (p1 (d1 (qpos mset1))) (initialize t1))
  (newplan (p2) (get-qhead d1))
  ;; plan to locate queue pointer 't1' at the head of message queue
  (do p2 p1)
L0:
  (newplan (p3 (d2 (qpos mset1))) (update (var t1)))
  (newplan (p4 (d3 (qpos mset1))) (next-qpos d2))
  ;; plan to locate queue pointer 't1' at the next location of message queue
  ;; if the message for the next location is not available, wait for it
  (throw t1 d3)
  (do p4 p3)
  (bind-msg t1 (t3 mset1)) ;; 't3' is bound to the message at 't1'
  (case-bind t3 ((:msg1-*-* (t4 [real int]) L1)) L2) ;; pattern matching
L1:
  (tuple-bind t4 ((t5 real) (t6 int)))
  (bind t5 (r real))
  (bind t6 (i int))
  (newplan (p7 (d6 (qpos mset1))) (delete))
  ;; plan to delete the message at 't1' from the message queue
  (throw t1 d6)
  (do p7)
  (newplan (p8 (d7 real)) (update (var s)))
  (newplan (p9 (d8 real)) (dist d7 d0))
  ;; 'd0' is the destination for the value of the whole 'wait-for' expression
  (newplan (p10 (d9 real) (d10 real)) (arith3 + d8))
  (throw s d9)
  (throw r d10)
  (do p10 p9 p8)
  (go L3)
L2:
  (go L0)
L3:
```

Figure 8.4: High Level Code for `wait-for`

which type must be a tuple type), while `bind` statement binds the newly declared read-only variable of its second argument to the value of its first argument. Here, the newly declared variables (pattern variables) are read-only variables; this characteristic provides an opportunity of advanced register/storage allocation as is described in Section 8.5.1.

8.4 Realizing Pipelined Sends

This section presents our compilation method for realizing eager data transfer, such as pipelined sends described in Chapter 6, by interpreting high-level plan-do style code.

The principle of plan-do style is that *the declaration of an action (*plan*) prior to its actual execution (*do*) provides the advanced (future dataflow) information.* This information helps the middle-level converter choose the middle-level code to implement the high-level semantics. The example information provided by the plan-do style code include (when fixing a value v):

1. Whether or not v is used eventually,
2. Which object will finally use v,
3. Into which location on memory v will be stored finally.

Especially 2. and 3. are indispensable to realizing the pipelined send.

8.4.1 Data Representation for Pipelined Sends

We have proposed the middle-level data representation for subtype relation in Section 8.2.3. This section gives the unspecified potion of the middle-level representation to implement message passings.

The middle-level type, $\langle \mathit{MLtype} \rangle$, is given by the syntax in Figure 8.5, which appends the definition of $\langle \mathit{MLimptype} \rangle$ to the syntax of middle level type which already appear in Figure 8.2.

Recall that a middle-level representation, $\langle \mathit{MLrepr} \rangle$, represents a high-level type. The representation uses two kinds of parenthesis [] and (). Using two kinds of parenthesis is just to make it easier for middle-level code to implement high-level field selection. A representation parenthesized by [] denotes a high-level tuple type. On the other hand, a representation parenthesized by () denotes a high-level type other than tuple type, whose value can not be decomposed in terms of high-level field selection.

The write pointers (**MLowptr**(), **MLkwptr**(), **MLrqstwptr**(), **MLrplywptr**()) are essential to our runtime architecture; they are used to point to objects, boxes for carried values of disjoint-union type data, message boxes for request sends, message boxes for reply sends, respectively, during their initialization. Compiler generated (middle-level) code (not runtime kernel code) is responsible for memory allocation/initialization. The process of initialization is pipelined and interruptible (can be interleaved with other

$$
\begin{array}{rcl}
\langle \mathit{MLrepr} \rangle & ::= & (\langle \mathit{MLtype} \rangle, \ldots, \langle \mathit{MLtype} \rangle) \mid [\langle \mathit{MLrepr} \rangle, \ldots, \langle \mathit{MLrepr} \rangle] \\
\langle \mathit{MLtype} \rangle & ::= & \langle \mathit{MLb} \rangle \mid \mathbf{MLidx} \mid \mathbf{MLuptr} \mid \mathbf{MLkptr}(\langle \mathit{MLrepr} \rangle) \mid \langle \mathit{MLimptype} \rangle \\
\langle \mathit{MLb} \rangle & ::= & \mathbf{MLint} \mid \mathbf{MLreal} \mid \mathbf{MLbool} \\
\\
\langle \mathit{MLimptype} \rangle & ::= & \mathbf{MLoptr} \mid \mathbf{MLowptr(}\langle \mathit{MLrepr} \rangle\mathbf{)} \mid \\
& & \mathbf{MLqpos} \mid \\
& & \mathbf{MLkwptr}(\langle \mathit{MLrepr} \rangle) \mid \\
& & \mathbf{MLrqstptr}(\langle \mathit{MLrepr} \rangle) \mid \mathbf{MLrqstwptr}(\langle \mathit{MLrepr} \rangle) \mid \\
& & \mathbf{MLrplyptr}(\langle \mathit{MLrepr} \rangle) \mid \mathbf{MLrplywptr}(\langle \mathit{MLrepr} \rangle)
\end{array}
$$

Figure 8.5: Syntax of Middle-Level Type for ABCL/EM-4

initialization processes). When the initialization completes, the write pointer is converted into the read pointer.

The distinguish of write pointers and read pointers is necessary for converting them into low-level representation, where a middle-level memory block may be represented by a tree of (fixed-size) memory boxes, and the read pointer may be represented by one word pointer to the root box and the write pointer may be represented by several words to all the boxes, because we want all the read pointers to have the same size, while the write pointers can be larger size to improve initialization speed.

The following function definitions are used to specialize the definition of *Repr* and *Arepr* in Section 8.2.3.

$$
\begin{array}{rcl}
\mathit{IMPoptr}(\tau) & = & (\mathbf{MLoptr}) \\
\mathit{IMPrptr}(\tau) & = & (\mathbf{MLoptr}, \mathbf{MLrplywptr}(\mathit{Repr}(\tau))) \\
\mathit{IMPqpos}(\tau) & = & (\mathbf{MLqpos})
\end{array}
$$

Examples are:

$$
\begin{array}{rcl}
\mathit{Arepr}(\mathbf{obj}\ s) & = & (\mathbf{MLoptr}, \mathbf{MLidx}) \\
\mathit{Arepr}(\mathbf{@\ int}) & = & (\mathbf{MLoptr}, \mathbf{MLrplywptr(MLint)})
\end{array}
$$

The first one is a typical representation of object names, and the second one is a typical representation of reply destinations.

8.4.2 Alternative Interpretation of Plan-Do Style Code for Pipelined Sends

This subsection describes the compilation process for realizing pipelined sends, by interpreting the high-level plan-do style code. First, we will take a simple message send example to illustrate the flow of compilation:

```
[obj2 <= [(+ i 1) (+ j 2)]]
```

This program fragment in ABCL denotes a request message passing to object `obj2`, whose message is a tuple of an integer `i+1` and an integer `j+2`. This source program is converted into the following plan-do style code:

```
(newplan (p1 (d1 (obj [int int])) (d2 [int int])) (request-send))
(throw obj2 d1)
(newplan (p2 (d3 int) (d4 int)) (make-tuple d2))
(newplan (p3 (d5 int) (d6 int)) (arith3 + d3))
(throw i d5)
(throw 1 d6)
(do p3)
(newplan (p4 (d7 int) (d8 int)) (arith3 + d4))
(throw j d7)
(throw 2 d8)
(do p4 p2 p1)
```

Figure 8.6 shows the translation function Tr from the high-level code into the middle-level code. Here, when $hcode$ is high-level plan-do style code, $Tr\,[\![hcode]\!]\,(penv, denv)$ returns the middle-level code in the context of the plan environment $penv$ and the destination environment $denv$. $penv$ is a function from a set of plan identifiers (ranged over by p) to plan definitions (p corresponds to a dataflow node in terms of the dataflow model). $denv$ is a function from a set of destination identifiers (ranged over by d) to pairs of p and integers (ranged over by i). (d stands for i-th dataflow arc to p in terms of the dataflow model.) Note that, in Figure 8.6, type information is omitted for simplicity.

$Trnsfr\,[\![v_l, fl, lf, d]\!]\,(penv, denv)$ returns a fragment of middle-level code which transfers a value v_l to a field of destination d; the field is selected by the high-level field number list fl and the middle-level field number lf. In contrast to the standard interpretation of plan-do style code describe in Section 8.3.2, the results of $Trnsfr$ are significantly influenced by the dataflow context of ($p$env,$d$env) in order to select the eager data transfer instructions under certain conditions.

In Figure 8.6, for a function f, "$f\{x \mapsto y\}$" denotes the function f' such that $Dom(f') = Dom(f)$, $f'(x) = y$ and $f'(x') = f(x')$ for all $x' \in Dom(f) - \{x\}$. By "$h :: r$", "::" denotes consing of the head h and list r, and "$[a, b, c]$" denotes the list consisting of a, b, and c. By "l_1 @ l_2", "@" denotes the concatenation of two lists l_1 and l_2.

The translation function basically translates the high-level code in the following way:

- For a plan declaration, it memorizes the plan and destinations in the environment $penv$ and $denv$. New variable identifiers for middle-level code will be introduced if necessary.

- For a plan execution, it generates a code fragment for finishing the plan, then if the result value has not been eagerly sent, it generates data transfer instructions by calling the $Trnsfr$ function.

$$Tr\,[\![\texttt{(newplan } p_1\ d_1\ d_2\ \texttt{(req-send))} :: r]\!]\,(penv, denv) =$$
$$Tr\,[\![r]\!]\,(penv\{p_1 \mapsto \textbf{req-send}(t_1, t_2, t_3)\}, denv\{d_1 \mapsto (p_1, 1), d_2 \mapsto (p_1, 2)\})$$
$$(t_1,\ t_2,\ t_3 \text{ new})$$

$$Tr\,[\![\texttt{(newplan } p_1\ d_1\ d_2\ \texttt{(make-tuple } d_3\texttt{))} :: r]\!]\,(penv, denv) =$$
$$Tr\,[\![r]\!]\,(penv\{p_1 \mapsto \textbf{make-tuple}(d_3)\}, denv\{d_1 \mapsto (p_1, 1), d_2 \mapsto (p_1, 2)\})$$

$$Tr\,[\![\texttt{(newplan } p_1\ d_1\ d_2\ \texttt{(arith3-+ } d_3\texttt{))} :: r]\!]\,(penv, denv) =$$
$$Tr\,[\![r]\!]\,(penv\{p_1 \mapsto \textbf{arith3-+}(d_3, (t_1, t_2, t_3))\}, denv\{d_1 \mapsto (p_1, 1), d_2 \mapsto (p_1, 2)\})$$
$$(t_1,\ t_2,\ t_3 \text{ new})$$

$$Tr\,[\![\texttt{(do } p_1\texttt{)} :: r]\!]\,(penv, denv) =$$
$$\texttt{(get-read-pointer } t_2\ t_3\texttt{)} :: \texttt{(req-send } t_3\ t_1\texttt{)} :: Tr\,[\![r]\!]\,(penv, denv)$$
$$\text{if } penv(p_1) = \textbf{req-send}(t_1, t_2, t_3)$$

$$Tr\,[\![\texttt{(do } p_1\texttt{)} :: r]\!]\,(penv, denv) = Tr\,[\![r]\!]\,(penv, denv) \qquad \text{if } penv(p_1) = \textbf{make-tuple}(d_1)$$

$$Tr\,[\![\texttt{(do } p_1\texttt{)} :: r]\!]\,(penv, denv) =$$
$$\texttt{(arith3-+ } t_1\ t_2\ t_3\texttt{)} :: (\mathit{Trnsfr}\,[\![t_3, \textbf{nil}, 0, d_1]\!]\,(penv, denv) @ Tr\,[\![r]\!]\,(penv, denv))$$
$$\text{if } penv(p_1) = \textbf{arith3-+}(d_1, (t_1, t_2, t_3))$$

$$Tr\,[\![\texttt{(throw } v_h\ \texttt{d1)} :: r]\!]\,(penv, denv) =$$
$$(\mathit{Trnsfr}\,[\![v_{l1}, \mathit{fl}_1, \mathit{lf}_1, d_1]\!]\,(penv, denv) @ \cdots @ \mathit{Trnsfr}\,[\![v_{ln}, \mathit{fl}_n, \mathit{lf}_n, d_1]\!]\,(penv, denv)) @$$
$$Tr\,[\![r]\!]\,(penv, denv)$$
$$\text{where } \mathit{Decomp}(v_h) = \{(v_{l1}, \mathit{fl}_1, \mathit{lf}_1), \ldots, (v_{ln}, \mathit{fl}_n, \mathit{lf}_n)\}$$

$$\mathit{Trnsfr}\,[\![v_l, \textbf{nil}, 0, d_1]\!]\,(penv, denv) = [\texttt{(assign } v_l\ t_1\texttt{)}, \texttt{(get-rqst-mbox-on } t_1\ t_2\texttt{)}]$$
$$\text{if } denv(d_1) = (p_1, 1) \text{ and } penv(p_1) = \textbf{req-send}(t_1, t_2, t_3)$$

$$\mathit{Trnsfr}\,[\![v_l, \mathit{fl}, \mathit{lf}, d_1]\!]\,(penv, denv) = [\texttt{(setarg-remote } v_l\ t_2\ \mathit{fl}\ \mathit{lf}\texttt{)}]$$
$$\text{if } denv(d_1) = (p_1, 2) \text{ and } penv(p_1) = \textbf{req-send}(t_1, t_2, t_3)$$

$$\mathit{Trnsfr}\,[\![v_l, \mathit{fl}, \mathit{lf}, d_1]\!]\,(penv, denv) = \mathit{Trnsfr}\,[\![v_l, (i-1) :: \mathit{fl}, \mathit{lf}, d_2]\!]\,(penv, denv)$$
$$\text{if } denv(d_1) = (p_1, i) \text{ and } penv(p_1) = \textbf{make-tuple}(d_2)$$

$$\mathit{Trnsfr}\,[\![v_l, \textbf{nil}, 0, d_1]\!]\,(penv, denv) = [\texttt{(assign } v_l\ t_i\texttt{)}]$$
$$\text{if } denv(d_1) = (p_1, i) \text{ and } penv(p_1) = \textbf{arith3-+}(d_3, (t_1, t_2, t_3)) \ \ (\text{for } 1 \le i \le 2)$$

Figure 8.6: Translation Function from Plan-Do Style Code into Middle-Level Code for Pipelined Sends.

- For throwing a value to a destination, it decomposes the (possibly) nested higher-level tuple value into a set of triples of (1) middle-level values, (2) the high-level field number lists and (3) the middle-level field number by calling *Decomp*, and generates data transfer instructions by calling *Trnsfr*.

Tr and *Trnsfr* embody the translation into eager data transfer; i.e., they realize the pipelined sends. For this purpose, *Tr* translates the high-level code in the following way:

- For simple arithmetic operations, it simply prepares three temporary variables for the operation, and the result of the operation is transferred by *Trnsfr*.
- For tuple construction, it eagerly transfers the tuple elements without buffering.
- For request message passings, it prepares pointers for a request message box to realize pipelined send, and it generates only instructions for pointer manipulation.

Trnsfr realizes data transfer in the context of ($penv$,$denv$) in the following way:

- If the destination is just for an arithmetic operation, the value is simply assigned to the temporary variables for the operation.
- If the destination is for a tuple construction, it eagerly transfers the value to the more specific field of destination.
- If the destination is for a message, it generates a "remote write" instruction for pipelined send.
- If the destination is for a target of the request message send, the value is simply assigned to the temporary variables for the target, but, immediately after that, it allocates a message box for pipelined send.

Figure 8.7 shows the result of conversion from the high-level code into middle-level code for an example code fragment of simple message send. The high-level code is also shown as comments. Eager data transfers appear at `(do p3)` and `(do p4)`, where the elements of the tuple are eagerly sent to the pre-allocated message box on a remote node. The conversion proceeds as follows:

1. When `p1` is declared, the translator produces a temporary variable `t1` for `d1` in the same manner as the standard interpretation, but it produces message box pointers `t2` and `t3` rather than temporary variables for `d2`.
2. When `obj2` is thrown to `d1` in the high-level code, the translator assigns the corresponding middle-level variable `obj2-0` to `t1`. Once `t1` is initialized, The translator allocates a message box on the node of `t1` for the plan `p1` prior to its execution of `(do p1)`.
3. When `p2` is declared, the translator does not produce temporary variables for `d3` and `d4`, because it performs eager throws to the specific parts of `d2` rather than indirect throws buffered by temporary variables.

```
;; Comments at this level are the high-level code.
;; (newplan (p1 (d1 (obj [int int])) (d2 [int int])) (request-send))
  (newtmp t1 MLoptr)
  (newtmp t2 (MLrqstwptr [(MLint) (MLint)]))
  (newtmp t3 (MLrqstptr [(MLint) (MLint)]))
;; (throw obj2 d1)
  (assign obj2-0 t1)
  (get-rqst-mbox-on t1 t2)      ; allocate a message box t2 on the node of t1
;; (newplan (p2 (d3 int) (d4 int)) (make-tuple d2))
;; (newplan (p3 (d5 int) (d6 int)) (arith3 + d3))
  (newtmp t4 MLint)
  (newtmp t5 MLint)
  (newtmp t6 MLint)
;; (throw i d5)
  (assign i-0 t4)
;; (throw 1 d6)
  (assign 1 t5)
;; (do p3)
  (arith3 + t4 t5 t6)
  (setarg-remote t6 t2 (0) 0) ; remote write of the result of i+1
;; (newplan (p4 (d7 int) (d8 int)) (arith3 + d4))
  (newtmp t7 MLint)
  (newtmp t8 MLint)
  (newtmp t9 MLint)
;; (throw j d7)
  (assign j-0 t7)
;; (throw 2 d8)
  (assign 2 t8)
;; (do p4)
  (arith3 + t7 t8 t9)
  (setarg-remote t9 t2 (1) 0) ; remote write of the result of j+2
;; (do p2)
;; (do p1)
  (get-read-pointer t2 t3)      ; complete the initialization of the message box
  (request-send t3 t1)          ; send the message box address
```

Figure 8.7: Generated Middle Level Code for Example Simple Message Send

4. Declaration of `p3` is processed in the same manner as the standard interpretation.

5. When `(do p3)` throws the result of $i+1$ to `d3`, the translator scans the history of the definition of `d3`, and discovers (1) `d3` is used as the first element of tuple construction in `p2` , (2) the tuple will be thrown to `d2`, (3) `d2` is used as the message of a request message send in `p1`, and (4) the message box for the request message send has been allocated and `t2` holds its address; it then generates the `setarg-remote` instruction which remotely writes `t6` to the remote message box pointed to by `t2`. (the field is selected by a high-level field number list `(0)` and a middle-level field number `0`.)

6. Declaration of `p4` and `(do p4)` is processed similarly to `p3`.

7. For `(do p2)`, the translator produces no instructions, because it has decided to use eager throws to `d2` and the eager throws have been performed in the process of `(do p3)` and `(do p4)`.

8. For `(do p1)`, the translator produces only instructions for pointers, because eager data transfer has already sent the contents of the message.

Compilation with plan-do style code is different from standard instruction scheduling. Let us take the following example:

```
[obj3 <= [:msg2 [:cons 10 [:cons 20 [:nil]]]]]
```

This program fragment in ABCL/ST denotes a request message send to `obj3`, whose message consists of an injection tag `:msg2-*` and a list of integers `10` and `20`. Compilation with plan-do style code makes it possible for the middle-level converter to produce *remote cons* code rather than *local cons* plus *remote copy* for the list construction.

Next, we examine more typical example mentioned in Section 8.3.1:

```
[obj1 <= [:msg1 x (+ i 1)]]
```

and its plan-do style code:

```
(newplan (p1 (d1 (obj mset1)) (d2 mset1)) (request-send))
(throw obj1 d1)
(newplan (p2 (d3 [real int])) (inject :msg1-*-* d2))
(newplan (p3 (d4 real) (d5 int)) (make-tuple d3))
(throw x d4)
(newplan (p4 (d6 int) (d7 int)) (arith3 + d5))
(throw i d6)
(throw 1 d7)
(do p4 p3 p2 p1)
```

Figure 8.8 shows the result of conversion from the high-level into middle-level. In the last instruction, two data `t3` (of middle-level type **MLuptr**) and `t4` (of middle-level type **MLidx**) are simultaneously sent. This skips an extra (remote) memory allocation for **MLrqstwptr**. This scheme is used because, in the typical message send, the size of message (other than the carried value of disjoint union) is small, and the greater part of pipelined send is realized through the *remote cons* of the disjoint-union value.

```
;; Comments at this level are the high-level code.
;; (newplan (p1 (d1 (obj mset1)) (d2 mset1)) (request-send))
  (newtmp t1 MLoptr)
  (newtmp t2 MLidx)
  (newtmp t3 MLuptr)
  (newtmp t4 MLidx)
;; (throw obj1 d1)
  (assign obj1-0 t1)
  (assign obj1-1 t2)
;; (newplan (p2 (d3 [real int])) (inject :msg1-*-* d2))
  (newtmp t5 (MLkwptr [(MLreal) (MLint)]))
  (newtmp t6 (MLkptr [(MLreal) (MLint)]))
  (get-keyword-mbox-on t1 t5)   ; allocate a middle-level memory cell t5
                                ;  on the node of t1 for 'remote cons'
;; (newplan (p3 (d4 real) (d5 int)) (make-tuple d3))
;; (throw x d4)
  (setarg-remote x-0 t5 (0) 0) ; remote write of 'x'
;; (newplan (p4 (d6 int) (d7 int)) (arith3 + d5))
  (newtmp t7 MLint)
  (newtmp t8 MLint)
  (newtmp t9 MLint)
;; (throw i d6)
  (assign i-0 t7)
;; (throw 1 d7)
  (assign 1 t8)
;; (do p4)
  (arith3 + t7 t8 t9)
  (setarg-remote t9 t5 (1) 0)   ; remote write of the result of i+1
;; (do p3)
;; (do p2)
  (get-read-pointer t5 t6)      ; complete the initialization of
                                ; the middle-level memory cell
  (assign t6 t3)
  (assign 1 t4) ; if index-diff([:msg1 real int], mset1) = 1
;; (do p1)
  (arith3 + t4 t2 t4)           ; send the index and
                                ; the middle-level memory cell address
  (request-send2 t3 t4 t1)
```

Figure 8.8: Generated Middle Level Code for Typical Message Send

8.5 Optimization of Register Allocation

8.5.1 Minimization of Register Context

Let us consider the following example:

```
(wait-for
 (=> [:msg1 r i o]
     [o <= (+ r i)]))
```

pattern variables `r`, `i`, and `o` are bound to the message contents. In ABCL/ST, pattern variables are read-only. Possible implementations of these variables are:

1. Three additional temporary variables are used to hold these values as a part of local context. Performing the pattern matching copies these values to the temporary variables. These copies themselves will be performed as just loads to the registers from the keyword memory box which contains these tree values, but the values will be stored into the memory as a part of local context upon other operations, such as context switches, which destroy the register contents

2. Only one temporary variable is used as a part of local context to hold the address of keyword memory box which contains these three values. Performing the pattern matching copies the single pointer to the temporary variable. Each access to the pattern variables is basically implemented as an extract operation from the memory box; but the subsequent access to the same variable may be cached in registers. This mechanism is easily realized by assigning, as home positions, the location of the values in the memory box to these pattern variables; the memory at the home position is non-dirty, which means no need to store value on spilling. This is non-traditional usage of home position, because storage for home positions are usually allocated as local context by storage allocator of the compiler

We adopted scheme 2. This scheme decreases the amount of context. In the above example, a context switch occurs when receiving (remote) allocated message box for message send `[o <= (+ r i)]`, the number of store operations before the context switch is less than that of scheme 1.

8.5.2 Least Immediately Used Algorithm

For register allocation, we do not employ the standard graph coloring algorithm; we rather use a more optimal algorithm, where the spilled value is one that is used farthest in the future. Let us call it the *Least Immediately Used* (LIU) algorithm by analogy to the *Least Recently Used* (LRU) algorithm.

The LIU algorithm corresponds to a famous (imaginary) page replacement policy in the research area of operating system, which is called *optimal*, because it has been proven that the policy is the optimal algorithm. It would be in a sense imaginary as a page replacement policy because it uses future information, but it is not imaginary in register allocation because the "future" for register allocator of the compiler can be determined statically (except for branches).

The LIU algorithm is not a new algorithm, but the advantage of LIU over graph coloring is strengthened by the characteristics of EM-4. In EM-4, registers are more volatile than usual; registers' values are not guaranteed to be kept over context switches and the context switch is always necessary for receiving a new packet from the network. Thus we should switch the followings over context switches to increase register utilization:

1. variables to which registers should be allocated and
2. registers which should be allocated to the same variable.

This is automatically done with the LIU algorithm by simply giving a specification that the instructions receiving packets use all the registers for working registers, whereas the graph coloring algorithm must be modified to support code partition mechanism whose development is difficult.

8.5.3 Register Allocation to Encapsulated Variables

In object-oriented languages, state (instance) variables are usually *encapsulated*. In ABCL, the encapsulation is achieved by the following constraints:

1. Only the code that belongs to the same class (or superclass/subclass, if exists) can manipulate the state (instance) variables.
2. Only a single thread of control manipulates the state (instance) variables simultaneously (not shared). (This is trivial in sequential languages, but not in concurrent languages.)

These constraints make it possible to allocate register to state (instance) variables. The register allocator should reflect this feature to optimize the speed of accesses to the state (instance) variables. So, our compiler supports this allocation.

By contrast, most C (C++) compilers do not support this allocation, where only temporary (auto) variables are the target of register allocation other than function parameters.[4] This is because the instance variables of the C++'s object are held in the memory pointed by a C's pointer, *self pointer*, and C is originally designed for description of operation systems where the memory access can not be eliminated to communicate with the hardware or user processes.

[4]Recent GNU CC++ for SPARC (ver. 2.4.5) solves this problem in a limited form by peephole optimization, which eliminates several memory operations on the same address, but not ones which are interleaved with other memory operations on different addresses.

Chapter 9

Applications

9.1 Algorithms Based on Concurrent Object-Oriented Model

In Section 2.1.1, we described our concurrent object-oriented computation model; however, in order to use the concurrent OO model for algorithm evaluations, we should specify some more details. A concurrent OO model for algorithm evaluations can be defined as follows:

1. Each concurrent object has equal computation power.

2. Each message transmission between concurrent objects is performed within an equal communication latency.

3. Each creation of concurrent object is performed within an equal creation latency.

Some of the ideas behind the above concurrent OO model are shared by a recently proposed model, LogP[5], which is introduced with the motivation that (1) PRAM is unrealistic model of real highly parallel computers and (2) a new parallel model which reflects the real hardware architectures is required.

The LogP model characterizes a parallel machine by the number of processors(P), the communication bandwidth(g), the communication delay(L), and the communication overhead(o). In LogP, a good algorithm embodies a strategy for adapting to different machines, in terms of these parameters. LogP encourages techniques, such as careful data placement and work assignment and careful scheduling and overlapping of computation and communication. Thus somewhat low-level algorithm specification is required to demonstrate its performance.

On the other hand, the concurrent OO model provides more abstract characteristics, where the requirements to be a good algorithm are:

- As many concurrent objects as possible are employed,

- Concurrent objects are equally employed, and message bottlenecks are avoided.

Note that the concurrent OO model and the LogP model do not conflict. They are at the different levels of abstraction; the concurrent OO model provides a convenient technique which simplifies analysis, as long as a constraint is met[5]. The constraint is finite network capacity given by L/g ($1/g$ corresponds to the per-processor communication bandwidth). The concurrent OO model addresses this constraint at its implementation level, where LogP model could be used.

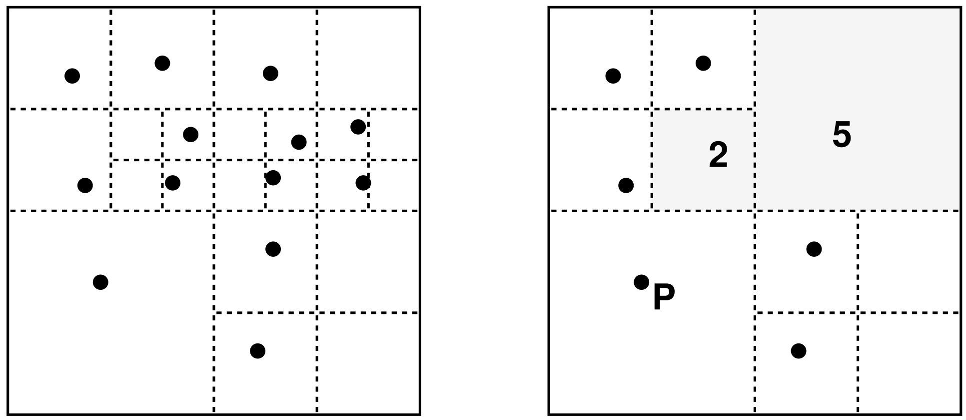
In the next section, we propose a fast parallel algorithm for N-body problem based on the concurrent OO model. This algorithm is a good algorithm in terms of requirement described above. This algorithm is also used to evaluate our concurrent OO language system in Chapter 10.

9.2 N-Body Problem

The Newtonian N-body problem is to find the behavior of N particles interacting with each other through a long range force such as gravity or Coulombic force. Efficient methods for solving this problem provide powerful research tools in physics and astronomy. Since a naive (exact) simulation needs to compute the interactions of all the pairs of particles, its time complexity, which is $O(N^2)$, becomes prohibitive as N increases. Thus, methods have been developed to reduce the computational cost by approximation. One such method was developed by J. Barnes and P. Hut[4]. The time complexity of their method is $O(N \log N)$ and its rigorous upper bound of errors can be estimated. Another approach has been taken by L. Greengard [7], who used the multipole expansion method. This thesis presents a new parallel algorithm for the Newtonian N body problem, which is inspired by the method of Barnes and Hut. Our algorithm [29, 30] is based on the use of a large number of concurrent objects[32, 1, 37]. The time complexity of this algorithm is $O(\log N)$ using $O(N)$ concurrent objects. This algorithm exploits the characteristic nature of object-oriented concurrency[32, 1, 37] using dynamic concurrent object creation and the dynamic reconfiguration of the message passing topologies.

9.2.1 Serial Algorithm (Barnes and Hut)

The essence of Barnes and Hut's algorithm[4] is the use of a tree structure, which corresponds to the spatial distribution of particles. Nodes in the tree structure correspond to cells (or subspaces) which are obtained by dividing the entire space containing all the particles in the following manner. The entire space is recursively divided into cells with child cells having exactly half the length, breadth and width of their parent. Cells containing just one particle (or none at all) are not further subdivided. Figure 9.1(a) shows such a division of space in the two dimensional case. Dots denote particles. A cell containing no particles has no corresponding node in the tree structure. A cell containing just one particle has a single corresponding node which is a terminal (leaf) node in the tree structure. The root node corresponds to the entire space and non-terminal nodes correspond to cells containing at least two particles.



(a) Entire Space is Divided Recursively. (b) Virtual Particles(gray zones)

Figure 9.1: Division of Space (2-D case)

Having such a tree structure, we assume that each node in the tree structure keeps the total mass and the center-of-gravity of the particles contained in the corresponding cell. Thus, a non-terminal node can be viewed as a terminal node for a cell which contains a single *virtual* particle. The virtual particle's mass and position are the total mass and center-of-gravity, respectively, of the particles actually contained in the cell. In Figure 9.1(b), a number in a cell stands for a virtual particle and indicates the number of actual particles represented.

If a particle P whose distance from the center-of-gravity is sufficiently larger than the diameter of the cell (i.e., the longest distance between any pair of particles in the cell), the total force exerted on P by the particles in the cell does not have to be computed as the sum of the individual forces exerted by the particles in the cell. The total force can be estimated within a required error bound as the force exerted by the virtual particle. Whether or not the approximation technique is employed is determined by a threshold value θ derived from the specified error bound. Letting the diameter of a cell be s, and the distance between P and the position of the virtual particle be d, if the inequality

$$s/d < \theta$$

holds, approximation is permissible. Figure 9.1(b) shows how this approximation is done. The virtual particles gray-zoned in Figure 9.1(b) are used when the force exerted on P is estimated. The analysis by Barnes and Hut[4] shows that the time required to compute the total force exerted on a single particle by all other particles in the space is $O(\log N)$. Thus, to compute the force exerted on N particles requires $O(N \log N)$ time.

9.2.2 Overview of Our Parallel Algorithm

An overview of our parallel algorithm is given as follows:

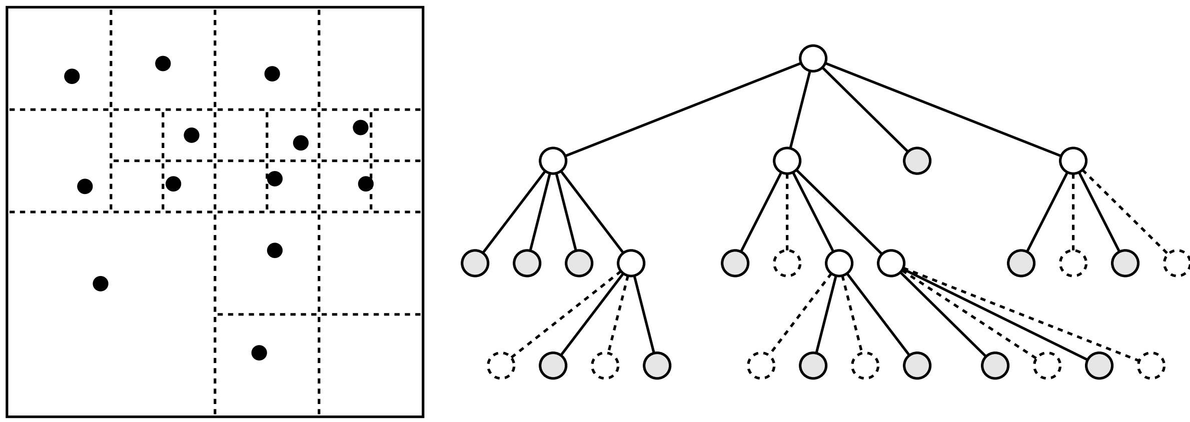


Figure 9.2: Tree Structure of Cell Objects

1. For each node in the tree structure of Barnes and Hut's algorithm, there is a concurrent object, called a *cell object*, which keeps the physical information such as the number of the particles in the corresponding cell, their total mass, and their center-of-gravity.

2. Cell objects form a tree structure isomorphic to that of Barnes and Hut's algorithm. (See Figure 9.2.) Thus each cell object also keeps the structural information (of the corresponding tree structure) such as the names of its parent cell objects, child cell objects (and sibling cell objects if necessary). This tree structure is constructed by performing parallel *summation* operations on tree structures, which will be explained in Section 9.2.4.

3. The *terminal* cell object residing at a terminal node of the tree structure represents a single particle in the system. The non-terminal cell object residing at a non-terminal node of the tree structure represents a virtual particle or a cell containing more than one particle.

4. The total force exerted on a particle is calculated by its corresponding terminal cell object by summing up forces. Each force is derived by using the mass and position of a non-negligible (virtual) particle; this information is contained in a message sent to the terminal cell object from the cell object representing the (virtual) particle. The range of the message distribution and how messages are distributed will be described in Section 9.2.3.

5. Such messages are sent concurrently to different terminal cell objects. Thus a very high degree of concurrency is obtained.

6. At the beginning of each time interval, a new tree structure composed of cell objects which reflect the updated values of particle's velocity and position is constructed.

Keeping this general picture of our parallel algorithm in mind, let us go into details.

9.2.3 Parallel Calculation of Forces

We assume that each cell object of the tree keeps the following data about the cell (subspace) that it represents:

1. the size of the cell,
2. the position of the center of the cell,
3. the center-of-gravity of the particles contained in the cell,
4. the number of the particles contained in the cell,
5. the total mass of the particles contained in the cell,
6. the structural information such as who are its parent, children, etc.

As described in the previous section, to calculate the total force exerted on a particle P within an error bound, the terminal cell object TC representing P must know the mass, and position of all non-negligible (virtual) particles. Messages containing this information, which we call *interaction-force messages* or *IFM*s, are transmitted from the cell objects representing the non-negligible (virtual) particles. Upon receiving a relevant *IFM*, the terminal cell object TC calculates the force exerted by the corresponding (virtual) particle. This calculation continues until no more relevant *IFM*s are received. Exhaustion of the relevant *IFM*s can be easily detected by counting the particles represented in the *IFM*s.

What needs to be more precise about is the range of the distribution of *IFM*s and how they are distributed. First, let us consider the range of the distribution. The set of terminal cell objects that need to be sent an *IFM* from a cell object C is the set of terminal cell objects whose particles are "in range" of the (virtual) particle represented by C. Note that two *IFM*s, one sent by a cell object and the other sent by the parent of the cell object, may reach the same terminal cell object. In such cases, the terminal cell object may include the force of the same particle more than once. This must be avoided. The distribution area of an *IFM* from C is the set of points (1) for which approximation relative to C is possible and (2) for which approximation relative to PC, the parent of C, is not possible. This area is illustrated as the gray zone in Figure 9.3.

How is the distribution of an *IFM* done? An *IFM* transmitted from C first travels upward to its PC. Then, the PC adds information about its approximation boundary to the *IFM* and launches it into the network of cell objects. Seemingly, the routing of *IFM* can be performed efficiently by cell objects that have links to their parent and children. However, it should be noted that all the non-terminal cell objects concurrently transmit *IFM*s. Thus a single cell object may simultaneously receive many *IFM*s originated from different cell objects causing a message processing bottleneck, especially near the root cell

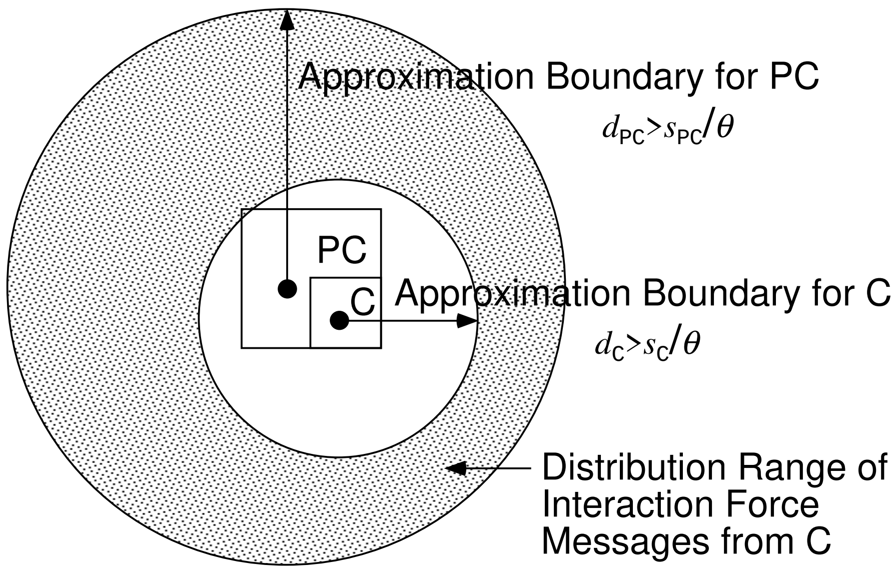


Figure 9.3: The Range of the Distribution of Interaction-Force Messages (2-D case)

object. To alleviate this problem, we put *horizontal* links between adjacent cell objects at the same level of the tree structure (i.e., which are also of the same size) if adjacent ones exist. See Figure 9.4(a).

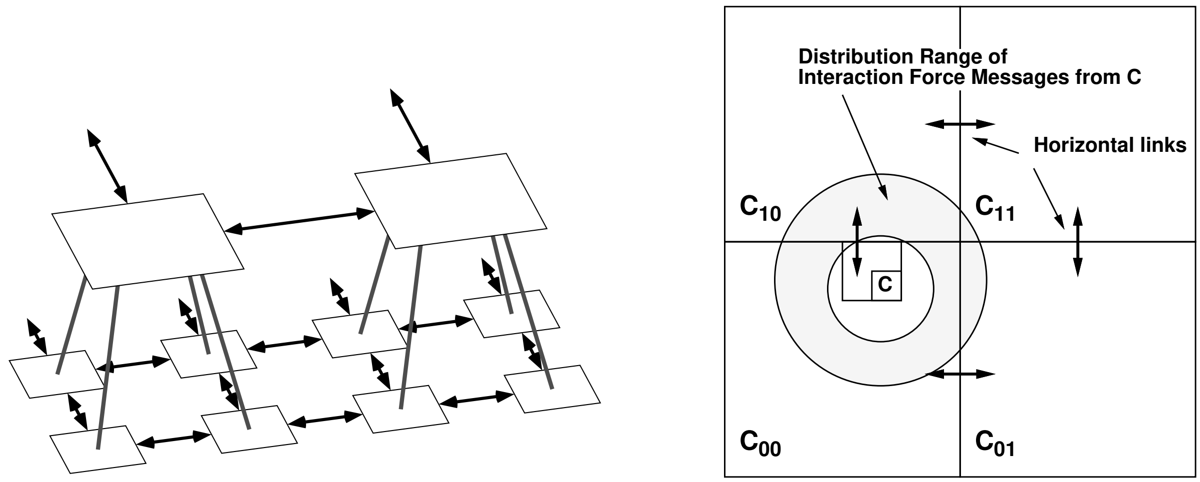
Such horizontal links allow interaction-force messages to bypass parent cell objects, reducing the congestion on vertical links. We can distribute an *IFM* in the following manner:

1. The *IFM* is repeatedly forwarded upward[1] to the first cell object whose cell size is larger than the size of the distribution range of the *IFM* (C_{00} in Figure 9.4(b)). Note that the cell does not always contain the distribution range because their centers usually differ (e.g., C_{00} in Figure 9.4(b) does not contain the distribution range).

2. Along the horizontal links, the *IFM* is distributed among the multiple cell objects so that the union of their corresponding cells covers the distribution range (C_{00}, C_{01}, C_{10} and C_{11} in Figure 9.4(b)). Note that the parent cell of C_{00} does not always cover the distribution range; thus such multiple cell objects which are connected by horizontal links are employed to avoid further upward-forwarding.

3. Each distributed *IFM* is recursively forwarded downward to the cell objects whose corresponding cells intersect the range of the distribution.

[1]By 'upward', we mean the direction to the parent cell object



(a) Horizontal Links (b) Use of the horizontal links

Figure 9.4: Horizontal Links between Adjacent Cell Objects of the Same Level (2-D case)

Our analysis shows that this eliminates the bottlenecks[2]. Having no bottlenecks allows us to estimate the upper bound of the time complexity of the parallel force calculation of this section as the total length of path along which an *IFM* travels. This is at most $O(\log N)$. Furthermore, it can be easily shown the construction of such links requires only $O(\log N)$ time complexity, assuming that the tree structure already exists.

9.2.4 Parallel Tree Construction

In order to achieve $O(\log N)$ time complexity for our entire algorithm, we must be able to construct the tree structure in $O(\log N)$ time.

Summation of Tree Structures

We devised a commutative operation called *summation* for tree structures, which is used in our parallel tree construction algorithm. Let A and B be tree structures of cell objects whose roots represent the same space. We denote the tree structure obtained by the summation operation of A and B by:

$$A + B$$

The root of $A+B$ represents the same space represented by the root of A and/or B and $A+B$ contains the union of the sets of particles contained in A and B. Of course, $A+B$ has the same structural property explained in Section 9.2.1.

Given two tree structures A and B which represent the same space, how do we obtain $A+B$? First, the data to be kept in the root cell object of $A+B$ is computed by using

[2]By a bottleneck, we mean that more than constant time is required for a message to be processed after arriving at a concurrent object.

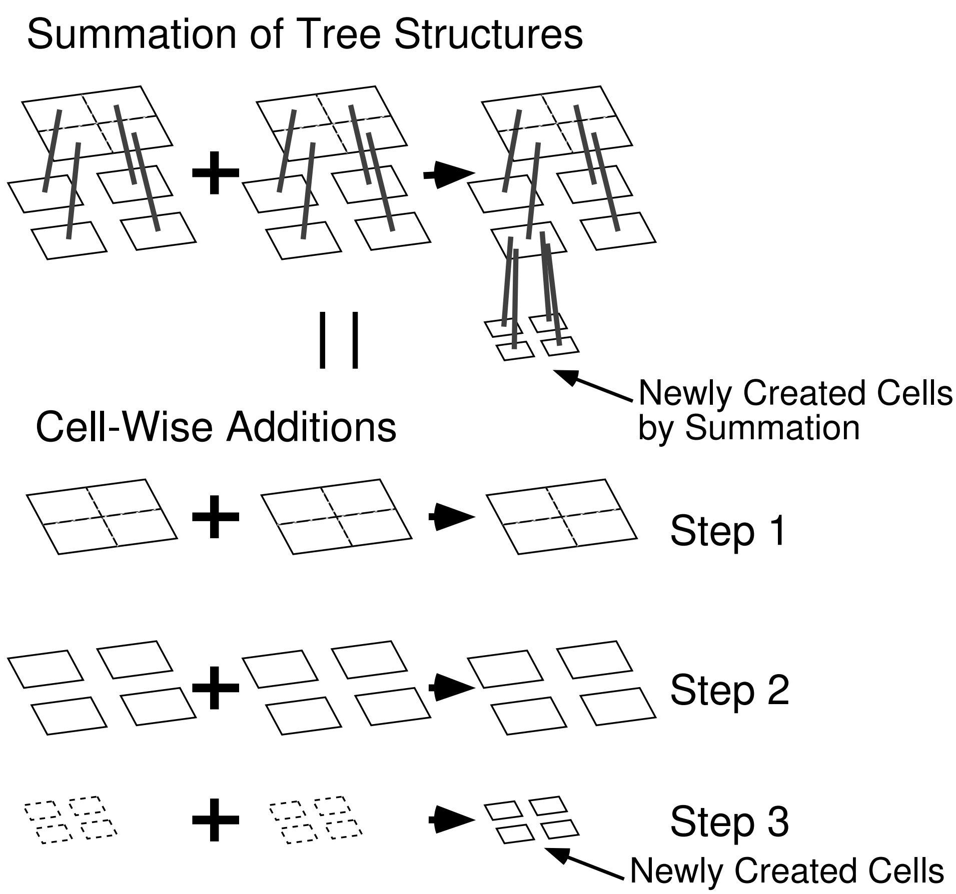


Figure 9.5: Summation of Tree Structures

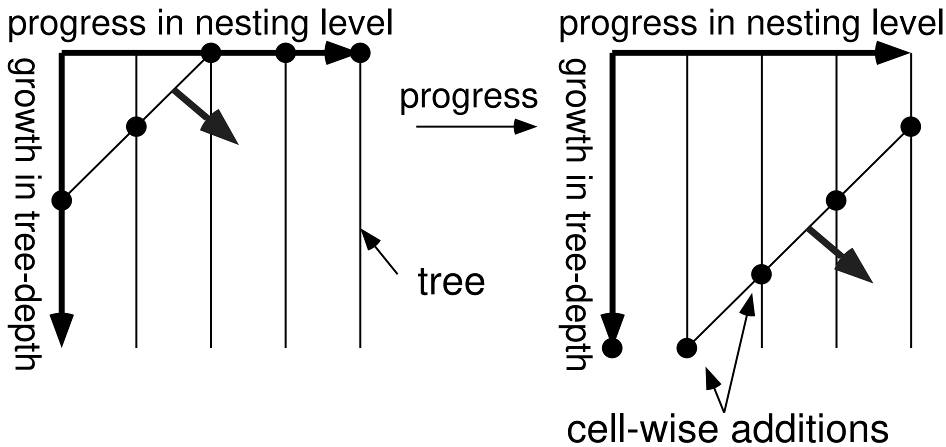


Figure 9.6: Pipelined Parallel Summation of Tree Structures

the total masses, the centers-of-gravity and the total numbers of particles kept in the root cell objects of A and B. Then, the data kept in each child of $A+B$ is computed by using the data kept in the corresponding children of A and B. (See Figure 9.5.) This process of *cell-wise additions* is repeated until all the terminal nodes of $A+B$ are reached. Note that the total number of cell objects in $A+B$ may be greater than that of A and B. Furthermore, the depth of $A+B$ may be greater than that of A and B. (Note Step 3 in Figure 9.5.)

Since the summation operation on tree structures is performed from the root towards their leaves sequentially, its time complexity is the depth of the resulting tree structure. In the case where the depth of tree structure is $O(\log N)$, the time complexity is $O(\log N)$,

Tree Construction by Pipelined Parallel Summations

Now we consider how a tree structure is created from the initial set of data about the particles. Suppose we have particles a, b, c, d, e, f, g and h. Correspondingly, we create cell objects A, B, C, D, E, F, G, and H, all of which we regard as (singleton root-only) tree structures. Instead of adding each tree structure one by one, the following nested summation is performed to obtain a tree structure T containing all the particles a through h.

$$T = (((A+B)+(C+D))+((E+F)+(G+H)))$$

Note that the summation operations at a same level of nesting are performed in parallel. Thus, $(A+B)$, $(C+D)$, $(E+F)$, and $(G+H)$ are performed in parallel. Since the number of nesting levels is bounded by $O(\log N)$ and the each individual summation operations requires $O(\log N)$ time at most (as noted at the end of the previous subsubsection), the total construction of T requires $O((\log N)^2)$ time at worst. Unfortunately, our goal is $O(\log N)$ not $O((\log N)^2)$.

The reason the construction of T requires $O((\log N)^2)$ is the fact that a summation operation of two trees at a given nesting level is performed after completing the construction of the two trees. To achieve our goal of attaining $O(\log N)$ time complexity, we can make the summation operation of two trees at a given nesting level and the construction of the two trees overlap in time. In other words, the summations are pipelined over nesting levels. This pipelining is illustrated in Figure 9.6 where the vertical coordinate represents the growth (construction) of individual trees and the horizontal coordinate represents the progress in the nested summation operation. The circles on a diagonal line represent the cell-wise additions in the summation operations being performed in parallel. For example, the cell-wise addition of the third level (from the root) cell objects of A and B, the cell-wise addition of the second level cell objects of $A + B$ and $C + D$, and the cell-wise addition of the root cell objects of $(A+B)+(C+D)$ and $(E+F)+(G+H)$ are performed in parallel. Now it is easily seen that it takes $O(\log N)$ time for the diagonal line to finish sweeping the whole area.

9.2.5 Summary of N-body Problem

We have presented a $O(\log N)$ parallel algorithm for the Newtonian N-body problem using $O(N)$ concurrent objects. The main techniques devised are: (1) elimination of message bottlenecks by originating *interaction-force messages* from non-terminal cell objects instead of terminal cell objects, (2) elimination of message bottlenecks by routing through *horizontal* links connecting adjacent cell objects of the same level, (3) employing *summation* operations for tree structures, (4) tree construction by *pipelined* parallel summations.

Chapter 10

Performance Measurements with Real Applications

This chapter presents the evaluations of our concurrent OO language system, ABCL/EM-4, and our concurrent OO algorithms for N-body problems.[31]

10.1 Evaluation Conditions

First, we briefly present the structure of the program of the N-body algorithm, which is used for performance measurements. Appendix A gives the whole program.

The program is based on the new algorithm described in the Section 9.2[29, 30]. The time complexity of the algorithm is $O(\log N)$ using $O(N)$ concurrent objects. The essence of the algorithm is the use of a tree structure, which corresponds to the spatial distribution of particles. The algorithm repeats two phases, namely tree construction phase and force calculation phase.

The program is written for the 1-D N-body problem, and it mainly consists of the following three classes:

1. `Tree-Cons-Cell` class

2. `Force-Calc-Cell` class

3. `main` class

A single `main` class object is created at first, and it directs other objects. `Force-Calc--Cell` class objects compose the main tree structure, on which forces are calculated. `Tree-Cons-Cell` class objects are employed as temporary cells for tree construction. Roughly speaking, `Force-Calc-Cell` class objects have `Force-Calc-Obj` type and `Tree--Cons-Cell` class objects have `Tree-Cons-Obj` type. Since the program defines `Force--Calc-Obj` as a subtype of `Tree-Cons-Obj`, `Force-Calc-Cell` class objects are also employed as `Tree-Cons-Obj` type at the final stage of a tree construction.

The evaluation conditions are as follows:

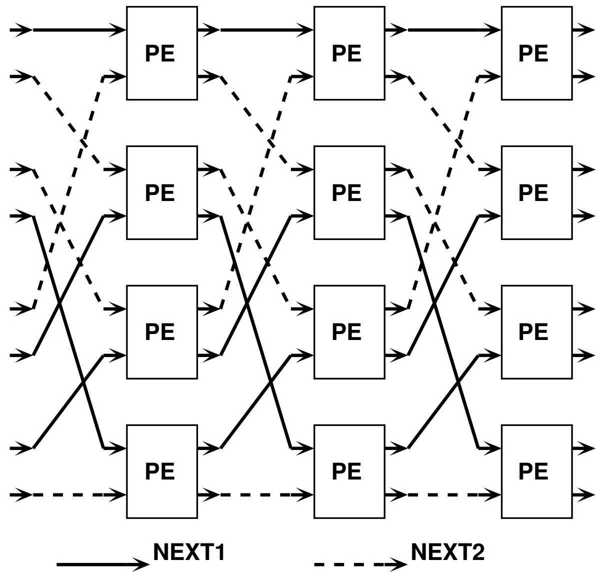


Figure 10.1: Omega Network of EM-4 ($3 \times 2^{3-1}$ case for short, $5 \times 2^{5-1}$ in reality)

1. It uses whole 80 PEs.
2. Approximation rate (θ) is set to 0.4.

Since this measurement is performed with the 1-D N-body problem, we adopt the following two particle distributions:

Even: the positions of particles are distributed evenly.

Series: the positions of particles are distributed as a series whose difference is from $2L/3N$ to $4L/3N$, where L is the size of the entire space. (We adopt this distribution because the interval of the even distribution may be resonant with the interval of the space division.)

Since it is 1-D case, the number of child objects is 2. The network of EM-4 is illustrated in Figure 10.1; we adopt the following schemes to determine nodes at which objects are created:

N1: One is created at NEXT1, the other is created at NEXT2.

N11: One is created at NEXT1 from NEXT1, the other is created at NEXT2.

In the above schemes, we create objects at as near nodes as possible, in order to reduce the consumption of the network resources.

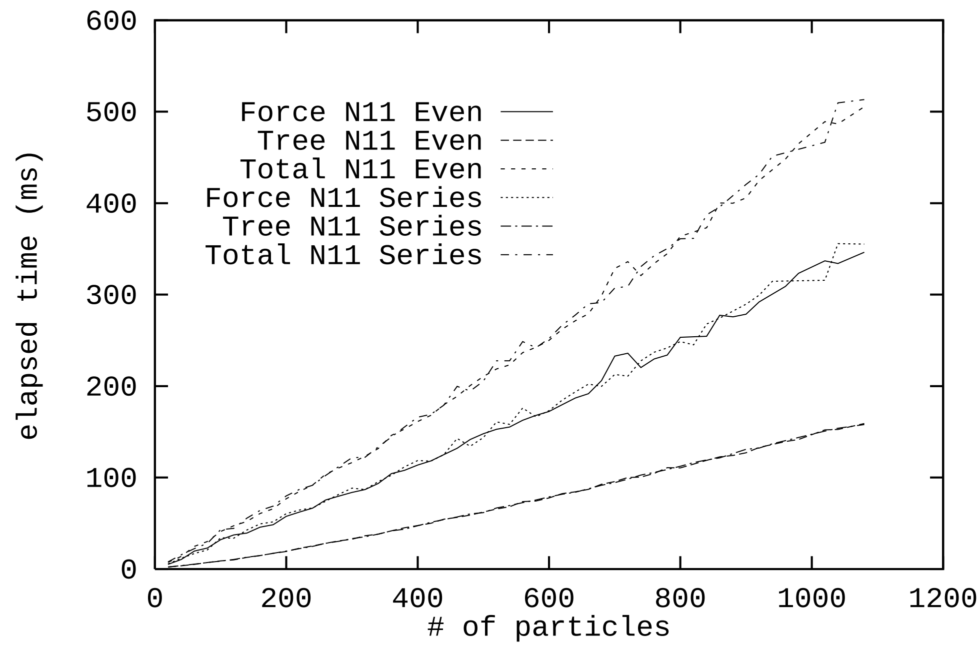


Figure 10.2: Elapsed Time of N-body Problem

10.2 Evaluation Results

Figure 10.2 shows the elapsed time of tree construction (`Tree`), force calculation (`Force`), and their totals. Here, we measured the elapsed time of 1-iteration (1 time step) of N-body problem.

Figure 10.3 shows the elapsed time of `Force` using N1 and N11 methods. N1 method shows behavior, which has peaks. This is explained by the following reason: In the network of Figure 10.1, the one step deeper node on the tree structure just corresponds the one step righter node of the network; consequently the leaf objects of the tree are lined at one column, causing the load imbalance.[1]

Figure 10.4 shows elapsed time per $N \log N$ against N. Since the total amount of computation is $O(N \log N)$, this graph shows the performance of the system; as we can see, the system achieves the peak performance from $N \sim 200$. Since (1) the number of processors (P) is 80 and (2) the number of leaf (terminal) objects used for `Force` is N,[2] when $N = 200$, the rate N/P is 2.5. This small number indicates that: Our concurrent OO algorithm and language system do not require so large N in order to exploit all P processors. This property is important, especially in parallel computers with large P.

[1] In fact, N11 is introduced after finding the reason.

[2] Totally, the number of objects is approximately $2N$ because the tree is a binary tree, but the dominant part of computation (i.e., calculation of Newton's equation) is performed by the leaf objects.

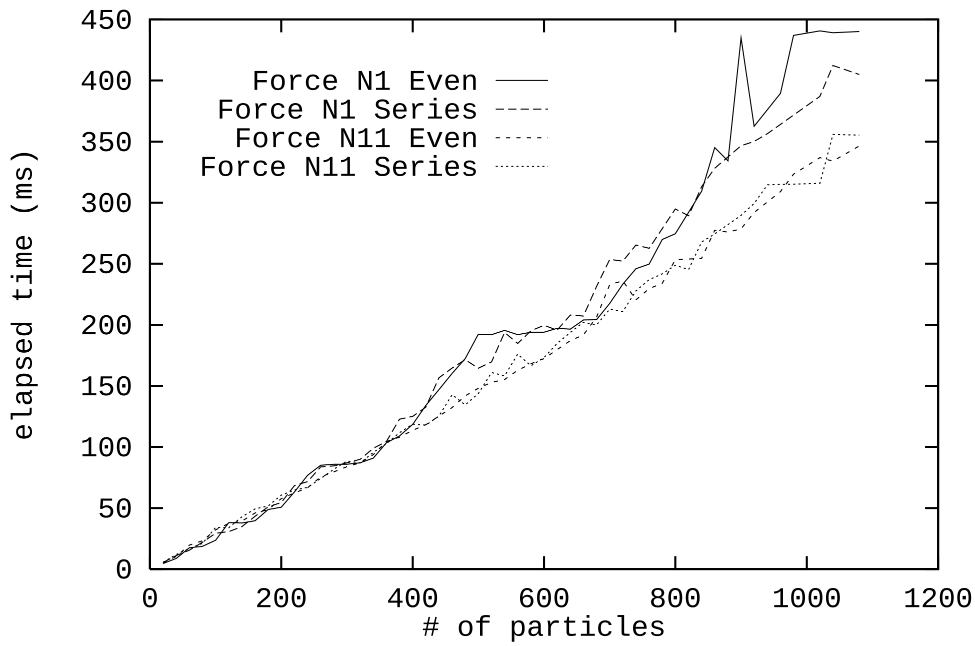


Figure 10.3: Comparison between PE Mapping Schemes

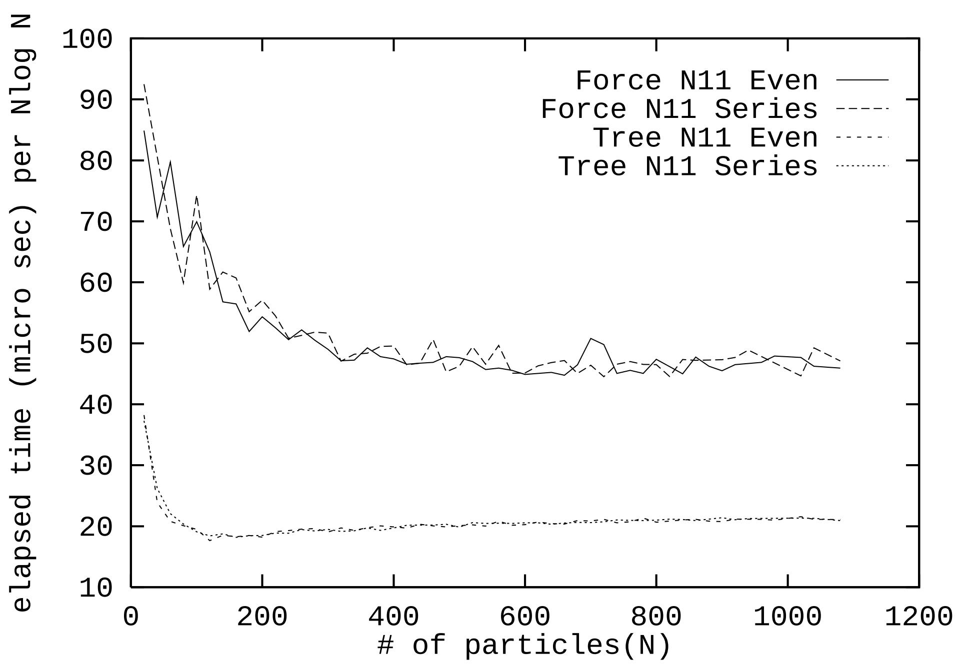


Figure 10.4: Elapsed Time per $N \log N$

N(# of particles)	520		1040	
phase	tree cons.	force calc.	tree cons.	force calc.
# of Objects creation	27,140	0	60,516	0
# of Messages	126,826	64,730	282,754	140,774
Elapsed time (μs)	65,787	152,913	152,675	334,105

Table 10.1: Evaluation with N-body Problem

Compiler / Machine	Time
ABCL/EM-4 (EMC-R, 12.5MHz, 80PEs, no FPU)	0.49 s
EM-C / EM-4 (EMC-R, 12.5MHz, -O, sequential execution on 1PE, no FPU)	17.9 s
GCC / SS1 (SPARC, 20MHz, -O2, FPU)	1.64 s
GCC / SS1+ (SPARC, 25MHz, -O2, FPU)	1.43 s
GCC / SS2 (SPARC, 40MHz, -O2, FPU)	0.76 s
GCC / SS10 (Supar SPARC, 36MHz, -O2, FPU)	0.47 s
GCC / LUNA88K (m88k, 25MHz, -O2, FPU)	1.07 s

Table 10.2: Performance Comparison on Different Software/Hardware using N-body Problem ($N = 1040$)

Table 10.1 lists some numbers from the evaluations; when N=1040, it creates more than 60,000 objects and exchanges more than 420,000 messages, and the elapsed time is less than 0.5s.

Table 10.2 lists the elapsed time measured on different language system/architecture combinations. The C source program used in the measurement exploits an optimization specific to sequential execution (e.g., global variables are used to represent messages). ABCL/EM-4 shows the secondly good performance, in spite of the slowest clock speed. Moreover, EM-4 uses integer instructions to perform real-number arithmetic operations; it does not have FPUs. For this reason, EM-C is the worst in the table. The next-generation EM-4, EM-X, includes FPUs, which should increase the total performance considerably.

10.3 Feedback to Language System

The result of the previous section provides us important information for building an automatic load balancing mechanism in our language systems. The enhanced language system would automatically extract the runtime object topologies from the source language as precisely as possible, and choose a pseudo-optimal mapping to the real nodes of the hardware architecture.

Of course, the precise runtime information is not available at the compile time. However, the first approximation of the object topology is provided by creator-createe relation of objects in the program. (The second approximation would be provided by sent-received relation of messages in addition to the creator-createe relation of objects.) The compiler would extract the following creator-createe relation to estimate the object topology:

1. Simultaneously multiple creations

2. Repeated creations

3. Recursive creations

The extraction of the above information can be performed as a part of flow analysis which is already required for optimization.

Chapter 11

Conclusion

This thesis explores a wide range of research topics for obtaining high performance by concurrent object-oriented language systems on highly-parallel computers, including language design, runtime architectures, optimizing compilers, and applications.

We have proposed several techniques, including reactive representations of objects, efficient management of message buffers with freelists, interruptible pipelined message sends, plan-do style intermediate code, an efficient implementation scheme for table look-up, advanced register/storage allocation, and a concurrent object-oriented algorithm for N-body problem.

I believe that these techniques are equally important. Moreover, they do not work alone; their effects are multiplied by each other.

Performance measurements on EM-4 show that the cost of each remote object-creation and remote message-passing can be, in order, comparable to that of a sequential procedure call and the N-body algorithm based on the concurrent object-oriented models can be executed efficiently for a wide range of N. Our results indicate that the object-oriented concurrent computation models and languages are extremely suitable for realizing practical parallel systems.

We have developed a prototype compiler and a runtime system which run ABCL/ST programs, such as the N-body problem program proposed in this thesis, on the real machine EM-4. I would like to continue the implementation work for pursuing high-performance, on EM-4 and other highly parallel machines.

I hope that the techniques and results we have described in this thesis will pave the way towards the development of even more efficient implementation technologies for concurrent object-oriented languages as well as languages of other computational paradigms, I also hope this work leads concurrent object-oriented languages into the more advanced implementation on new-generation massively-parallel computers.

Bibliography

[1] G. Agha. *Actors : A Model of Concurrent Computation in Distributed Systems.* The MIT Press, 1987.

[2] P. America and J. Rutten. A layered semantics for a parallel object-oriented languages. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Proc. of REX/FOOL, Noordwijkerhout, The Netherlands*, volume 489 of *Lecture Notes in Computer Science*, pages 91–123. Springer-Verlag, May/June 1990.

[3] T. Baba, T. Yoshinaga, T. Iijima, Y. Iwamoto, M. Hamada, and M. Suzuki. A parallel object-oriented total architecture: A-NET. In *Proc. of Supercomputing'90, New York*, pages 276–285, November 1990.

[4] J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, 1986.

[5] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming PPOPP*, pages 1–12, San Diego, California, May 1993.

[6] W. J. Dally, J. S. Fiske, J. S. Keen, R. A. Lethin, M. D. Noakes, P. R. Nuth, R. E. Davison, and G. A. Fyler. The message-driven processor: A multicomputer processing node with efficient mechanisms. *IEEE MICRO*, 12(2):23–39, Apr. 1992.

[7] L. Greengard. *The Rapid Evaluation of Potential Fields in Particle Systems.* ACM Press, 1987.

[8] T. G. Griffin. A formulae-as-types notion of control. In *Proceedings of 17th ACM Symposium on Principles of Programming Languages, Sanfrancisco*, pages 47–57, January 1990.

[9] K. Hiraki, T. Shimada, and S. Sekiguchi. Empirical study of latency hiding on a fine-grain parallel processor. In *Proc. of 7th ACM International Conference on Supercomputing, Tokyo, Japan*, pages 220–229, July 1993.

[10] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In P. America, editor, *Proc. of ECOOP'91, Geneva, Switzerland*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer-Verlag, July 1991.

[11] W. Horwat. Concurrent smalltalk on the message-driven processor. Technical Report MIT-AI-TR 1321, MIT Artificial Intelligence Lab., 1991.

[12] W. Horwat, A. A. Chien, and W. J. Dally. Experience with CST: Programming and implementation. In *Proc. of the SIGPLAN'89 Conference on Programming Language Design and Implementation, Portland, Oregon*, pages 101–109, June 1989.

[13] H. Ishihata, T. Horie, S. Inano, T. Shimizu, and S. Kato. An architecture of highly parallel computer AP1000. In *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 13–16, May 1991.

[14] D. R. Jefferson. Virtual time. *ACM Trans. Prog. Lang. Syst.*, 7(3):404–425, 1985.

[15] T. Kamada, S. Matsuoka, and A. Yonezawa. An algorithm of distributed garbage collection on a multicomputer and its performance evaluation. In *Proc. of JSSST 10th Annual Convention, Waseda-University, Japan*, pages 21–24, June 1993. (in Japanese).

[16] Y. Kodama, S. Sakai, and Y. Yamaguchi. A prototype of a highly parallel dataflow machine EM-4 and its preliminary evaluation. In *Proc. of InfoJapan '90*, pages 291–298, 1990.

[17] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, A. Yonezawa, and P. Wegner, editors, *Research Directions in Concurrent Object-Oriented Programming*. The MIT Press, 1993.

[18] J. Meseguer. A logical theory of concurrent objects. In *Proc. of OOPSLA/ECOOP'90, Ottawa, Canada*, pages 101–115, October 1990.

[19] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An architecture of a dataflow single chip processor. In *Proc. of the 16th Annual International Symposium on Computer Architecture*, pages 46–53, June 1989.

[20] M. Sato, Y. Kodama, S. Sakai, and Y. Yamaguchi. Em-c: A parallel programming language for the em-4 multiprocessor. In *Proc. of Joint Symposium on Parallel Processing 1993(JSPP'93), Waseda-University, Japan*, pages 183–190, May 1993. (in Japanese).

[21] M. Sato, Y. Kodama, S. Sakai, Y. Yamaguchi, and Y. Koumura. Thread-based programming for the EM-4 hybrid dataflow machine. In *Proc. of the 19th Annual International Symposium on Computer Architecture*, May 1992.

[22] E. Shibayama. *An Object-Based Approach to Modeling Concurrent Systems*. PhD thesis, Department of Information Science, The University of Tokyo, 1991.

[23] E. Shibayama and A. Yonezawa. Distributed discrete event simulation in ABCL/1. In A. Yonezawa, editor, *ABCL: An Object-Oriented Concurrent System*, chapter 9. The MIT Press, 1990.

[24] K. Taura, S. Matsuoka, and A. Yonezawa. An efficient implementation scheme of concurrent object-oriented languages on stock multicomputers. In *Proceedings of the ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming PPOPP*, pages 218–228, San Diego, California, May 1993.

[25] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *The 19th Annual International Symposium on Computer Architecture*, volume 20, pages 256–266, Gold Coast, Australia, May 1992.

[26] M. Yasugi, S. Matsuoka, and A. Yonezawa. ABCL/onEM-4: A new software/hardware architecture for object-oriented concurrent computing on an extended dataflow supercomputer. In *Proc. of 6th ACM International Conference on Supercomputing, Washington D.C.*, pages 93–103, July 1992.

[27] M. Yasugi, S. Matsuoka, and A. Yonezawa. ABCL/onEM-4: An efficient implementation of object-oriented concurrent computing system on a data-driven computer. In *Proc. of Joint Symposium on Parallel Processing 1992(JSPP'92), Yokohama, Japan*, pages 171–178, June 1992. (in Japanese).

[28] M. Yasugi, S. Matsuoka, and A. Yonezawa. ABCL/onEM-4: An efficient implementation of object-oriented concurrent computing system on a data-driven parallel computer. In *Object-Oriented Computing I (WOOC'92)*, pages 289–304. Kindai Kagaku Sha, July 1993. (in Japanese).

[29] M. Yasugi and A. Yonezawa. An object-oriented concurrent algorithm for N-body problem. In *Proc. of JSSST 8th Annual Convention, Sapporo, Japan*, pages 405–408, September 1991. (in Japanese).

[30] M. Yasugi and A. Yonezawa. An object-oriented parallel algorithm for the Newtonian N-body problem. Technical Report TR92-6, Department of Information Science, Faculty of Science, University of Tokyo, 1992.

[31] M. Yasugi and A. Yonezawa. Towards performance evaluation of a N-body problem algorithm in an object-oriented concurrent language on a date-driven parallel computer. In *Object-Oriented Computing II (WOOC'93)*. Kindai Kagaku Sha, 1994. (to appear) (in Japanese).

[32] A. Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System — Theory, Language, Programming, Implementation and Application.* The MIT Press, 1990.

[33] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In *Proc. of ACM Conference on OOPSLA*, pages 258–268, 1986.

[34] A. Yonezawa, S. Matsuoka, M. Yasugi, and K. Taura. Efficient implementations of concurrent object-oriented languages on multicomputers. In *PARALLEL PROGRAMMING SYSTEMS*, pages 50–52. World Scientific, May 1992.

[35] A. Yonezawa, S. Matsuoka, M. Yasugi, and K. Taura. Implementing concurrent object-oriented languages on multicomputers. *IEEE Parallel & Distributed Technology*, 1(2):49–61, May 1993.

[36] A. Yonezawa and I. Ohsawa. Object-oriented parallel parsing for context free grammars. In A. Yonezawa, editor, *ABCL: An Object-Oriented Concurrent System*, chapter 11. The MIT Press, 1990.

[37] A. Yonezawa and M. Tokoro, editors. *Object-Oriented Concurrent Programming*. The MIT Press, 1987.

Appendix A

ABCL/ST Program of N-body Algorithm

```
;;; Program for N-Body Problem  ( 1-D case )
;;; Written by Masahiro YASUGI. ( yasugi@is.s.u-tokyo.ac.jp )
;;; Dec. 22 1993
(defconstant G 1.0) ; normalization is assumed
(defconstant tstep 0.01)
(defconstant ep 0.01)
(defconstant appro 5.0)
(deftype particle
    (union [:noparticle]
           [:n int :mass real :pos real :vel real :pre-dt real]))
(deftype (obj-msg Tree-Cons-Obj)
    (union [:n-waits int]
           [:target Tree-Cons-Obj]
           [:data int particle Tree-Cons-Obj]
           [:suicide]
           [:forward Tree-Cons-Obj Tree-Cons-Obj]))
(deftype (obj-msg Force-Calc-Obj)
    (union (obj-msg Tree-Cons-Obj)  ;; inherit the interface of Tree-Cons-Obj
           [:bottom-up-data real real]
           [:term]
           [:link-lchild bool real Force-Calc-Obj]
           [:link-rchild bool real Force-Calc-Obj]
           [:add-links bool real Force-Calc-Obj bool real Force-Calc-Obj]
           [:accel-calc int real]
           [:spread-this-info int real real]
           [:up-this-info int real real real real]
           [:down-this-info int real real real real]
           [:construct Tree-Cons-Obj real]))
(deftype (obj-msg Printer-Obj)
    (union [:print-int int]
           [:print-real real]))
(deftype collist (union [:nil] [:col Tree-Cons-Obj :next collist]))
[class Tree-Cons-Cell Tree-Cons-Obj ((real size pos))
  (state (int (m-num 0) (n-new 0))
         (particle (pa [:noparticle]))
         (Tree-Cons-Obj lchild rchild)
         (real (size-half (* size 0.5)))
         (collist (oldobj-list [:nil])))
```

```
(wait-for
 (=> [:n-waits n]
     (dotimes (n-count n)
       (wait-for
        (=> [:data mn p oldobj]
            (incf m-num mn)
            (if (= mn 1) [pa := p])
            (if (> mn 0)
                (progn
                  (incf n-new)
                  (if oldobj
                      [oldobj-list := [:col oldobj :next oldobj-list]]))
                [oldobj <= [:suicide]]))))))
(if (/= m-num 1) [pa := [:noparticle]])
(tagbody
   (if (= n-new 0)
       (progn
         (wait-for
          (=> [:target nobj]
              [nobj <= [:data 0 [:noparticle] nil]]))
         (go exit)))
   (if (= n-new 1)
       (progn
         (match oldobj-list
           (=> [:nil]
               (go cont))
           (=> [:col oobj :next olist]
               (wait-for
                (=> [:target nobj]
                    [nobj <= [:data m-num pa oobj]]))))
         (go exit)))
 cont
   (if (> m-num 1)
       (progn
         ;; create child objs
         [[lchild rchild] := [(new-Tree-Cons-Cell size-half
                                                  (- pos size-half))
                              (new-Tree-Cons-Cell size-half
                                                  (+ pos size-half))]]
         [[lchild rchild] <= [:n-waits n-new]]
         (tagbody loop
            (match oldobj-list
              (=> [:nil])
              (=> [:col oobj :next olist]
                  [oobj <= [:forward lchild rchild]]
                  [oldobj-list := olist]
                  (go loop)))))
       (tagbody loop
          (match oldobj-list
            (=> [:nil])
            (=> [:col oobj :next olist]
                [oobj <= [:suicide]]
                [oldobj-list := olist]
                (go loop)))))
   (wait-for
    (=> [:target nobj]
        [nobj <= [:data m-num pa Me]]))
   (wait-for
    (=> [:suicide])
    (=> [:forward ls rs]
        (if (= m-num 1)
            (progn
              ;; create child objs in response to the forward message
              [[lchild rchild] := [(new-Tree-Cons-Cell size-half
                                                       (- pos size-half))
```

```
                                (new-Tree-Cons-Cell size-half
                                                   (+ pos size-half))]]
             (match pa
               (=>[:n n :mass mass :pos pa-pos :vel vel :pre-dt pre-dt]
                  (if (>= pa-pos pos)
                      (progn
                        [lchild <= [:n-waits 0]]
                        [rchild <= [:n-waits 1]]
                        [rchild <= [:data 1 pa nil]])
                      (progn
                        [rchild <= [:n-waits 0]]
                        [lchild <= [:n-waits 1]]
                        [lchild <= [:data 1 pa nil]]))))))
          [lchild <= [:target ls]]
          [rchild <= [:target rs]]))
    exit)]

[class Force-Calc-Cell Force-Calc-Obj
    ((real size pos) (Force-Calc-Obj dad) (Printer-Obj pr))
  (state
   (int (m-num 0) (term-count 0) total-num (curr-num 0))
   (real (size-half (* size 0.5)) (m-sum 0.0) (m-center 0.0) dt (accel 0.0))
   (particle (pa [:noparticle]))
   (Force-Calc-Obj lchild rchild left-side right-side))
  (tagbody loop
     (wait-for
      ;;
      ;; for tree construction
      ;;
      (=> [:data mn p oldobj]   ; from collector object
          [m-num := mn]
          (if (= m-num 1)
              (progn
                [pa := p]
                (match pa
                  (=> [:n n :mass mass :pos pos :vel vel :pre-dt pre-dt]
                      [m-sum := mass]
                      [m-center := pos]))))
          (if (> m-num 1)
              (progn
                [[lchild rchild]
                 := [(new-Force-Calc-Cell size-half
                                          (- pos size-half) Me pr)
                     (new-Force-Calc-Cell size-half
                                          (+ pos size-half) Me pr)]]
                [oldobj <= [:forward lchild rchild]]
                (wait-for
                 (=> [:bottom-up-data son1-m-sum son1-m-center]
                     (wait-for
                      (=> [:bottom-up-data son2-m-sum son2-m-center]
                          [m-sum := (+ son1-m-sum son2-m-sum)]
                          [m-center
                           := (/ (+ (* son1-m-sum son1-m-center)
                                    (* son2-m-sum son2-m-center))
                                 m-sum)])
                      ))
                 ))
              [oldobj <= [:suicide]])
          [dad <= [:bottom-up-data m-sum m-center]])
      (=> [:term]
          ;; check termination of
          ;; 1. lateral link
          ;; 2. delivering cell data
          (incf term-count)
          (if (= term-count 2)      ; both of lchild, rchild
              (progn [dad <= [:term]] [term-count := 0])))
```

```
;;
;; for adding lateral links
;;
(=> [:link-lchild p pos s]
    [lchild <= [:add-links p pos s (not rchild) 0.0 rchild]])
(=> [:link-rchild p pos s]
    [rchild <= [:add-links (not lchild) 0.0 lchild p pos s]])
(=> [:add-links lp lpos ls rp rpos rs]
    [left-side  := ls]
    [right-side := rs]
    (if lp
        [Me <= [:link-lchild lp lpos
                             (if (and left-side
                                      (>= (+ lpos size) (- pos size)))
                                 left-side nil)]]
        (let ((bool (p (not lchild))))
          [left-side <= [:link-rchild
                         p m-center
                         (if (and p (= m-num 1)) Me lchild)]]))
    (if rp
        [Me <= [:link-rchild rp rpos
                             (if (and right-side
                                      (<= (- rpos size) (+ pos size)))
                                 right-side nil)]]
        (let ((bool (p (not rchild))))
          [right-side <= [:link-lchild
                          p m-center
                          (if (and p (= m-num 1)) Me rchild)]]))
    (if (not lchild) [dad <= [:term]]))
;;
;; for delivering cell information.
;;
(=> [:accel-calc t-num t-dt]
    (if (= m-num 1) (progn [total-num := t-num] [dt := t-dt]))
    (if (> m-num 0)
        [dad <= [:spread-this-info m-num m-sum m-center]]
        [dad <= [:term]])
    [[lchild rchild] <= [:accel-calc t-num t-dt]])
(=> [:spread-this-info mn ms mc]
    [dad <= [:up-this-info mn ms mc m-center (* appro size)]]
    ;; # of particle, sum of mass, center of mass,
    ;; dad's center of mass, size of distribution range
    )
(=> [:up-this-info mn ms mc dmc sz]
    (if (>= sz size)
        [dad <= [:up-this-info mn ms mc dmc sz]]
        (progn
          [Me <= [:down-this-info mn ms mc dmc sz]]
          (if (< dmc pos)
              [left-side <= [:down-this-info mn ms mc dmc sz]]
              [right-side <= [:down-this-info mn ms mc dmc sz]]))))
(=> [:down-this-info mn ms mc dmc sz]
    (let ((real (hsz (* sz 0.5))))
      (if (= mn 1) [hsz := 0.0])
      (if lchild
          (if (and (> (+ dmc sz) (- pos size))
                   (<= (- dmc sz) (+ pos size))
                   (or  (<= (+ mc hsz) (+ pos size))
                        (> (- mc hsz) (- pos size))))
              [[lchild rchild] <= [:down-this-info mn ms mc dmc sz]])
          (if (and (= m-num 1)
                   (>  (+ dmc sz) m-center)
                   (<= (- dmc sz) m-center)
                   (or  (<= (+ mc hsz) m-center)
                        (>  (- mc hsz) m-center)))
              (progn
```

```
                    ;;
                    ;; acceleration calculation
                    ;;
                    (incf accel (* G ms
                                   (let ((real (x (- mc m-center))))
                                     (if (= x 0.0)
                                         x
                                         (let ((real (absx (if (> x 0.0) x
                                                                (- 0.0 x)))))
                                           (/ x (+ (* absx absx absx)
                                                   ep)))))))
                    (incf curr-num mn)
                    (if (= curr-num total-num)
                        (progn
                          (match pa
                            (=>[:n n :mass mass :pos pos :vel vel
                                   :pre-dt pre-dt]
                               (let* ((real
                                        (new-vel
                                         (+ vel (* accel (+ dt pre-dt) 0.5)))
                                         (new-pos
                                          (+ pos (* new-vel dt)))))
                                 [pr <= [:print-real new-pos]]
                                 [pa := [:n n :mass mass
                                            :pos new-pos
                                            :vel new-vel :pre-dt dt]])
                               ))
                          [dad <= [:term]])))))))
    ;;
    ;; for next tree construction
    ;;
    (=> [:construct collect-to space-size]
        (let ((Tree-Cons-Obj (new-self (new-Tree-Cons-Cell space-size 0.0))))
          (if lchild
              (progn [new-self <= [:n-waits 2]]
                     [[lchild rchild] <= [:construct new-self space-size]])
              (progn  [new-self <= [:n-waits 1]]
                      [new-self <= [:data m-num pa nil]]))
          [new-self <= [:target collect-to]]))) (go loop))]
[class miniPrinter Printer-Obj ()
  (tagbody loop
     (wait-for
      (=> [:print-int i]
          (print i))
      (=> [:print-real r]
          (print r))) (go loop))]
[class main Force-Calc-Obj ((int np nl))
  (let*
      ((Tree-Cons-Obj obj0 new-collector cur-collector)
       (Force-Calc-Obj root0 next1)
       (Printer-Obj (pr (new-miniPrinter)))
       (real (a (/ 1.0 np)) (p (* a -0.5)) pos (dt tstep)
             (space-size (* np 0.5)))
       (int (total-num 0)))
    (dotimes (i np)                       ; # of particle
      [new-collector := (new-Tree-Cons-Cell space-size 0.0)]
      [obj0 := (new-Tree-Cons-Cell space-size 0.0)]
      (incf p a)
      [pos := (* (- (* 2.0 p) 1.0) space-size)]
      [pr <= [:print-real pos]]
      [obj0 <= [:data 1
                      [:n (incf total-num) :mass 1.0 :pos pos
                          :vel 0.0 :pre-dt dt]
                      nil]]
```

```
  [obj0 <= [:n-waits 1]]
  [obj0 <= [:target new-collector]]
  (if cur-collector
      (progn
        [cur-collector <= [:target new-collector]]
        [new-collector <= [:n-waits 2]])
      [new-collector <= [:n-waits 1]])
  [cur-collector := new-collector])
[root0 := (new-Force-Calc-Cell space-size 0.0 Me pr)]
[cur-collector <= [:target root0]]
[cur-collector := nil]
(wait-for (=> [:bottom-up-data ms mc]))
[pr <= [:print-int (get-utime)]]     ; end-first-tree-construction
(dotimes (k nl)
  ;; next tree construction
  [next1 := (new-Force-Calc-Cell space-size 0.0 Me pr)]
  [root0 <= [:construct next1 space-size]]
  [root0 := next1]
  (wait-for (=> [:bottom-up-data ms mc]))
  [pr <= [:print-int (get-utime)]]  ; end-tree-construction
  ;; acccel calc
  [root0 <= [:add-links true 0.0 nil true 0.0 nil]] ; add lateral links
  (wait-for (=> [:term]))
  [pr <= [:print-int (get-utime)]]  ;end-adding-lateral-link
  [root0 <= [:accel-calc total-num dt]] ; trigger acceleration calc
  (tagbody loop
     (wait-for
      (=> [:spread-this-info mn ms mc]
          [root0 <= [:down-this-info mn ms mc 0.0
                                     (* 2.0 appro space-size)]])
      (=> [:up-this-info mn ms mc dmc sz]
          [root0 <= [:down-this-info mn ms mc dmc sz]])
      (=> [:term] (go exit)))
     (go loop) exit)
  [pr <= [:print-int (get-utime)]]  ; end-accel-calc
  )
[pr <= [:print-int (get-utime)]]     ; end
)]
```