# MultiScheme:

A Parallel Processing System
Based on MIT Scheme

by

James Slocum Miller

S.B. Massachusetts Institute of Technology
(1976)
M.S. University of Alaska, Fairbanks
(1982)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

at the
Massachusetts Institute of Technology
August, 1987

©Massachusetts Institute of Technology, 1987

Signature of Author _____
Department of Electrical Engineering and Computer Science
May 19, 1992

Certified by _____
Robert H. Halstead, Jr.
Associate Professor of Electrical Engineering
Thesis Supervisor

Accepted by _____
Arthur C. Smith
Chairman, Department Committee

**Abstract**

MultiScheme is a fully operational parallel-programming system based upon the Scheme dialect of Lisp. Like its Lisp ancestors, MultiScheme provides a conducive environment for prototyping and testing new linguistic structures and programming methodologies. MultiScheme supports a diverse community of users who have a wide range of interests in parallel programming. MultiScheme's flexible support for system-based experiments in parallel processing has enabled it to serve as a development vehicle for university and industrial research. At the same time, MultiScheme is sufficiently robust, and supports a sufficiently wide range of parallel-processing applications, that it has become the base for a commercial product, the Butterfly Lisp System produced by BBN Advanced Computers, Inc.

**Keywords:** Parallel computing, Lisp, Scheme, garbage collection, speculative parallelism, futures, placeholders

This research was supported in part by the Defense Advanced Research Projects Agency and was monitored by the Office of Naval Research under contract numbers N00014–83–K–0125, N00014–84–K–0099, N00014–86–K–0180, and MDA903–84–C–0033. Additional funds and resources were provided by BBN Advanced Computers Inc., and the Hewlett-Packard Corporation.

## Acknowledgements

The work described here could not have been undertaken without the help and support of a number of people. First among these is my wife Barbara, who never complained (well, almost never) about the strange hours I kept and the even stranger friends I acquired. She kept my entire support system alive and running when my mind was turned elsewhere. She helped create and maintain the MIT Scheme Team — perhaps the single best legacy I leave behind.

In addition, there has been a long line of students with whom I have worked, sometimes on MIT Scheme, sometimes on MultiScheme, and sometimes on their own research projects: Laura Bagnall, Andy Berlin, Todd Cass, Mathews Cherian, Stewart Clamen, Anthony Courtemanche, Morry Katz, Peter Nuth, Randy Osborne, Curtis Scott and Henry Wu. Their help and inspiration have kept me going when my own interest or ability lapsed.

The members of the BBN Butterfly Lisp group (especially Don Allen and Seth Steinberg) have contributed their time, energy and the project's money to backing the MultiScheme project. This collaboration has been mutually beneficial, and has certainly been one of the reasons that MultiScheme has become as solid a system as it currently is. Chris Hanson and Bill Rozas of the MIT Scheme project provided major technical help, and have been very supportive of my sometimes flaky implementation efforts.

Both my thesis supervisor, Bert Halstead, and my readers Gerry Sussman and Bill Dally have spent time with me, endlessly helping me repair my poorly worded text and poorly constructed reasoning. Bill Siebert has been a major source of help as I have confronted various parts of the MIT bureaucracy and was a great comfort in helping me adjust: first to the life of a staff member, then the life of a graduate student, and finally to the transition away from MIT.

Finally, there are three people to whom I owe a debt of gratitude which I will never be able to repay. Bill Rozas has been a friend through thick and thin, from our very first meeting. I stand completely in awe of his technical ability. Hal Abelson has provided both the emotional support and the detailed critique of my writing that I could find nowhere else at MIT. And Mike Eisenberg, despite his own tremendous backlog of writing and research activities, has spent hours and hours and hours helping me prepare the final draft of this report.

# Contents


**1 Introduction** **9**

1.1 Engineering Notes . . . . . 10
1.2 Origin . . . . . 13
1.3 Initial Conditions . . . . . 13
1.3.1 Parallel Computation . . . . . 15
1.3.2 Event-Driven Computation . . . . . 17
1.3.3 Explicit Parallelism . . . . . 19
1.3.4 Automated Resource Allocation . . . . . 20
1.3.5 Scheme as a Base Language . . . . . 21
1.4 Overview of the Thesis . . . . . 22
1.5 Summary . . . . . 23

**2 Extended Sequential Scheme** **27**

2.1 Placeholders: A New Data Type . . . . . 28
2.1.1 What Does "Invisible" Mean? . . . . . 30
2.1.2 Example: A (Contrived) Data Base Example . . . . . 31
2.2 Talking to the Garbage Collector . . . . . 34
2.2.1 Garbage Collection Demons . . . . . 35
2.2.2 Weak Pointers . . . . . 36
2.2.3 Object Finalization . . . . . 41
2.3 Example 1: Logic Variables in Lisp . . . . . 44
2.4 Example 2: Normal Order Evaluation . . . . . 47
2.5 Example 3: Stream Processing . . . . . 51
2.5.1 Uniformly Delayed Arguments . . . . . 51
2.5.2 AMB and Fair Merge . . . . . 53
2.6 Summary . . . . . 55

**3 Parallel Processing Extensions** **57**
3.1 Processor Coordination . . . . . 58
3.1.1 Starting Garbage Collection . . . . . 58
3.1.2 Language Extensions for Coordination . . . . . 59
3.1.3 `Pause-Everything` . . . . . 60
3.1.4 `Within-Task` . . . . . 63
3.2 Minor Extensions . . . . . 64
3.3 Per-Task Storage . . . . . 65
3.3.1 Packaging and Lexical Scoping . . . . . 66
3.3.2 Dynamic Binding . . . . . 67
3.3.3 Fluid Variables . . . . . 71
3.3.4 Fluid Variables in a Parallel System . . . . . 72
3.3.5 Task-Private Storage . . . . . 74
3.4 Introducing the Task . . . . . 76
3.5 Demand-Driven Computation . . . . . 81
3.6 Summary . . . . . 84

**4 Implementing the Scheduler** **87**
4.1 Scheduler Data Structures . . . . . 88
4.1.1 Placeholders . . . . . 89
4.1.2 Tasks . . . . . 91
4.1.3 Runnable Task Queue . . . . . 93
4.2 Overall Concepts and Utility Routines . . . . . 94
4.2.1 Atomicity . . . . . 94
4.2.2 Task Switch . . . . . 96
4.2.3 Other Utility Routines . . . . . 100
4.3 Task Creation and Termination . . . . . 102
4.3.1 Ordinary Task Creation . . . . . 103
4.3.2 Alternative Ways to Create a Task . . . . . 104
4.3.3 Task Termination . . . . . 107
4.4 Suspending a Task . . . . . 108
4.5 Storing the Value of a Placeholder . . . . . 112
4.6 Single Task Interludes . . . . . 114
4.7 Summary . . . . . 116

**5 Examples** **117**
5.1 Speculative Computation . . . . . 118
5.1.1 Parallel Control in a Rule Interpreter . . . . . 119
5.1.2 Speculative Computation: Summary . . . . . 124
5.2 Object Interaction Simulator . . . . . 125
5.2.1 Simple Sequential Implementation . . . . . 126
5.2.2 Parallel Particle Interaction . . . . . 132
5.2.3 Particle Interaction: Summary . . . . . 139
5.3 Parallel Pipelining . . . . . 140
5.3.1 Building the Pipeline . . . . . 142
5.3.2 Pipe Vectors and Objects in the Pipe . . . . . 146
5.3.3 Integration, Pipeline Style . . . . . 153
5.3.4 Parallel Pipeline: Performance . . . . . 157
5.3.5 Parallel Pipeline: Summary . . . . . 159
5.4 Other Methodologies . . . . . 161
5.4.1 QLambda . . . . . 161
5.4.2 Fork and Join . . . . . 167
5.4.3 Uniform System . . . . . 168
5.4.4 Other Methodologies: Summary . . . . . 172

**6 Conclusion** **175**
6.1 What Have We Wrought? . . . . . 175
6.2 What Remains To Be Done . . . . . 177
6.2.1 Performance Enhancements . . . . . 177
6.2.2 User Interface . . . . . 179
6.3 Parting Shots . . . . . 180

**Bibliography** **183**

**A Fundamentals of Scheme** **189**
A.1 Introduction to Lisp . . . . . 190
A.2 Scheme as a Dialect of Lisp . . . . . 193
A.2.1 The Standard Scheme Dialect . . . . . 195
A.2.2 MIT Extensions to Scheme . . . . . 195
A.3 Procedures as Objects . . . . . 199
A.4 Continuations as Objects . . . . . 200
A.4.1 Continuations: An Introduction . . . . . 201

A.4.2 Continuations in the Scheme Language . . . . . 202
A.4.3 Continuations: One Implementor's View . . . . . 205
A.5 MIT Scheme as a System . . . . . 210
A.5.1 Machine Model or Core Interpreter . . . . . 211
A.5.2 Primitive Procedures . . . . . 212
A.5.3 System Code . . . . . 213
A.6 Summary . . . . . 214

**B Implementation of the Pipeline** **217**
B.1 Locks for Serializing Access . . . . . 217
B.2 General Utilities . . . . . 218
B.3 Hash Tables . . . . . 218
B.4 Simple Pipeline Constructors . . . . . 220
B.5 Complicated Pipeline Constructors . . . . . 220
B.6 Creating a Pipe Vector . . . . . 223
B.7 Task Creation and Removal . . . . . 223
B.8 Making an Object . . . . . 225

**C Performance Measurements** **227**
C.1 The Measured System . . . . . 227
C.2 Measurement Technique . . . . . 229
C.3 Analysis: The Cost of a `future` . . . . . 230

**D Exception Handling in MultiScheme** **233**
D.1 Twin Notions . . . . . 234
D.2 Exception Handling . . . . . 235
D.3 Control State . . . . . 237
D.4 Dynamic State . . . . . 238
D.5 Connecting the States . . . . . 239
D.6 Exception Handling via State Space . . . . . 241

# List of Figures


2.1 Copying a cons cell . . . . . 39
2.2 Copying a weak-cons . . . . . 40

3.1 Simplified Code for `Pause-Everything` . . . . . 62
3.2 Simplified Code for `Within-Task` . . . . . 64
3.3 A Simple Package . . . . . 68
3.4 Multi-level Exit using `Call-With-Current-Continuation` . . 82
3.5 A Replacement for `Call-With-Current-Continuation` . . . . 83

4.1 Placeholder Data Structure . . . . . 89
4.2 Task Data Structure . . . . . 91
4.3 Saving state for future execution: `Store-My-State` . . . . . 97
4.4 Relinquishing the processor: `Release-Task` . . . . . 98
4.5 Activating a chosen task: the `run` procedure . . . . . 99
4.6 Tri-state Flag Representation . . . . . 100
4.7 Activating a waiting task . . . . . 101
4.8 Code for `Saving-State` . . . . . 102
4.9 Simplified Code for `disjoin` . . . . . 106
4.10 Code for `Await-Placeholder` . . . . . 111
4.11 Code for `determine!` . . . . . 113
4.12 `Make-Returned-Object`, support for `pause-everything` . . . 115

5.1 Recursive Control Structure . . . . . 122
5.2 A Simple Mathematics Library . . . . . 127
5.3 Operations on Particles (serial) . . . . . 128
5.4 Operations on Systems of Particles (serial) . . . . . 128
5.5 Differential System Generator . . . . . 130
5.6 Integration Procedure (serial) . . . . . 131

5.7 Parallelism in Particle Force . . . . . 134
5.8 Parallelism in Adding Systems . . . . . 136
5.9 Parallelism in Scaling Systems . . . . . 137
5.10 Details of Scaling Systems . . . . . 138
5.11 Creating Vector Elements . . . . . 152
5.12 Pipeline Version of `Integrate-System` . . . . . 156
5.13 `QLambda` with `normal` Eagerness . . . . . 164
5.14 `QLambda` with `eager` Eagerness . . . . . 166
5.15 Uniform System Simulator . . . . . 170
5.16 Core of `Fact` for Uniform System . . . . . 171

A.1 Standard Scheme Syntax Used in Report . . . . . 194
A.2 Standard Procedures Used in Report . . . . . 196
A.3 A Simple Scheme Program . . . . . 197
A.4 MIT Syntax Used in Report . . . . . 198
A.5 Three Ways to Define a Procedure-Generator . . . . . 200
A.6 Use of `Make-Incrementer!` . . . . . 201
A.7 The Semantic Function for `if` . . . . . 202
A.8 Simple Use of `Call-With-Current-Continuation` . . . . . 204
A.9 Scheme's Stack vs Pascal's Stack . . . . . 207

C.1 Internal Timing Measurements . . . . . 231
C.2 Components of the Cost of a Future . . . . . 232

# Chapter 1

# Introduction

MultiScheme is a fully operational parallel-programming system based upon the Scheme dialect of Lisp. Like its Lisp ancestors, MultiScheme provides a conducive environment for prototyping and testing new linguistic structures and programming methodologies. MultiScheme supports a diverse community of users who have a wide range of interests in parallel programming. MultiScheme's flexible support for system-based experiments in parallel processing has enabled it to serve as a development vehicle for university and industrial research. At the same time, MultiScheme is sufficiently robust, and supports a sufficiently wide range of parallel-processing applications, that it has become the base for a commercial product, the Butterfly Lisp System produced by BBN Advanced Computers, Inc.

This report pursues three interrelated topics that are elucidated by the MultiScheme work. First, we describe extensions that suffice to transform sequential Scheme into a powerful vehicle for multiprocessing. Gratifyingly, only a few extensions are needed. The two most important extensions are the inclusion of placeholders — data types used for representing values that have not yet been computed — and the use of the garbage collector as a visible part of the language. Placeholders are the implementation base for the future construct of Halstead's Multilisp[24] language. In MultiScheme, however, they are strictly a data type rather than a combination of data type, task, and syntax as in Multilisp. Most of the other extensions have been familiar as "folklore" within the Lisp community. Our essential contribution here is to present carefully-documented, robust implementations and to demonstrate the synergy of including these within one coherent system. In particular,

the combination of placeholders, controlled garbage collection, and a few operations for handling processor synchronization forms a sufficient base for implementing the MultiScheme scheduler.

Secondly, we argue that even in the sequential context, adding these constructs significantly extends the expressive power of the Scheme language. Placeholders, for example, allow one to embed logic variables within standard Scheme data structures. They also provide a mechanism for mingling normal-order evaluation with an otherwise applicative-order interpreter. The visible aspects of the garbage collector provide a convenient base for the implementation of populations[47] (also known as "weak sets"), association tables (an extension of Lisp's "property lists" to objects other than symbols), and producer-to-consumer links (vital to automatic deallocation of tasks).

Finally, we demonstrate how MultiScheme allows one to attack a wide range of problem domains and to investigate many different methodologies for parallel programming. For example, by measuring the time required to compute the future state of an 8-body gravitational system we demonstrate how appropriately adding `futures` can convert a serial solution to a parallel solution that runs 6.6 times faster (on a 16 processor machine). By studying the utilization of the processors in this solution we are led to a variant on the dynamic dataflow methodology for solving the same problem. By expressing the same problem in this new methodology we incur 17% overhead in simulating the dataflow processor, but are able solve the problem 6.9 times faster than the original serial solution. Other methodologies that can be conveniently expressed within MultiScheme include fork/join structures, Gabriel and McCarthy's `QLambda`[21], and the Butterfly `Uniform System`[56].

## 1.1 Engineering Notes

Building MultiScheme was an exercise in both research and nuts-and-bolts engineering. For people who are interested in the engineering aspects of parallel-programming systems, a number of important lessons are described throughout this report. This section surveys these lessons in the hopes that it will alert interested readers to points that might otherwise be overlooked.

**Garbage Collection**

Perhaps the most interesting engineering result comes from attempting

to implement a system for garbage collecting useless tasks. The intention was to provide support for speculative computation — trying a number of alternative ways of solving the same problem and selecting the result of the first method to succeed. The idea of extending the garbage collector to collect useless tasks is not new (see, for example, [4]), but to our knowledge, MultiScheme provides the first working implementation of this idea. In the process of building such a garbage collector several small discoveries were made. Most importantly, the choice of the root for garbage collection is critical to the success of such a system (Section 3.5). MultiScheme uses a root that combines the global name space (the ordinary root for garbage collection) and those tasks created explicitly to report a value back to the user (i.e. tasks running the standard read-eval-print loop).

Furthermore, the data structures of the system must be carefully crafted so that the garbage collector "sees" only tasks that are performing useful work. In MultiScheme, this required the introduction of weak cons cells (Section 2.2.2) to allow the producer of a value to locate the consumers of that value without retaining the consumers after a garbage collection. Even with all of this support in place, the garbage collector doesn't always "do the right thing" since some tasks operate by side-effect and I have not found a mechanism for detecting when they are no longer needed (see the example of Section 5.3).

One additional result related to garbage collection deserves mention here but will not be described in the body of the report itself. A pair of memory management strategies were developed. Each of these includes a parallel stop-and-copy garbage collection algorithm along with related decisions dealing with the mechanism for distributing the physical memory across the shared address space and the area of memory available to each processor for allocating to user tasks. The initial MultiScheme memory management system is described by Courtemanche[12], including details of the garbage collection algorithm and measurements of its performance. Based on this work, Johnson[33] wrote and tested a different memory management system designed to repair some of the problems encountered by Courtemanche.

**Weak Cons Cells and Object Finalization**

Weak cons cells (`cons` cells whose `car` becomes `'()` rather than retain an object no longer needed elsewhere) and object finalization (retaining an object for one garbage collection cycle after it would normally disappear), or similar mechanisms, have long been known to Lisp implementors. They have been employed internally in Lisp systems for at least twenty years. The particular form they take in MultiScheme has two unusual properties: it is safe enough for use by "application programmers" who have little knowledge of the internals of the Lisp implementation, and its implementation consisting of a modification to the garbage collector and a post-garbage collection update pass is clearly documented (in this report). Similarly, the notion of object finalization (Section 2.2.3) has been around the implementation community for a long time. Again, the implementation described here is safe, and the use of an additional garbage-collection root to accomplish the goal is described here.

**Processor Synchronization**

The author was surprised by the discovery that MultiScheme needed a pair of low-level operations for synchronizing processing elements. This is true even though MultiScheme presents no notion of individual processing elements to the user. However, operations such as starting a garbage collection flip (see Section 3.1.1), pausing all tasks when an error is encountered (Section 3.1.3), and interrupting a task that is currently running on another processor (Section 3.1.4) inherently require the cooperation of the processors in the system. The particular choice of synchronization operations (synchronizers and global interrupts) is not profound, but the fact that even this simple set provides an adequate base for implementation of the higher-level constructs demonstrates that only small extensions are needed to convert Scheme into a convenient systems programming base for a parallel processor.

**Error and Exception Handling**

When Hanson and Lamping[26] showed how Stallman's `dynamic-wind` operation[51] could be seen as a convenient packaging for the two independent notions of a dynamic-state space and a control-state space (continuations), they argued that this provides the base needed for both

error and exception handling. In extending Scheme into a parallel-processing environment it was important to understand the ramifications this would have on these same issues. Appendix D documents the analysis of this area and provides a more detailed argument in support of their original claim. As a result of this analysis, no new exception-handling facility was added to MultiScheme. Instead, the dynamic-state space implementation was modified to operate without side-effects to the runtime data structures.

## 1.2 Origin

The MultiScheme effort began as an attempt to merge the best parts of two existing systems. Loaiza and Halstead had developed the Multilisp system[24] as a means of testing ideas about parallelism in Lisp. Their system had already been implemented and tested on the MIT Concert parallel processor and served as a vehicle for ongoing research into parallel processing.

In addition, a group of hackers had developed MIT Scheme, a system used in teaching undergraduate courses. Led by Chris Hanson, this group was constantly revising and refining both the local Scheme language dialect and the system implementation. Chris is responsible for a large number of the system ideas that ultimately made their way into MultiScheme. His genius for taking a passing comment, studying it, and producing a beautifully worked out and elegantly abstracted implementation of the key idea is a truly rare gift.

In attempting to integrate the ideas from these two systems the advice, help, and support of Guillermo (Jinx) Rozas was invaluable. Jinx has contributed to the MultiScheme effort a wide variety of ideas at all levels, from implementation details to overall system structures. His untiring efforts and his blasted perfectionism have helped move both MIT Scheme and MultiScheme from toys to robust, reliable systems.

## 1.3 Initial Conditions

By way of background to the remainder of this report, this section serves to position the MultiScheme project with respect to other work in the field of

parallel computation. Although it is tempting to provide an overview of that entire field, the theme of increased computational speed through the use of multiple computation units has excited so much work in the last five years that it would take several books to provide such an overview. The survey book of Filman and Friedman[18] is an excellent introduction to the subject.

The strategy adopted here, instead, is to provide a description of the five major boundary conditions within which the work has proceeded:

**Parallel Computation**

MultiScheme, from the outset, was intended to be a system for a *parallel*, rather than a *distributed*, computing environment. There is a fine dividing line between these two; and the distinction I draw between them centers around their ability to share objects. This boundary condition has ramifications for computer architectures that support MultiScheme, and the matter is discussed further in Section 1.3.1.

**Event-Driven Computation**

The primary method for ensuring precedence constraints between user tasks is based on an *event-driven* mechanism. Section 1.3.2 compares this mechanism to two common alternatives, *busy waiting* and *polling*. The twin acts of touching a placeholder and providing a value for a placeholder are the means used to express precedence constraints and thereby realize the event-driven model in MultiScheme.

**Explicit Parallelism**

MultiScheme was intended as a base for a wide range of experiments in parallel programming methods. As such, it allows the programmer to indicate explicitly how the parallelism of the underlying hardware is to be exploited. This point is elaborated upon in Section 1.3.3.

**Automated Resource Allocation**

Writing parallel programs is a difficult task. MultiScheme provides a default mechanism that automates resource allocation, making the programmer's job significantly easier. MultiScheme's approach to processor allocation is discussed in Section 1.3.4.

**Scheme as a Base Language**

The Scheme dialect of Lisp was chosen as a base language for MultiScheme because of its conceptual simplicity. This simplicity allows a

large number of sequential-programming models to be expressed within the language; MultiScheme is intended to extend this expressive power into the parallel-programming domain. This point is discussed in Section 1.3.5

### 1.3.1 Parallel Computation

MultiScheme was intended from the outset to run in an environment consisting of many computation units operating independently but efficiently sharing access to a set of (mutable) objects. As a result, the programmers' model of the system need not include the notion of *explicit communication* between tasks. Rather, the programmer thinks about objects and operations on objects without an externally imposed constraint that each object have a particular task or processor as its "owner." I take these to be the distinguishing features of a parallel system: in a distributed system the programmer's paradigm is one of communication between tasks, with a concomitant notion of copying; in a parallel system the paradigm is sharing between potentially concurrent computations.

Fundamentally, this is a constraint on the class of architectures conducive to a MultiScheme implementation. An architecture must possess two critical features in order to make an acceptable implementation base:

1. A large portion of the address space must be shared by (accessible to) all processors in the system.

2. The speed of reference to all portions of the address space must be comparable.

One of the other initial conditions is the choice of the Scheme language (see Section 1.3.5). Scheme and other Lisps support a large variety of first-class objects by employing pointers to objects, rather than the objects themselves, as the fundamental units of data transfer. Any procedure with access to such a pointer, either by receiving it as an argument or by accessing it as the value of a lexically visible variable, must have complete access to the object itself. But efficient implementation of this, in turn, requires a flat address space containing all referenceable objects. It is a matter of folk wisdom in the Lisp community that implementations of Lisp on segmented architectures (like the Intel 8086) suffer significant performance penalties unless a single segment

is large enough to hold the entire heap or the hardware supports the ability to dereference a pointer to any location in the heap area (as in the Multics Maclisp implementation).

On a parallel processor, this same problem appears in a slightly different guise. The language still requires a flat reference space, with no distinction between objects residing "local" to a processor and those residing "remote" from the processor; and again, an architecture that imposes heavy penalties for "remote" address references will lead to inferior performance. Moreover, because so many kinds of objects in Scheme are first-class, the possibility of analyzing programs to decide where to allocate objects for maximum locality of reference seems unlikely to succeed in general. Thus, equally rapid access to all objects is required for such a system to have any likelihood of success.

Shared address space machines[1], such as the BBN Butterfly[49], MIT Concert[25], and IBM RP3[46] systems come close to satisfying the above criteria. They have both "local" and "remote" memory, but the hardware is fully responsible for detecting references to each and reacting accordingly. As a result, on the largest such systems (roughly 1000 processors), local memory is no more than one order of magnitude faster to reference than remote memory *including* distinguishing the two forms of reference. By contrast, the Intel iPSC is estimated to have a two- to three-order-of-magnitude difference in reference speeds because software detection of remote references is required.

This difference in relative access speeds is expected to be significant in a parallel environment as defined here. Since the programming model does not include the notion of local and remote objects corresponding to the actual memory configuration, it is reasonable to consider how these systems perform under conditions of uniformly distributed access patterns and (worse yet) totally remote access operations. Under these kinds of conditions[2] the effective number of processors is reduced linearly by the ratio of local to remote access speeds. Thus, a Butterfly (or Concert or RP3) with 1000 physical processors should be capable of performing roughly two orders-of-magnitude faster than

[1]Since the start of the MultiScheme project, a class of machines known as "message passing machines" has been proposed, of which the shared address machines are a special case. These machines span a wide range of architectures, and those with memory access times that meet the two criteria mentioned above form an appropriate base for MultiScheme.

[2]Just how accurate these assumed access patterns turn out to be in practice is subject to measurement for any given system. Nuth[42] describes one investigation into this area.

a single processor from which the system is made. An iPSC of the same size could be expected to perform at best one order-of-magnitude faster.

In addition to these shared memory architectures, there are two additional machines that satisfy the two criteria listed above: the original design of the Connection Machine, and the Monarch computer. The original proposal for the Connection Machine[31] had a large number of processors, each with a very small amount of local state, interconnected by a sophisticated routing network. From a programmer's point of view the machine appeared to have a very large address space and most of it was equally accessible with only a small amount of state information more easily reached. Unfortunately, the fact that the machine had a single instruction stream makes it incompatible with the system design discussed here.

The BBN Monarch[9] has a large address space supported solely at the remote end of a switching network. The goal is to make the memory appear to be a single, thousand-way interleaved, memory unit. In addition, the system supports tagged data making it extremely well suited to the implementation of a MultiScheme system as described in this report. Unfortunately, the machine is still in the design stages.

### 1.3.2 Event-Driven Computation

The ability to synchronize tasks is extremely important in any kind of multi-tasking system, and it becomes more important as the interaction between tasks becomes tighter. In a parallel processing system such as the one envisioned for MultiScheme this problem is at the heart of the system design. There are three common synchronization methodologies leading to three very different kinds of system structures: busy-waiting, polling, and event-driven. MultiScheme, like most real systems, uses all three techniques but the emphasis at the language (as opposed to system) level has been placed on the third.

**Busy-waiting**

Busy-waiting is a very simple mechanism used to serialize access to a critical resource. There is a lock for the resource and a task repeatedly tries to acquire the lock in order to obtain access to the resource. This organization is useful when the likelihood of contention for a lock is very small and the lock is released rapidly. MultiScheme employs

busy-waiting for some operations such as locking a placeholder (see Section 4.2.1) and to compensate for some unfortunate defects of the current hardware (it lacks atomic 32-bit swap operations). In most cases it is possible to include a lock with the individual object to be locked so contention only occurs if two tasks attempt to access the same resource simultaneously. Where this is not possible because the number of objects requiring locks is large (for example, it is possible to atomically swap a value with the value of a variable), the objects are grouped together and locks are provided for relatively small numbers of objects. In addition, of course, the critical regions (where the lock is held by one task) are kept as short as possible.

**Polling**

Polling is a more sophisticated system structure involving a periodic test for events requiring service. This is a common technique in microcoded machine architectures for implementing interrupts. It is also found in higher-level software to allow interrupting of potentially long loops (for example, the search loop in many editors checks for user interrupts). Unfortunately, polling-based systems, such as the Pluribus[44], are difficult to write and maintain. The software is forced into a model of short program strips, each of which polls for an event, services that event, saves its new state so that it can be resumed, and then returns to poll again. Writing such a system with an eye toward efficiency is tricky; information ordinarily available from the location in the code and the contents of the stack must be converted into a data structure used to dispatch on events. This is equivalent to converting the program into a finite state machine (with random access memory), a representation that is hard to both understand and modify.

While contention is generally less of a problem in a polling-based system than in one with busy-waiting, contention for the next available event becomes a problem as the number of processors increases. MultiScheme employs polling for servicing interrupt conditions and an early version of MultiScheme dramatically demonstrated the existence of this form of contention within the system. Combining networks[14] provide a hardware mechanism for removing this contention, but the hardware on which MultiScheme was running did not support this mechanism. Instead, a software solution has proven to be quite effective: rather than

provide one centralized location polled by all processors, MultiScheme provides a location for each processor. Signalling an interrupt requires either choosing to alert a particular processor or writing the flag separately for each processor. Signalling all processors is clearly much slower, but it happens infrequently. The polling that must occur often to reduce interrupt latency, however, no longer generates contention.

**Event-Driven**

A third organization can be used when a resource, once allocated, is likely to be in use for a relatively large amount of time. In this case it is reasonable to expect the task using the resource to alert other tasks when the resource becomes available. Tasks suspend themselves and release the processor to a different task when they discover that a resource they need is unavailable. It is this system organization that both Multilisp and MultiScheme attempt to make readily available through the use of futures and placeholders, respectively.

One common implementation of an event-driven system is the mailbox or inbox approach. In this organization (see, for example, the CMOS operating system[8]) each task has a specific location (the mailbox) where it can be notified of events as they occur. Tasks poll this mailbox but, unlike the polling organization, they are suspended if no event is available for processing. Two variations on this theme are the ability to wait for a specific set of events to occur and the ability to have multiple mailboxes (such as the events of the Chrysalis system[6]). MultiScheme differs from this approach by augmenting the explicit polling with the automatic `touch` supplied for placeholders (see Section 2.1.1) and by using an outbox rather than an inbox. Each task in MultiScheme has an associated placeholder, the goal it is trying to compute (see Section 3.4). Thus a task has a unique location where it can store a result but it can await the result computed by any other task. The "events" in MultiScheme are occurrences of the `determine!` operation (see Section 4.5) used to store a result into a placeholder.

### 1.3.3 Explicit Parallelism

The MultiScheme language enables programmers to annotate programs, indicating where the programs are permitted to execute in parallel with an

existing computation task. The decision to make parallel execution visible and under the control of the programmer reflects two major language design concerns: the language should encompass both functional and non-functional programming styles, and it must also provide program analysis tools (themselves MultiScheme programs) a mechanism to express opportunities for parallelism.

Scheme includes side-effecting operations designed to enhance the modularity of many common programming styles. As a result, the automated detection of potentially parallel computations is quite difficult. Katz[34] and Knight[35] describe rather elaborate architectures designed specifically to make such a system feasible. Rather than require this form of support from the underlying architecture, MultiScheme relies on the program to indicate explicitly where opportunities for parallel computation can be exploited without affecting the semantics of the program. It is entirely possible, in MultiScheme, to request parallelism in a manner that makes the results inconsistent with the serial execution of a program — a reflection of the fact that the additional expressive power must be used with care.

Furthermore, if the language is to serve as the *output* language of automated program analysis tools, then it must allow those tools some way to express the opportunities for parallel execution. The MultiScheme project has not created any such tools, but Gray[23] and Wang[59] have undertaken such projects for the Multilisp language. The job of these tools is to make engineering trade-offs between the time required to create, dispatch, and kill tasks and the time required to complete the computation represented by a single task.

### 1.3.4 Automated Resource Allocation

As a system, MultiScheme supports automated scheduling of tasks onto processors as well as the detection and removal of no longer useful tasks. The underlying system support for scheduling, discussed in Chapter 4, is designed to allow users a large degree of flexibility in controlling the scheduling policy. The problem of programming a multi-processor can be divided into two parts: subdividing a problem into components for parallel execution, and scheduling these components onto the available processors. The explicit parallelism constructs support this first task, and the automated resource management supports the second. Isolating these two components gives programmers the

freedom to design and test their programs independent of the resource allocation considerations by relying on the default allocation strategies of the standard scheduler. Later, when these allocation strategies become a performance concern, the scheduler can be molded to fit the particular application.

In addition to a flexible scheduler for handling allocation of processors to tasks, MultiScheme extends the Lisp garbage collector to remove useless tasks in addition to its traditional role of recycling memory. This allowed the MultiScheme work to explore the area of demand-driven computation within Lisp, as discussed in Section 3.4.

### 1.3.5 Scheme as a Base Language

Our investigation into parallel-programming systems might have begun by the *de novo* construction of a language for expressing parallel computations. Instead, we chose to use an existing language, Scheme (a dialect of Lisp), as our base. A number of important considerations went into this choice:

- Scheme is an excellent medium in which to experiment with a wide range of sequential-programming methodologies (see, for example, Abelson and Sussman[2]). An important goal of MultiScheme is to provide this same rich medium for experimentation in the parallel-programming domain.

- Lisps, and Scheme in particular, lend themselves to a largely functional style of programming. Thus, the introduction of parallelism into Scheme could be expected to be a straightforward change yielding a good deal of increased speed.

- The Lisp community's staple domains, symbolic computation and artificial intelligence, have been relatively unaffected by the availability of hardware capable of extremely high speed (even parallel) arithmetic computations. Parallel processing appears to be a good candidate for speeding up computations in these domains.

- In comparison to CommonLisp (the other likely candidate Lisp system), Scheme is a simpler language, with far fewer special forms and required procedures. Furthermore, because of its first-class continuation objects (see Section A.4), Scheme has a more powerful control structure.

- A language extension for parallelism (the futures of Multilisp, which became the placeholders of MultiScheme) consistent with Scheme had already been developed and demonstrated. A trio of alternative constructs (the `qlambda`, `qlet`, and `qcatch` of QLisp[21]) had been analyzed and was readily implemented in MultiScheme, suggesting that the placeholders were an important data structure in any case.

## 1.4 Overview of the Thesis

The main body of this document is structured to reflect the three major topics mentioned at the outset: extensions to the Scheme language (Chapters 2 and 3), the use of these extensions within the sequential language (second half of Chapter 2), and their use within the parallel-programming domain (Chapters 4 and 5).

Chapter 2 discusses the extended sequential Scheme language, derived from the MIT Scheme system by adding placeholders to represent uncomputed values and modifying the garbage collector to provide several services directly visible to the programmer. The latter half of the chapter consists of three examples, each of which uses this extended language to demonstrate a solution to some problem of current interest to the Lisp and functional languages community. These examples cover a range of topics: embedding logic variables in Lisp (Section 2.3), combining normal and applicative order evaluation (Section 2.4), and higher-order and non-deterministic streams processing (Section 2.5).

Chapters 3 and 4 demonstrate that extended sequential Scheme is very close to a language suitable for systems programming of parallel processors. The extensions needed for creating this systems-programming language are described primarily in Chapter 3. They comprise two distinct changes.

- The addition of a few new primitive operations for processor coordination (Section 3.1), task distribution, and miscellaneous processor-oriented operations (both in Section 3.2).

- The re-implementation of two existing mechanisms for use on a parallel processor. The first mechanism, fluid variables, is used to provide private storage for tasks (Section 3.3). The second, dynamic state spaces,

provides hooks for users to write exception and error handling facilities (Appendix D).

In order to demonstrate that the resulting language does, in fact, make a good systems programming language, Chapter 4 documents the implementation of the MultiScheme scheduler in detail. The scheduler is responsible for creating, swapping, and terminating tasks.

Chapter 5 concentrates on expressiveness of the MultiScheme system. It provides three different example programs implemented in MultiScheme. The first of these, a simple rule-based system, demonstrates how MultiScheme can be used for speculative computation. The second is a program to solve the $n$-body problem of classical mechanics, derived from a serial program by the addition of `future` constructs. The third is a more elaborate example showing how a vector pipeline similar to a dynamic dataflow programming style can be used to solve the same $n$-body problem. These examples demonstrate a variety of ways in which MultiScheme's facilities can be combined to provide prototype implementations in different parallel programming paradigms. Each example is fully worked out, and measurements of its performance relative to the serial version are supplied.

In addition to the main body of the report, there are three appendices. The first of these contains background material for readers who may not be familiar with the MIT Scheme system. The text of the report assumes that the reader is familiar with the details of the standard Scheme language described in [48] and the MIT extensions to that language. Appendix A is intended for readers who are either somewhat rusty in their knowledge of Lisp or who are familiar with a different dialect or implementation.

Appendix B contains the implementation details of the dataflow (vector pipeline) system described as part of the example of Section 5.3. It is supplied for those readers who are interested in understanding how such a radically different programming model can be implemented using the facilities of MultiScheme.

The final appendix contains a variety of performance measurements taken from a number of different versions of the MultiScheme system.

## 1.5 Summary

The MultiScheme work centers around three major topics:

1. A few extensions were made to the Scheme language in order to provide a base for experiments in parallel processing. The most important of these are:

   - A new data type, placeholders, can be used to represent values not yet computed (Section 2.1).
   - The garbage collector provides services that are directly visible to programs (Section 2.2).
   - Primitive are provided for coordinating the activities of the system's processors (Section 3.1).

2. This extended Scheme language exhibits new expressive power for serial computations. Sections 2.3, 2.4, and 2.5 provide illustrative examples.

3. The extended language provides a base for implementing a range of parallel-programming methodologies (Chapter 5). It also serves as an elegant systems programming language for parallel processing (Chapter 4).

In the process of building MultiScheme a number of interesting engineering results were discovered. These discoveries center around four topics, summarized in Section 1.1 and repeated here:

1. Garbage collection (Section 3.4).

2. Weak cons cells and finalization (Section 2.2).

3. Processor coordination (Section 3.1).

4. Error and exception handling (Appendix D).

Section 1.3 describes the five major constraints imposed on the MultiScheme project at the outset:

1. The area of exploration is parallel, not distributed, systems (Section 1.3.1). This requires that the underlying machine support an address space shared across all of the processors with comparable access times to all parts of the space.

2. Most user synchronization is based on an event-driven model of computation (Section 1.3.2). Programs use the automatically supplied `touch` for placeholders to augment explicit requests for synchronization. The `determine!` operation is the source of the "events" in the system. Busy waiting and event polling are also employed.

3. User programs explicitly specify where parallelism is to be employed (Section 1.3.3).

4. Users do not deal with the allocation of tasks to processors (Section 1.3.4). The system detects and removes no longer useful tasks and supplies a structure that makes it straightforward for users to specify scheduling policies.

5. Scheme, a dialect of Lisp, serves as the language base. The MIT Scheme system serves as the implementation base. (Section 1.3.5.)

# Chapter 2

# Extended Sequential Scheme

This chapter describes two extensions to the MIT Scheme system that provide additional expressive power on a sequential processor. This "Extended Sequential Scheme" is the base from which MultiScheme is derived. But it is an interesting language in its own right.

The first extension, described in Section 2.1, is the addition of an object to represent values that have not yet been computed. These placeholders are similar to the delayed objects of standard Scheme, but (unlike thunks or delayed objects) they appear to be replaced by the value they represent once it has been computed. They also form the implementation base on which the `future` construct of Multilisp[24] is built in MultiScheme.

The second extension, described in Section 2.2, is more of a change in spirit. In standard Scheme the garbage collector is a completely invisible system service. Programmers are completely unaware of its operation, and have no way of interacting with it. In extended sequential Scheme, however, the garbage collector can be used as an active partner in the computation. The addition of `weak cons` cells allows programmers to create data structures without implying that the objects in them must be maintained when the garbage collector next recycles memory. The addition of a user-specified garbage collection root allows users to ask the garbage collector to indicate when the useful lifetime of specific objects has passed.

The remainder of the chapter uses extended sequential Scheme to solve three problems. These problems have been raised by other researchers and are not conveniently solved using the initial MIT Scheme system. The first problem (Section 2.3) is the embedding of logic variables in Scheme data

structures. The goal is to allow unification of expressions containing logic variables to produce ordinary Scheme values in completely independent data structures through the sharing of these variables. Since a placeholder becomes the object it represents (once its value is known), they provide a simple solution to this problem.

The second problem (Section 2.4) is the introduction of limited amounts of normal order evaluation into an otherwise applicative order language. By using placeholders for selected arguments to procedures, a very fine degree of control can be exercised over the time vs. space trade-off inherent in the choice of evaluation order. Finally, section 2.5 solves a collection of difficulties that crop up when dealing with stream processing in either an applicative order language (higher-order stream functions are difficult to write) or in a functional language (fair merge is not a function). The placeholder is a convenient mechanism for solving both of these problems.

## 2.1 Placeholders: A New Data Type

The first and most pervasive extension made to the original MIT Scheme system was the addition of a new data type with some unusual properties. This type, for historical reasons called future in the MultiScheme implementation[1], is best described as a placeholder; it provides a way to represent an object whose value is not known yet or may be subject to change later. It provides a form of sharing difficult to obtain without direct language support: the same object can occur in many data structures, but the actual choice of object can be altered without knowing the data structures containing the object[2].

An alternative view (more related to the implementation than to language design goals) is to consider a placeholder as an "invisible indirect reference" to another object. Yet another view (helpful in explaining to people familiar with Scheme) is that a placeholder is very similar to a delayed object (or a promise) except that the force operation required to access the value is

[1] Futures were described by Baker and Hewitt[4] and formed the basis for the implementation of the Multilisp system described by Halstead[24]. Placeholders in MultiScheme are directly based on these concepts but differ in a number of ways.

[2] As will be discussed in Section 2.3, this form of sharing appears to be the critical idea behind the "logic variable" of Prolog. This form of variable, in turn, provides a great deal of the power and flexibility of the logic programming languages.

performed automatically when needed. Accessing the value of a placeholder is called touching the placeholder.

This last view points up two important notions. The first is that, just as in stream processing in Scheme, the placeholder can be used to denote an object whose value is not yet known. Thus a placeholder object has at least two states. A placeholder is said to be undetermined when the object it references is not yet known. Unlike the delayed object, however, a placeholder need not have an unchanging value. Thus, a placeholder actually has three states: undetermined, determined, and mutably determined. A mutably determined placeholder may have the object to which it refers changed by re-determining the placeholder, while a determined placeholder cannot change its value. Notice, also, that `delay` is a special form in Scheme whose evaluation results in a delayed object. The placeholder being discussed here is a data type and *not* a special form. That is, "delay" refers to a programming construct, while "delayed object" and "placeholder" refer to data objects manipulated by programs.

Since a placeholder can represent an object before the object even exists, it is logical to ask what happens when a program touches such a placeholder. This is controlled by the scheduler, a portion of the system code written in MultiScheme and hence under programmer control. This portion of the scheduler is discussed in detail in Section 4.4, but in general the idea is that the task performing the touch is suspended until the value of the placeholder has been determined.

The second point raised by the comparison with Scheme's delayed objects, explored in detail in Section 2.1.1, is the fact that a placeholder object is *automatically* touched by certain primitive operations and language constructs. In stream processing it is important that the use of `force` to retrieve the value of a delayed object be deferred as long as possible. This leads to a "lazy evaluator" where stream processing can be used as a model for call-by-need computations. The question of when an object can remain a placeholder and when it must take on an actual value is the topic of the next subsection.

The scheduler is also involved in the creation of placeholder objects (as described in Section 4.3). It determines how and when computing resources should be devoted to finding the value of the placeholder. When the placeholder object is used to implement the equivalent of a Scheme delayed object no resources are allocated at the time the placeholder is created. Instead, they are allocated when it is touched. Parallel execution in MultiScheme is

closely related to the use of placeholders, and it is largely the action of the scheduler when a placeholder is created that distinguishes stream processing (lazy evaluation) from parallel processing (eager evaluation). A parallel computation is initiated by creating an undetermined placeholder to contain (eventually) the value returned by that computation. The scheduler also creates a `task` to perform the computation and provides it a continuation that causes the task to `determine` the placeholder and then terminate. The (undetermined) placeholder is thus immediately available to the initiating task, which continues its computation exactly as though it had received the final value of the computation. Meanwhile, the scheduler has released the task for potential simultaneous execution on another processor.

### 2.1.1 What Does "Invisible" Mean?

In order to allow an undetermined placeholder to act as an invisible placeholder as long as possible (and, therefore, permit as much parallelism as possible) it is necessary to define a set of rules for deciding when the placeholder can be used and when the object it references must actually be observed. A similar problem arises when creating a normal order interpreter as discussed by Abelson and Sussman[2] (section 4.2.1) and a solution similar to the one outlined there solves the problem very nicely.

The important thing to notice is that each of the Scheme special forms can be analyzed to determine whether it can deal with placeholders or must have the object itself in hand. The Scheme language consists only of these special forms, variable references and procedure calls. Thus, once this analysis is done the properties of all compound (i.e. user-defined) procedures will follow directly. The only remaining areas are thus the primitive procedures of the language and the somewhat subtle problems arising from the fact that Scheme can manipulate its programs as data objects. The results of this analysis are summarized below. Each case is annotated to indicate whether the `touch` is provided by the interpreter (or compiler), or is the responsibility of the code for some primitive procedures.

**Where to Automatically Touch**

1. The value of the predicate in the special forms `if`, `cond`, `and`, `or`, and `case`. This must be handled by the interpreter and/or compiler.

2. The value of the operator in an application. This must be handled by the interpreter and/or compiler.

3. An expression presented for direct evaluation (i.e. in the case dispatch of the interpreter itself a placeholder object will be `touch`ed rather than treated as a self-evaluating constant). This must be handled by the interpreter. This case does not arise in normal practice, although it would be important if the system were changed to allow the program constructors (the "syntaxer" of the MIT Scheme system) to operate in parallel.

4. The value of operands to a primitive procedure when they are required to be of a specific type or set of types, as well as any portions of a data structure inspected by such a primitive, unless (of course) the primitive requires a placeholder. This must be handled by the primitive itself. For example, the primitive operator `+`, which requires numeric arguments, must `touch` each of those arguments. The arguments to the `cons` and `list` operations, by contrast, may be of any type and thus need not be `touch`ed.

5. The value of operands to primitive predicate procedures (except the predicates `future?` and `non-touching-eq?`). For example, the primitive operation `eq?` is a predicate (it is guaranteed to return either `#T` or `#F` regardless of its arguments). This must be handled by the primitive itself.

6. Certain primitives added to deal with placeholders handle the `touch` operation explicitly. For example, the primitive operation `touch` is provided as a way to explicitly touch an object in case it is a placeholder. This must be handled by the primitive itself.

### 2.1.2 Example: A (Contrived) Data Base Example

As an example of the kind of operations possible with placeholders in the language, consider a Scheme program used to maintain a data base of students

currently enrolled in a class. Let us assume, for simplicity, that the data base is a list of records, one record for each student. The records contain some (immutable) data fields such as a name and teacher assignment. We also want to store, for each student, the grade on each quiz, exam, and problem set. As each of these is graded we store the grade into the appropriate field of the record. This requires fields that receive their value *after* the data base is created through a single-assignment operation. For simplicity, we'll consider only one such number, the grade on the final exam.

So far, this is an easy task (if we ignore the usual problems of efficiency, long-term data storage, and atomicity of simultaneous actions). All that is needed is to create a data abstraction for the records including a constructor (`Make-Student-Record`), selectors for each component of the record (`Student.Name`, `Student.Teacher`, `Student.Grade`), and a writer for the single-assignment entry (`Set-Student.Grade!`).

Now imagine that recitation instructors want to routinely examine the data base to see certain information about students assigned to their sections. They should be able to create structures of their own choosing containing parts of the main data records. For example, I might want to create a list of the final exam grades of students in my section. As the main data base is updated after the final exam, I expect my list to reflect the final exam grades of my students. When the data base is created, I plan to do the following: (the procedure `filter` returns a list of items from an input list that match a selection criterion)

```
(define My-Section-Grades
  (map Student.Grade
       (filter All-Students
          (lambda (Student-Record)
            (eq? (Student.Teacher Student-Record) 'JIM)))))
```

Unfortunately, this approach doesn't work if we use the "obvious" implementation of the data abstraction. Since the variable `My-Section-Grades` is created when the data base is initialized, the values for the `Student.Grade` field will have their original (uninitialized) value and this value will not be updated when the final exam grade is entered into the data base. This is just one instance of what Sussman and Abelson refer to as "time of selection" vs "time of construction": we would like `My-Section-Grades` to have its component values computed only when they are *selected* from the list, not when the list is *constructed*.

There are a number of ways, in standard Scheme, of dealing with the problem described here. Perhaps the simplest comes down to making each field that will receive a value after the data base is constructed contain a (single-element) list with the value of the field. Then this list is contained in `My-Section-Grades`, and the list is mutated by the `Set-Student.Grade!` procedure. Unfortunately, this requires that we think of `My-Section-Grades` as a list of lists rather than as a list of numbers. Thus, instead of the straightforward definition

```
(define My-Section-Average
  (/ (apply + My-Section-Grades)
     (length My-Section-Grades)))
```

we are forced to say

```
(define (Extract-Grade-From-Record Record)
  (first Record))

(define My-Section-Average
  (/ (apply +
       (map Extract-Grade-From-Record My-Section-Grades))
     (length My-Section-Grades)))
```

In essence, we are forced into introducing a new data structure for the sole purpose of providing these single-assignment fields. Furthermore, the modularity of our program is changed. It is necessary to provide support for the use of this new data structure for grades rather than using the support already provided for handling numbers. As the complexity of the data base increases by the addition of single-assignment fields or other structures sharing these fields, the number of changes needed to support the late-arriving data also increases.

Once placeholders have been added to the language, however, a more elegant solution becomes possible. The constructor `Make-Student-Record` is modified so that the initial value of the single-assignment field is an undetermined placeholder. The definitions for the writers of the student records must use `Determine!` instead of `Set!` to store the new values. Since the placeholder becomes invisible when it is determined, `My-Section-Grades` acts exactly like a list of numbers and the code we originally wrote for calculating averages works as we had expected.

But, by using the fact that placeholders can be mutated, we can solve an even harder problem. Imagine that, in addition to the single-assignment fields with quiz, exam, and problem set grades we want to maintain an estimated course grade for each student. This number should be updated as the new grades are entered, and thus this field is fully mutable rather than single-assignable. By using the `mutable-determine!` (rather than `determine!`) operation to update this field, we can allow this value to be selected by instructors when the data base is constructed. It, too, will be updated whenever the entry in the main data base is updated.

This example, while somewhat contrived, does represent an important contribution of the placeholder data type to Scheme. The kind of sharing described here coupled with the invisibility of the placeholder object itself provides the underlying support for a variety of capabilities not easily realized in the standard Scheme language. Some of these capabilities are explored in sections 2.3, 2.4, and 2.5.

On the other hand, of course, the more general problem of data base consistency is *not* solved by the use of placeholder. For example, the automatic touching rules listed earlier require that a placeholder be determined before a decision is made based on its value. Yet if the value of that placeholder is changed there is no mechanism to "unwind" decisions based on the previous value. Thus, while the mechanism supporting the placeholder construct is quite powerful and flexible, I do not know how to efficiently generalize it into a truth maintenance system.

## 2.2 Talking to the Garbage Collector

Scheme, in common with other Lisp dialects, provides operations to allocate space from memory. In addition, certain operations inherently require space to represent objects even when these objects are to be used as intermediate results. At the same time, Scheme does not provide any operations for releasing this memory. Instead it relies on a garbage collector (or GC) to recover allocated but unreferenceable memory

There have been three areas where the MultiScheme project has explored garbage collection. The first of these is the construction of two parallel stop-and-copy garbage collector implementations. The earlier of these implementations is described by Courtemanche[12], and was later refined by

Kirk Johnson to produce the second implementation. The second piece of work relates to the garbage collection of useless tasks, and is described in Section 3.4.

The third area, described in this section, is a pair of extensions (weak cons cells and the ability to discover when selected objects are no longer needed) intended to allow the garbage collector to act as a partner in the computation. These extensions allow the garbage collector to provide information derived during its pass over the entire active storage of the system. In order to accomplish this, however, the garbage collector can no longer promise to leave the state of the system apparently unaltered as has traditionally been the case — some aspects of the MultiScheme garbage collector are deliberately visible to users of the system.

The actual garbage collector used by MultiScheme requires that the system suspend all other activities while it is garbage collecting. The changes described here, however, appear to be compatible with the "real-time" garbage collection algorithms derived from the work of Baker and Hewitt[4]. In a system with a real-time garbage collector the GC demons described below would run at the time when semi-space flip occurs.

### 2.2.1 Garbage Collection Demons

Prior to the MultiScheme project, MIT Scheme provided a mechanism known as GC demons for interacting with the garbage collector. Users could provide procedures that would run immediately after each garbage collection and before returning to the work in progress when the garbage collection began. By default, the system supplied two demons to perform operations required by the system itself: a hash demon to support tables indexed by arbitrary objects and a files demon to close files no longer in use.

The hash demon maintains a pair of tables in order to provide a form of "unique ID" service for arbitrary Scheme objects. An object can be inserted into these tables using the primitive object-hash, which returns an integer that serves as a unique ID for that object (a subsequent attempt to insert the same object will return the same unique ID). The tables can be searched using object-unhash which expects as input this unique ID and returns the associated object. The special feature of these tables, requiring support from the garbage collector, is that an object not referenced by any *other* Scheme object will disappear from the hash tables. After each garbage collection the

tables are rebuilt by the hash demon based on only those objects surviving the garbage collection.

The files demon is responsible for detecting the fact that a file is no longer referenceable from within Scheme. It then calls on the operating system to close the file if the user hasn't already done so explicitly. This permits files to be handled rather like memory objects: they are created but need never be explicitly released. The operation is supported by a table of file objects hidden from the GC primitive. This table is rebuilt by the files demon, after each garbage collection, by copying entries seen during the garbage collection or by closing files that were not encountered.

Each of these demons requires the ability to examine the portion of the address space from which objects had been copied (old space), and therefore could not safely be written in Scheme. In the process of developing MultiScheme two new features were added, supported by changes to the garbage collector. The first of these, weak cons cells (discussed in Section 2.2.2), serves as an implementation base for a new hash demon that does not need to reference old space. This new hash demon, therefore, is written entirely in Scheme. Weak cons cells also serve a vital role in MultiScheme's ability to garbage collect useless tasks, as discussed in Section 3.4. Weak cons cells have since been used to perform other vital system services, especially in support of interfacing to compiled code. The weak cons cells, themselves, still require a demon that looks at old space, but it is far less complex and yet supports a wider range of services than the original hash demon.

The second feature, object finalization (discussed in Section 2.2.3), is of somewhat more limited use. It permits the files demon to be written in Scheme. We had also anticipated that it would provide a base for certain user services in MultiScheme. These services, however, have never been implemented.

### 2.2.2 Weak Pointers

The tables supported by the hash demon provide two independent services in one wrapper: interning and weak pointers. The interning service is the one directly visible from its interface functions, `object-hash` and `object-unhash`. Interning provides a (reasonably rapid) method for generating a unique ID for an arbitrary object. Since the unique IDs are integers they provide an ordering on Scheme objects. This permits ordered data structures to be

constructed from arbitrary Scheme objects.

The second service, weak pointers, is what allows objects referenced only by way of the interning service to disappear during a garbage collection. It is the vital support needed for maintaining an association between a property and the set of objects having that property. (Consider, for example, the set of objects that a user has `traced` for debugging purposes.) While it may be necessary to provide operations referring to all objects in the system having the chosen property, it is *not* intended that the objects remain in the system for the sole reason that they have that property. Maintaining this sort of association is sufficiently common that several Scheme dialects, including MIT Scheme, provide a data structure for handling this situation, known as a population. Empty populations can be created, and then objects can be added and removed from them. Operations are provided for applying a function to each remaining element of a population. An object leaves a population either by an explicit user operation or when the garbage collector discovers that no reference to it exists outside of populations.

A particularly important use of populations in MultiScheme is in maintaining the collection of tasks waiting for the result of a given computation. This collection is conceptually a population since there is no need to retain a task just because it is currently waiting for a result to be computed. Representing this collection using a population rather than a traditional (unordered) set data structure is what allows the MultiScheme garbage collector to quench speculative parallelism.

Populations can be built on top of the tables maintained by the hash demon. This, in fact, was the implementation used in MIT Scheme. But nowhere in the description of the population abstraction did it become necessary to rely on the ability to order the objects within the population. Thus, support for this abstraction uses only the weak pointer service of the rehash demon, not its interning service. Indeed, in examining the entire existing MIT Scheme system as well as programs written in Scheme, *all* uses of the unique IDs require only the weak pointer service (although uses for the interning are still anticipated). Maintaining the tables requires both time and space, and yet they could be completely omitted if MIT Scheme had support for weak pointers independent of the interning service.

The solution adopted in MultiScheme is to add a new data type, the weak-cons, to Scheme. A weak-cons is like an ordinary cons cell except that the `car` of a weak-cons is replaced by `'()` when the garbage collector discovers

that its former contents are no longer referenced elsewhere in the system. From the point of view of a programmer, the creation and use of a weak-cons is similar to that of an ordinary cons cell, except that different selectors, constructors, and mutators are used. Weak-cons cells are very efficient to create and reference (unlike the interned unique IDs they replace), although they do impose some overhead at garbage collection time.

The unusual handling of the weak-cons is divided into two parts, both occurring during the garbage collection cycle. First, the actual weak-cons data object is handled specially by the GC primitive. An ordinary cons cell would be copied from old space to new space, and then the contents of its `car` and `cdr` would ultimately be scanned and copied as well, as shown in Figure 2.1. Instead, the contents of a weak cons are deliberately *mis*copied as illustrated in Figure 2.2: the `cdr` is copied as usual, but the `car` is copied and marked as a non-pointer. Thus, when the `car` and `cdr` are encountered later in the garbage collection cycle the `car` is *not* copied but remains with the address of the original contents in old space. During this (mis)copy operation the original weak-cons object in old space is also modified. Its `car` is replaced with a "broken heart" (forwarding pointer) (as is the `car` of an ordinary cons cell) but its `cdr` is also changed. The `cdr` is modified to contain the type code of the old `car` (this was lost when the copy was marked as a non-pointer) and a pointer to the previous weak-cons cell encountered by the garbage collector.

The net result of this skullduggery, pictured in Figure 2.2, is that all of the information necessary to reconstruct the contents of the `car` of a weak-cons is available, and yet it has not actually been updated. Furthermore, there is a chain (in old space) of *all* of the weak conses encountered during the garbage collection. An important aspect of this design (in fact, a driving concern) is that no additional space is required for this handling of the weak-cons cells. The actual copying of a weak-cons is somewhat slower than the copying of a cons cell, but the difference is very small indeed.

The second part of the work is performed by a new gc demon. The demon walks down this chain and updates the new space copies with either a pointer to the relocated contents of the `car` or a `'()`. This decision is based on examining, in old space, the original contents of the `car` of the weak-cons. If it contains a broken heart then the object was copied during the garbage collection and hence is still in the system. If not, then the garbage collector did not encounter the object and hence it has been removed from

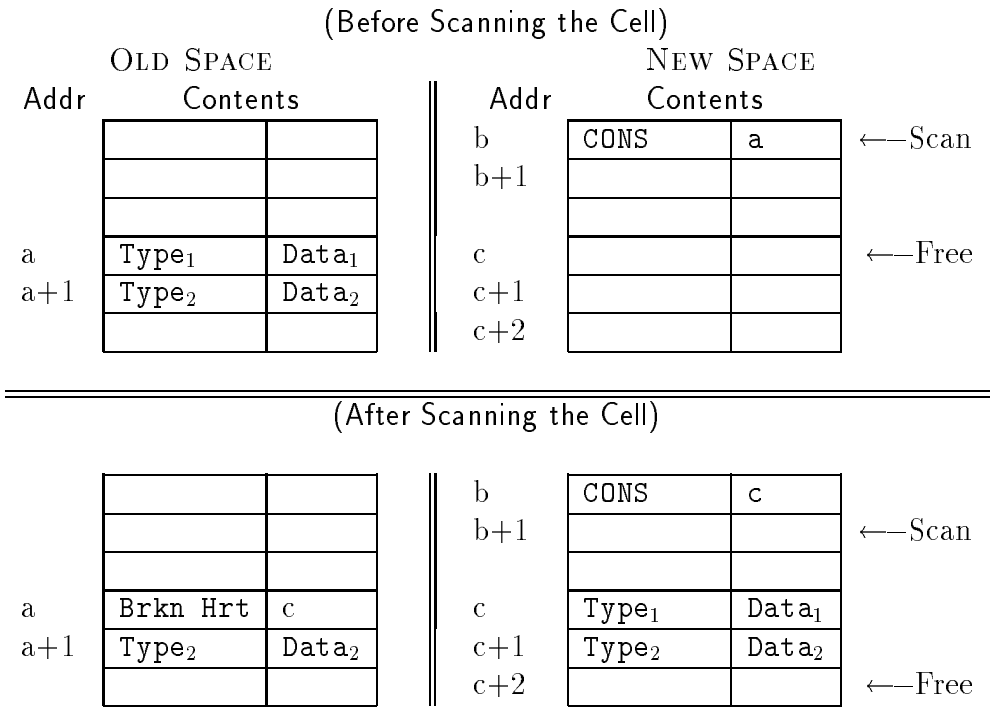


Figure 2.1: Copying a cons cell

(Before Scanning the Cell[1])

| | Old Space | | | New Space | | |
|---|---|---|---|---|---|---|
| Addr | Contents | | Addr | Contents | | |
| | | | b | `WeakCons` | `a` | ←—Scan |
| | | | b+1 | | | |
| | | | | | | |
| a | $\texttt{Type}_1$ | $\texttt{Data}_1$ | c | | | ←—Free |
| a+1 | $\texttt{Type}_2$ | $\texttt{Data}_2$ | c+1 | | | |
| | | | c+2 | | | |

(After Scanning the Cell[2,3])

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | b | `WeakCons` | `c` | |
| | | | b+1 | | | ←—Scan |
| | | | | | | |
| a | `Brkn Hrt` | c | c | *Number* | $\texttt{Data}_1$ | |
| a+1 | $\textit{Type}_1$ | *previous* | c+1 | $\texttt{Type}_2$ | $\texttt{Data}_2$ | |
| | | | c+2 | | | ←—Free |

(After the Demon Runs[3,4])

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | b | `WeakCons` | `c` | |
| | | | b+1 | | | ←—Scan |
| | | | | | | |
| a | `Brkn Hrt` | c | c[4] | $\textit{Type}_1$ | $\texttt{Data}_1$ | |
| a+1 | $\textit{Type}_1$ | *previous* | c+1 | $\texttt{Type}_2$ | $\texttt{Data}_2$ | |
| | | | c+2 | | | ←—Free |

**Notes:**

1. Assume the variable `PreviousWeakCons` contains the value `previous`.
2. The variable `PreviousWeakCons` will now contain the address `a`.
3. *This type style* indicates changes from the handling of a normal cons cell.
4. This cell will contain `'()` if the object no longer exists.

Figure 2.2: Copying a weak-cons

the system. This new GC demon requires time proportional to the number of weak-conses in the system (but no additional space). Hence the garbage collection is somewhat slower, although the number of weak-conses is small enough that this change is also negligible.

Notice that it is critical that this operation be performed after the garbage collector has been run to completion so that all objects remaining in the system will have been moved from old space to new space. It also requires access to the contents of old space and hence cannot be (safely) written in Scheme. Finally, it is important that none of the GC demons executed before this one in any way reference a weak-cons, since all of the weak-conses in the system are in an inconsistent state until this demon has been run.

These simple changes allow MultiScheme to efficiently implement weak pointers. The population abstraction is easily implemented using weak-cons, as are most of the other existing uses of `object-hash` and `object-unhash`. But they have not eliminated the need for the hash demon, since there is no support for the interning services of that demon. However, once the system contains these weak-cons objects, the hash tables (previously special objects hidden from the view of the garbage collector) can be replaced by ordinary tables constructed from regular Scheme data structures and using a weak-cons to hold the keys. The tables still need to be rebuilt after each garbage collection since the hash is based on the address of the objects in memory, but they are no longer deeply intertwined with the workings of the garbage collection algorithm. Furthermore, by separating the two services the size of the hash tables is significantly reduced, since they need only contain objects passed to the actual interning service. As a result, the time spent during each garbage collection for the maintenance of the hash tables is reduced and the overall amount of time spent during garbage collection is reduced.

### 2.2.3 Object Finalization

The file demon supplies a system service of a very specific nature. It detects the disappearance of certain objects (files) in the system and executes a specific procedure for each object (it closes the file). But clearly it would be nice to generalize this to allow for arbitrary objects to be "finalized" when they are no longer needed by the system. This would be useful, for example, as the basis for a resource system designed to allow tasks to allocate selected resources while assuring that they will be released even if the task "dies" in

some way.

Such a resource system is an object-oriented version of the clean-up code provided by a language like Mesa[40], and is modeled on the finishing code of Praxis[58]. In Mesa, clean-up code is attached as an exception handler to a specific *program fragment* to allow actions to be taken if that fragment encounters an error. These actions can include, for example, releasing locks or de-allocating objects created by the program. This behavior can be obtained using existing mechanisms in Scheme. The new mechanism, however, attaches the clean-up code to specific *objects* in the system. It is in some sense the logical inverse of initialization code run when an object is created, being run instead when the object is removed from the system. The notion of some particular *area of program* responsible for clean-up is replaced by the notion of a *self-cleaning object*.

Since the critical issue here is again the "disappearance" of an object from the system, it is clearly related to the garbage collector. And again, a simple extension to the garbage collector provides a good base for this generalization. MultiScheme allows the system to provide one object, called the finalization object, to be used as a root for the primitive GC *after* all of the other roots have been scanned. The primitive GC reports back the location into which this root was copied so that objects surviving the garbage collection normally can be distinguished from those surviving because they were reachable only from this special root. This technique, of course, depends upon the fact that space for copying objects from old space is allocated sequentially in new space during the garbage collection. While this has not actually been implemented on the parallel processor, the extension to that implementation is straightforward. On top of this very low-level mechanism it is easy to provide an extensible mechanism for object finalization.

MultiScheme interleaves the support for finalization and weak-conses in a well-defined order. It first performs all of the normal garbage collection cycle *except* that the `car` of a weak-cons is not copied. Next it copies the finalization object and notes where in memory this copying process began. Finally it runs the user supplied demons, one of which updates the `car` of the weak-conses to reflect the existence of objects retained either in the ordinary process or by the finalization object. As a result, data structures constructed using a weak-cons *do* allow the objects to which the `car` refers to be finalized. This has two ramifications. First, a resource system can be constructed using finalization code as a means of granting serialized access to a resource. The

data structures used by the system must be built using a weak-cons to allow the finalization code to run even though the system can continue to reference the objects (via the weak-conses). Second, it is *not* correct for finalization code to assume that there are no references to the object remaining in the system. Rather, the correct assumption is that the only remaining references are by way of the weak-cons mechanism. This interaction between weak-cons and finalization is useful, but it does require the exercise of care when using both mechanisms in the same system.

This is an easy and efficient way to implement finalization code. It integrates conveniently into the system, allowing certain forms of consistency requirements to be guaranteed. There are, however, some drawbacks that limit its usefulness. They derive from two different aspects of garbage collection. First, there is no way to predict the amount of time before a freed resource will be cleaned up. As a result, at least for resources in frequent demand, this mechanism must be looked on as a form of fail-safe device. It should be used only in cases where it is essential to the system that a final operation be performed to maintain an important consistency requirement even when the ordinary mechanisms for releasing a resource may not be invoked by the user's program. Alternatively if a task discovers that a resource it requires is not available it may choose to wait for an interval sufficient for the resource to be released and then force a garbage collection if the resource is still not available. Thus, the finalization code approach helps solve the problem of a "disappearing resource," but it does not truly support automatic releasing of a critical resource.

The second difficulty is somewhat more subtle. An object is finalized only when all methods of locating it (except through weak-cons cells) have been removed from the system. Thus, if an object is placed in a data structure by some task that "owns" the object, the object will not be finalized until the data structure becomes inaccessible *and* the task loses any means to reference the object directly. This is a two-edged sword. For some system organizations it is perfectly reasonable to assume that an object is in use as long as it can be found by any method whatsoever. For other systems, however, it may be better to view an object as in use only as long as it can be accessed in particular ways — regardless of other ways the garbage collector might be able to come across it. While MultiScheme does not support any general mechanism to help this latter case, at least one common case is easily accommodated. If the access path of interest is always via a known set of

objects (for example, a resource might be considered inaccessible when the task that allocated it disappears), then these objects rather than the resource can be the subject of finalization. Thus, a task might contain a list of all the resources it has allocated and not released. The finalization code for each task would then examine the list of resources acquired by the about-to-disappear task and assure that they are all released.

## 2.3 Example 1: Logic Variables in Lisp

With the recent interest in "logic programming languages," a good deal of attention has been paid to techniques for merging logic programming into Lisp[45]. There are three important notions inherent in these languages from the implementor's point of view: unification, backtracking, and embedding variables in data structures. Unification algorithms can be easily written in Lisp (see, for example, the one in [2], Section 4.5). Similarly, complicated control structures involving backtracking (both temporal and dependency-directed) can be expressed in Scheme using a combination of first-class continuations and procedures. Thus, they pose no difficulties when attempting to embed a logic language within Lisp. The logic variable, however, can be embedded in several data structures in such a way that providing a value to the variable in *any* structure causes that value to appear in *all* of the data structures. This is where a clean embedding in Lisp becomes difficult: a good embedding should permit logic variables to be part of arbitrary data objects and when the variable receives a value all of the data objects should contain this value, for use in whatever manner that value would normally be manipulated. It is precisely here that the addition of placeholders to Scheme plays a role. By representing logic variables using placeholders, provided we do not in some way `touch` them before their values are computed, we need never concern ourselves with whether the data structure was made using normal Scheme techniques or through the use of unification.

The remainder of this section demonstrates an implementation of logic variables using the mechanisms already discussed. The focus here is on the form of sharing provided by logic variables. Thus, the implementation does not attempt to address issues such as backtracking[3] or the deeper questions

[3]To provide a version of the unifier that supports backtracking is not a difficult task, but requires more code than shown here. The major differences are to pass explicit con-

of logic raised by the particular unification algorithm shown here.

Let us represent a logic variable as a placeholder, initially mutably determined to a detectable value. In this case, it is the name of the variable (for us to look at) and a special tag. The placeholder is mutably determined so that when it successfully unifies with an expression we can change the value.

```
(define variable-tag '(LOGIC VARIABLE))

(define (make-variable name)
   (let ((result (make-placeholder)))
      (mutably-determine! result (cons name variable-tag))
      result))

(define (variable? object)
  (and (pair? object)
       (eq? (cdr object) variable-tag)))
```

If we unify a logic variable with an expression, the variable must henceforth take on the value of the expression. Since we are not concerned with backtracking we can use the standard `determine!` operation to accomplish this:

```
(define (unify-variable variable expression)
  (determine! variable expression))
```

To unify two expressions we must handle a number of cases. If the expressions are identical, then they unify to themselves. If either is a variable, then we can use `unify-variable` to combine it with the other. If both expressions are composed of sub-expressions we recursively unify the subexpressions. This particular unification procedure will return `#t` if the two expressions can be successfully unified, or `#f` if they cannot. After successful unification, logic variables in either expression have acquired the values of the corresponding components of the other expression. This is easily implemented:

```
(define (unify exp1 exp2)
  (cond
   ((eq? exp1 exp2) #t)
   ((variable? exp1) (unify-variable exp1 exp2) #t)
   ((variable? exp2) (unify-variable exp2 exp1) #t)
```

tinuations for use when a unification fails, and to use mutable (rather than immutable) `determine!` to provide the values of variables that are successfully unified.

```
((and (pair? exp1)
      (pair? exp2)
      (unify (car exp1) (car exp2)))
 (unify (cdr exp1) (cdr exp2)))
(else #f)))
```

Notice that this unifier is simpler than a more traditional unifier (such as the one in [2]) written in Lisp. Because a determined placeholder is used to represent a variable after it has been unified it is not necessary to consider this case specially. A determined placeholder *is* the value it is unified with, and thus once a variable is unified it no longer matches the `variable?` clauses.

As mentioned at the outset, this program does not address the issues related to unification and backtracking. It lacks two features important in a logic programming system: an occur check, and backtracking. The occur check is easily added, but at a large performance penalty (the occur check is omitted in Prolog for this same reason). Furthermore, if this unifier fails to unify two expressions it may still have determined the values of some of the variables in the expressions. This problem is easily solved in a version written to include a backtracking control structure: unification of variables could be done with a mutable determine and backing out beyond the point where it was unified would restore the previous value.

Using the program shown here, we can create some Scheme variables whose values will be created by the process of unification:

```
; Create some "logic variables"
(define ?a (make-variable 'a))
(define ?b (make-variable 'b))
(define ?c (make-variable 'c))
; Some simple expressions including the variables for unification
(define EXP1 (list 'A ?a  'B ?b))
(define EXP2 (list 'A 10  'B ?c))
; And a "Scheme" expression  not for unification
(define SCHEME-EXPR (list ?a ?b ?c))

; Try it out...

⇒ SCHEME-EXPR ⇝ (#[FUTURE 16] #[FUTURE 84] #[FUTURE 52])
; None of the variables has an (immutable) value and all are different
```

```
⇒ (unify EXP1 EXP2) ⇝ #t
⇒ SCHEME-EXPR ⇝ (10 #[FUTURE 64] #[FUTURE 64])
; ?A has a value, but ?B and ?C do not. ?B and ?C are identical.

⇒ (unify ?b 3) ⇝ #t
⇒ SCHEME-EXPR ⇝ (10 3 3)
⇒ (apply + SCHEME-EXPR) ⇝ 16
⇒ exp1 ⇝ (A 10 B 3)
⇒ exp2 ⇝ (A 10 B 3)
```

By embedding "logic variables" in the expression `SCHEME-EXPR`, this expression changes as the variables receive values. Most importantly, as shown in the expression `(apply + SCHEME-EXPR)`, it is not necessary to know which elements of the data structure received their value by unification and which received them through other mechanisms.

## 2.4 Example 2: Normal Order Evaluation

One of the fundamental results of the $\lambda$-calculus is the Church-Rosser Theorem — a proof that (in a system without side-effects) any order of argument evaluation leads to the same answer *provided* the process terminates. The theorem also shows that one particular order of application, called normal order, will terminate if *any* order will terminate.

The essence of normal order evaluation is "substitute first" on function call — unevaluated arguments are substituted into the body of a function before the body of the function is evaluated. Expressions tend to expand during a substitution phase and then contract in a reduction phase. While there are few theoretical results to support a general assertion, it appears that this expansion during the substitution phase yields an unacceptable *space* requirement in actual implementations. By contrast, Scheme employs applicative order, as do most other programming languages, in which the arguments are evaluated prior to evaluating the body of the function. Attempts to reduce the space complexity of pure normal order systems have made very little headway. For example, even a simple iterative factorial program that executes in constant space in (applicative order) Scheme requires a linear amount of storage in (normal order) Miranda. This is currently a major research area for the functional languages community.

Unfortunately, the use of applicative order for function call leads to infinite loops or errors in some programs that would terminate correctly under normal order. A classic example of this problem is the `if` special form of Scheme. Consider a modified version of Scheme where `true` and `false` are implemented as in Church's work, and predicates (including `=`) return one of these new values:

```
(define true (lambda (X Y) X))          ; Replaces #T
(define false (lambda (X Y) Y))         ; Replaces #F
```

In such a system we might attempt to define our own function, following Church, expecting it to work like the standard `if` special form:

```
(define (our-if predicate consequent alternative)
  (predicate consequent alternative))
```

Finally, we could try it out as follows:

```
(define x 0)
(our-if (= x 0) 0 (/ 10 x))
```

In Scheme, or any other applicative order system, the final line will result in a runtime error. The three arguments to `our-if` will all be evaluated *before* the body of `our-if` is examined, and the third argument `(/ 10 x)` will give a divide by zero error. By contrast, a normal order system would substitute the expressions themselves into the body of the function. Thus this final line would be transformed in successive steps as follows:

```
(our-if (= x 0) 0 (/ 10 x))
((= x 0) 0 (/ 10 x))
((lambda (X Y) X) 0 (/ 10 x))
0
```

Thus, the programming community seems to be faced with an unfortunate trade-off. Normal order systems are convenient because they get stuck in computational blind alleys only when there is no other choice. Applicative order systems are space efficient because they produce data objects at an earlier stage of the computation. A variety of avenues have been explored for combining the best parts of both kinds of systems:

- Strictness Analysis, a static technique for discovering which arguments to a procedure can be passed by value (i.e. using applicative order) without running into the attendant difficulties.

- Syntactically marking entire procedures as "by value" or "by name," giving programmer control over the trade-off on a procedure by procedure basis.
- Marking individual procedure parameters as "by value" or "by name."
- Allowing individual calls to procedures to pass `delay`ed arguments. This is the approach permitted in Scheme; unfortunately, the procedure must be written so that it anticipates which arguments are (or may be) delayed.

Placeholders provide a time-efficient vehicle for solving the difficulty mentioned in this last approach. With the addition of a placeholder object the need to explicitly `force` an argument is replaced with the automatic `touch` supplied for placeholders. Thus procedures can be called with a "normal-order" parameter by merely specifying a placeholder for that parameter. The code for the procedure is unchanged, and can thus handle *any set* of its parameters being passed using either method.

In order to make this use of placeholders convenient, MultiScheme provides a standard scheduling policy known as the `delay-policy` (see Section 4.3.1). This policy causes the placeholder to contain a procedure (historically known as a `thunk`) that will be called when the value of the parameter is required. The scheduler responds to a `touch` of this placeholder by calling the thunk and determining the value of the placeholder. Thus, we could rewrite our earlier trial program as:

```
(our-if (= x 0) 0 (future (/ 10 x) delay-policy))
```

and it will now work without any other changes, producing the desired `0` result.

Thus, in much the same way that the future mechanism allows manual annotation of parallelism it also allows manual annotation of normal order parameter passing[4]. One particularly interesting use of this ability is in relation to the `letrec` construct used to define a set of mutually recursive identifiers. Scheme provides a special form for this purpose whose syntax is

[4]The mechanism described here is *not* sufficient to provide full "call by name" parameters. Call by name would allow the body of the procedure to mutate the formal parameter and have this side-effect be transmitted back to the argument. The mechanism is more properly a "call by need" (memoized) optimization of normal order evaluation.

the same as that of the `let` special form. The standard Scheme reference manual[48], however, states:

> One restriction on `letrec` is very important: it must be possible to evaluate each [initial value expression] without referring to the value of any [of the introduced variables]. If this restriction is violated, then the effect is undefined ... The restriction is necessary because Scheme passes arguments by value rather than by name. In the most common uses of `letrec`, all the [initial value expressions] are `lambda` expressions and the restriction is satisfied automatically.

Thus, the `letrec` special form in Scheme is really not intended for use in creating recursive data structures or any other elaborate mutually recursive definition. By using the placeholder implementation of delayed objects, however, this restriction can be eliminated. In fact a simple macro can be used to translate a `letrec` expression into an equivalent `let` expression. The new expression will converge to a value whenever the original `letrec` will converge. Consider the circular list created by

```
(letrec ((ones (cons 1 ones)))
  ones)                    ; body
```

This leads to an error in Scheme, since the variable `ones` is introduced by this `letrec` form and also used in the initial value form. This can be mechanically translated, however, into the following expression:

```
(let ((ones))              ; Create variable, no value
  (set! ones (future (cons 1 ones) delay-policy))
                           ; "Futurized delay" of
                           ; initial value expression
  ones)                    ; body
```

This correctly implements the version of the original `letrec` expression. A similar syntactic transformation can be used for any `letrec` expression, effectively removing the language restriction quoted earlier. Notice that this same transformation would *not* work using the `delay` special form of Scheme; my method depends on the automatic `touch` provided for placeholders (but not for ordinary delayed objects).

## 2.5 Example 3: Stream Processing

One very interesting technique for structuring (primarily functional) systems, called stream processing, involves the use of a data structure (a stream) similar to a list but with the `cdr` of each cell containing a delayed object instead of the actual object. Thus as one recursively walks down a stream the first item (its `car`) is always available but the remaining items must be `force`d in order to access their value. Since only the items actually needed during a computation are `force`d during that computation, a stream can represent an infinite set of values while occupying a finite amount of space. If the stream "remembers" the values of items already referenced so that they needn't be recomputed (called memoization), then the data structure is efficient in both time and space. Streams are easily implemented in Scheme, and lead to a very convenient abstraction.

An excellent introduction to stream processing in Scheme is found in Abelson and Sussman[2], Section 3.4. They demonstrate how streams are used in a variety of examples. Two of these examples, in particular, highlight problems with the stream implementation as normally supported in Scheme: the need for uniformly delaying arguments to procedures (in the `integrate` procedure), and the difficulty of producing a fair merge operation. By using placeholders rather than the standard implementation of delayed objects MultiScheme provides solutions to both of these problems.

### 2.5.1 Uniformly Delayed Arguments

The first difficulty arises out of the need for mutually recursive streams in order to solve certain simple problems. The example explored in Abelson and Sussman is a simple (numerical) integrator used to solve differential equations. The development of that example is repeated here (in condensed form) to illustrate the general problem.

It is easy enough to write a procedure that receives as input a stream of incoming sample values (`integrand`), the value of the integral at time $t = 0$ (`initial-value`), and the time interval between samples (`dt`) and produces a stream of the integrated values:

```
(define (integral integrand initial-value dt)
  (define int
    (cons-stream initial-value
```

```
                (add-streams (scale-stream dt integrand)
                             int)))
  int)
```

Recall that the second argument to `cons-stream` is delayed rather than evaluated, so the fact that it references the value of `int` as part of the definition of `int` does not lead to a problem. There is a built-in unit of delay in this branch of the stream.

We might try to use this procedure as part of a program to solve a first-order differential equation. Given a procedure that calculates the value of $f(y)$, we want to find the stream of values $y(t)$ such that $\frac{dy}{dt} = f(y)$. This would be straightforward if Scheme supported mutually recursive data structure definitions (for example, if Scheme were a normal order language). In that case we could write:

```
(define (solve f initial-y-value dt)
  (define y (integral dy initial-y-value dt))
  (define dy (map-stream f y))
  y)
```

Unfortunately, in order to calculate the value of `y` (a stream) we must supply the value of `dy` (also a stream) to the integral procedure. But in order to calculate `dy` we need to know the value of `y`. One way out of this dilemma is to explicitly introduce a `delay` operation to "cut the loop" by changing the integrand argument from call by value to call by name. In MIT Scheme, however, this requires that the body of integral have a corresponding `force` added. This, in turn, requires that *all* calls to integral have a delayed argument for the integrand.

The problem, then, is that the modularity of the program is disrupted. Because one call to the integral procedure needed to be delayed, the external interface to the procedure was altered, and all calls need to be located and changed correspondingly. This is the same problem explored in Section 2.4 appearing in a different guise. It is not surprising, then, that the same solution can be used. In MultiScheme it is not necessary to explicitly force a delayed object if the object is implemented using a placeholder. Forces (touches) are supplied automatically when the object is referenced. Thus in MultiScheme we need only insert delays in those calls arising from mutually recursive declarations.

### 2.5.2 AMB and Fair Merge

In a purely functional stream processing system objects are modeled by providing functions used to derive the new state of an object from information in the previous state. Furthermore, it is assumed that these modeling functions can be executed *at any time* and *in any order*. This proves to be an extremely elegant formulation for many self-contained systems. Unfortunately, interactions with objects external to the system may have a very different character. It is not possible to provide a function describing the state of, for example, a computer terminal as it will appear five minutes from now. We would prefer, instead, to model the terminal as a device that issues events. Thus we would like to make a stream corresponding (say) to the sequence of characters typed on a particular terminal line, along with a tag specifying which terminal they came from (so we can support multiple terminals). This is simple enough to implement in Scheme, although there is at least one oddity here. What is the first item in this stream (recall that the first item in a stream is instantly available and only subsequent items may require a delay while they are computed)?

```
(define (make-terminal-stream terminal)
  (define (loop)
    (cons-stream
     (cons terminal (read terminal)) ; Tag + Data
     (loop)))      ; More elements
  (cons-stream 'dummy-first-element (loop)))
```

Now the stream behaves normally, although attempts to read beyond the first item in the stream will cause the system to delay until characters are typed on the keyboard — a potentially unbounded amount of time.

But building an event based system, a very productive technique in many operating systems, is not this simple. Henderson's paper[29] pursues this same example to the next step. What if our goal is to write a program in order to listen to two terminals and process the key strokes from them *in the order in which they arrive* from the terminals? This, of course, is precisely where event driven systems are at their best. If we have more than one source of events (say we have two terminals with a stream for each) we now must specify *in which order* we will examine the streams. And here is the fundamental problem. Our program is no longer truly free to apply its state transition functions in an order of its own choosing. Whichever stream

the program chooses to examine first, it is possible that the corresponding terminal will not have any data available. Yet the framework described so far has no way to examine a stream without waiting for the data to arrive. So regardless of which stream the program examines first, there is a possibility that it will delay when data was available on the other stream.

In most event based systems there is a procedure allowing a program to say "please delay me until *any one* of the following events occurs, then tell me which one it was". This is precisely the ability we need to solve this problem. There are a number of ways to formalize and solve the problem, all of them requiring the addition of a relation (rather than a function) to the system. Two such relations are McCarthy's amb operator[39] and fair merge. For my purposes, the following informal descriptions of these relations will suffice:

**Amb** takes two inputs. If only one of its inputs is undefined, then the value is the other input. If both inputs are undefined, then the value is undefined. If both inputs are defined, the value is either of the two (I do not require `amb` to be "fair").

**Fair Merge** takes two streams and returns a stream containing all of the elements from its inputs in the same relative order in which they occurred in the inputs (hence it is a merge). The output stream has no predefined order of interleaving of the two streams, but rather includes items from each stream in the order in which their values become available. In cases where values are available from both streams at the same time, fair merge does not give preference to either stream.

MultiScheme's placeholders, along with specific help from the scheduler, provide an event-driven mechanism for supporting the creation of these procedures. The basic support comes from the `disjoin` operation of the scheduler whose implementation is shown in Figure 4.9 on page 106. `Disjoin` takes an arbitrary number of arguments and returns a placeholder whose value, ultimately, is the first of the arguments to be given a value. In order to be useful, of course, `disjoin` is ordinarily called with undetermined placeholders for all of its arguments. `Disjoin` is driven by the mechanism that gives values to placeholders rather than by any kind of polling or busy-waiting.

Disjoin is thus a perfectly adequate implementation of `amb` as described above[5]. Fair merge is easily written using `disjoin`:

```
(define (fair-merge s1 s2)
  (touch (disjoin (car s1) (car s2)))   ; (1)
  (if (undetermined? (car s2))          ; (2)
      (cons-stream (car s1) (fair-merge s2 (cdr s1)))
      (cons-stream (car s2) (fair-merge (cdr s2) s1))))
```

There are three subtle (perhaps) points to be understood about this small program. The line marked (1) in the program contains a `touch` to guarantee that the first element of the output stream is available before proceeding. Fair merge returns a stream, so this first element must be known before it can return a value. But disjoin does not guarantee that its value has actually been computed. The `touch` will suspend this computation until one of the two values becomes available.

The second marked line is a rather inelegant but effective way to decide which element will be placed in the output stream. Having guaranteed that *at least* one of the initial elements in the input streams is now known to have a value, it tests *a particular* element to see if that is the one whose value is known. A more powerful variant of `disjoin` that returns the identity of the argument which has acquired a value, rather than the value itself, can be easily built on top of this one.

Finally, notice that the two recursive calls to `fair-merge` reverse the order of the two streams being merged. This guarantees that fair merge will alternate between the streams in case both streams have values immediately available. In an $n$-way fair merge this would be replaced by a cyclic permutation of the streams.

## 2.6 Summary

By adding two new features (placeholders, described in Section 2.1, and a visible interface to the garbage collector, described in Section 2.2) to Scheme we

[5]Some authors use slightly different definitions for `amb`, implying fairness or certain formal mathematical properties. It is because MultiScheme's `disjoin` does not necessarily support these assumptions that a new name was chosen.

create a new language (extended sequential Scheme) which has considerably more expressive power than the original.

Because they introduce a new method of sharing, placeholders provide a means of easily embedding logic variables into LISP (Section 2.3). In addition they provide a convenient way to embed normal-order evaluation into an otherwise applicative-order language (Section 2.4). This allows recursive data structures to be conveniently defined, and provides a good deal of the power of a normal-order language without the problems of either strictness analysis or loss of control over space utilization.

This power can be conveniently exploited by using placeholders in the implementation of streams of data objects (Section 2.5). Doing so solves two often troublesome problems:

- Because a computation is automatically delayed when a value it requires is not available, it is not necessary to uniformly impose this `delay` on all uses of a procedure just because it is required for one use.

- As will be shown in Section 4.3.2, placeholders can be created to represent "the first value to converge." This provides an elegant base for implementing McCarthy's `amb` operator and the `fair-merge` of stream processing.

# Chapter 3

# Parallel Processing Extensions

This chapter describes the work that went into converting the extended sequential Scheme language of Chapter 2 into a systems programming language for a parallel processor. Sections 3.1 and 3.1.1 motivate and describe the `global-interrupt` operation and synchronizer objects used to coordinate the actions of processors (as opposed to the placeholders that coordinate the actions of tasks). Sections 3.1.3 and 3.1.4 then show additional uses for these extensions. A few additional procedures, of much less interest, are described for completeness in Section 3.2.

Section 3.3 discusses the design decisions related to providing storage on a per-task basis. Dynamic and fluid binding are introduced and compared in Sections 3.3.2 and 3.3.3, respectively. The mechanism used by MultiScheme to implement fluid variables in a parallel-processing system is described in Section 3.3.4. Finally, Section 3.3.5 describes several mechanisms available without any further extensions that can be used to provide storage intended to be completely private to a task.

Finally, Section 3.4 introduces MultiScheme's task object. The important concept of a task goal, which enables the garbage collector to detect tasks that are no longer needed, is described in Section 3.5. Again, no language extension is needed to support tasks, but the separation of tasks and placeholders is explored in MultiScheme for the first time.

## 3.1 Processor Coordination

MultiScheme provides two methods (global interrupts and synchronizers) for coordinating the activities of processors. Unlike placeholders, which coordinate the activity of *tasks* (logical processes generated by running programs), these two operations deal with the physical processing units of the hardware. As such, they are more often used by the MultiScheme system itself than by application programs. The two operations, while not necessarily novel, have the virtues of simplicity and compatibility. They also serve as a base for higher-level constructs useful in parts of the system not related to garbage collection (see the examples of Section 3.1.3 and 3.1.4).

### 3.1.1 Starting Garbage Collection

MultiScheme was built by first extending the Scheme language as discussed in Chapter 2. This system was tested on a sequential computer by using time-slicing to simulate a parallel processor. The next stage of development involved moving this implementation to an actual parallel processor. One of the early problems encountered in this move was modifying the mechanism used to initiate a garbage collection in order to make it compatible with a parallel processor. In (serial) MIT Scheme, garbage collection is initiated in three phases, using a system modeled after a hardware priority interrupt mechanism:

**Interrupt Request**
: During some operation the processor notices that it is low on memory and sets a bit requesting a garbage collection interrupt.

**Interrupt Detect**
: The interpreter and compiled code periodically poll the interrupt bits. A pending interrupt is serviced if no higher level interrupt is pending or in progress.

**Interrupt Service**
: Before executing the next instruction from the running program, the machine calls a procedure supplied by the Scheme runtime system (the interrupt handler), in this case corresponding to the garbage collection interrupt.

In moving to a parallel-processor hardware base there was no need to modify the basic interrupt mechanism but some of the details were modified. Because all of the processors share a common address space for the heap it is essential that they cease computing[1] before the garbage collector begins relocating objects.

The system must, therefore, support some mechanism for forcing all of the processors to synchronize. The ability to initiate such global synchronization from software (in addition to any hardware initiation that might be available) is essential to several system services (see the examples of Sections 3.1.3 and 3.1.4).

### 3.1.2 Language Extensions for Coordination

Only three modifications to extended sequential Scheme are used in MultiScheme to provide coordination among the processors: the interrupt levels intersperse global and local interrupts, the primitive procedure `global-interrupt` allows any processor to interrupt the others, and synchronization objects permit all processors to proceed in unison.

The new primitive operation `global-interrupt` is a software interface to the global interrupt facility:

```
(global-interrupt interrupt-priority-level
                  interrupt-handler
                  all-clear?)
```

The `interrupt-handler` is a procedure that is executed by all the *other* processors once the interrupt is detected by that processor. Initiating a global interrupt is a system-wide resource, and the `global-interrupt` mechanism forces serialized access to the resource. A processor receives permission to initiate a global interrupt only when no interrupt (local or global) of a higher priority is pending. At that time, it calls the `all-clear?` procedure to determine whether or not the interrupt should actually be initiated. This test is used, for example, to guarantee that a garbage collection global interrupt is issued exactly once even though the need for it may be detected independently by multiple processors. The value returned by a call to `global-interrupt`

[1] MultiScheme uses a stop-and-copy garbage collection algorithm. The subsequent discussion, with only slight modification, applies equally well to initiating the space flip in a real-time copying garbage collection algorithm.

is the value returned by `all-clear?` so the processor issuing the interrupt can determine whether or not the interrupt was actually generated.

The `global-interrupt` primitive returns control to the caller only after the interrupt is initiated or the `all-clear?` procedure indicates that no interrupt should take place. It guarantees that all of the processors will stop their ordinary work as soon as they poll their own interrupt bits, an event that the interpreter and compiler force to occur fairly often. This alone, however, is not sufficient to solve the problem of starting a garbage collection. Before *any* processor can begin the actual garbage collection operation, *all* processors must be entering the garbage collection operation: the global interrupt mechanism provides a way of *initiating* an action, but does not provide *synchronization.*

Instead, MultiScheme provides a pair of procedures for this purpose: `make-synchronizer` and `await-synchrony`. To synchronize all of the processors, one processor makes a synchronizer object and then forces all of the other processors to call `await-synchrony` with the synchronizer as argument. When all processors are waiting for synchrony on the same synchronizer object they all return from the call to `await-synchrony`. Typically, one processor makes one or more synchronizers and then uses `global-interrupt` to force the other processors to begin waiting on them.

While separating these two operations can cause deadlocks if they are used improperly, the operations do serve two distinct purposes. The examples of the next two sections show how the separate operations can be used to provide higher-level operations that are not as easily provided if the low-level operations are bundled together.

### 3.1.3 `Pause-Everything`

Clamen[11] describes an early investigation into debugging tools for dealing with the parallelism of MultiScheme programs. He identified a variety of situations requiring a program to temporarily stop all other work on the system, perform some action, and then allow the work to proceed. In order to provide this ability, he implemented the original `pause-everything` procedure. A new implementation extending Clamen's original version is depicted in Figure 3.1, simplified for purposes of explanation. It provides a message-passing interface to an object representing the tasks that were available for execution at the time of the call to `pause-everything`. This interface is de-

scribed in Section 4.6 and the support routines referenced in the procedure are discussed in detail in Chapter 4.

`Pause-everything` uses both the global interrupt mechanism and the synchronizers. A global interrupt is necessary to force the other processors to save their state and become idle. The synchronizers are used to divide the work into two phases.

The first phase is initiated by the call to `global-interrupt` and ends when all processors have arrived at the first synchronization point. The interrupt guarantees that all processors except the one that called `pause-everything` will begin executing the code in `interrupt-handler`. Thus the other processors save away the state of the task they are executing and place it on the queue of work to be performed. They then wait for all processors to execute `(await-synchrony drain-synch)`. When all the processors arrive at this point, the tasks available for execution (including the ones that were formerly executing) have been saved on the work queue.

The processors proceed past the synchronization point, beginning the second phase. All but the initiating processor will arrive immediately at the second rendezvous point, the call to `(await-synchrony proceed-sync)`. The initiating task, however, first saves away the contents of the work queue in the variable `queue` and empties the queue. All the processors again rendezvous, ending the second phase.

The initiating task makes the message accepting object based on the value of `queue` (by calling the procedure `make-returned-object` shown in Figure 4.12 on page 115). This becomes the value of the original call to `pause-everything`. The other processors, however, have now finished the procedure which is the argument to `saving-state`. But `saving-state` does not return to the procedure that called it. Instead it tries to get work from the (now empty) work queue. So at this point, the initiating task is still running on the initiating processor. All the other processors have relinquished the task they were executing and are waiting for more work. The system has "paused."

An interesting detail has been deliberately omitted in this description. What should happen if, while other work is suspended, the running task `touch`es the placeholder associated with one of the suspended tasks? While there is no obviously correct answer, the actual MultiScheme system marks these suspended tasks `paused`. Touching a placeholder that is marked `paused` is handled by the scheduler of Chapter 4 in exactly the same way the scheduler

```
(define (pause-everything)
  (let ((drain-synch (make-synchronizer))
        (proceed-synch (make-synchronizer)))
    (define (interrupt-handler)
      (saving-state                          ; (1)
       (lambda ()
         (await-synchrony drain-synch)
         (await-synchrony proceed-synch))))
    (global-interrupt high-priority
      interrupt-handler (lambda () #T))
    (await-synchrony drain-synch)
    (let ((queue (drain-work-queue)))        ; (2)
      (await-synchrony proceed-synch)
      (make-returned-object queue))))        ; (3)
```

**Notes:**

1. `Saving-State` stores away the state of the task currently executing on this processor and places it on the work queue. It then calls the procedure which is its only argument. `Saving-State` never returns to its caller: it looks for work from the queue when the argument procedure is finished. See the discussion in Section 4.2.3 and the code in Figure 4.8 on page 102.
2. `Drain-work-queue` empties the queue of tasks awaiting processors and returns a list of the tasks removed. See the description of `drain-work-queue` in Section 4.1.3.
3. `Make-returned-object` creates the message-accepting object that is the result of a call to `pause-everything`. See the discussion of `make-returned-object` in Section 4.6 and the code in Figure 4.12 on page 115.

Figure 3.1: Simplified Code for `Pause-Everything`

handles touching a placeholder marked delayed: the associated task will be reactivated. One of the advantages of writing the scheduler in MultiScheme is that such decisions can be easily changed. For example, it is quite easy to add a "scheduler hook" to allow users to specify their own way of handling this situation.

### 3.1.4 Within-Task

The final example of processor coordination is a critical part of the user interface, allowing the user to interact with previously started tasks. The primary use of this facility occurs in the top level interaction between a user and the MultiScheme system. In MIT Scheme, a user can at any time interrupt execution of a program and create an interaction environment within the state of the program at the time the interrupt was serviced (by issuing a "breakpoint interrupt"). A user can also interrupt the program and force it to `throw` back to an earlier interaction environment, effectively aborting the current computation

The direct extension of this into MultiScheme would allow a user to interrupt the system and interact with any one of the tasks in the system[2] at the time of the interrupt. Because a task has state information visible to the programmer (see Section 3.3) it is important that the correct task actually interact with the user. As a result, there must be some way to force a selected task to call the procedure which implements the interaction environment or invoke a continuation that aborts the current computation.

`Within-task` is designed to facilitate this and other similar operations. Some of Clamen's debugging tools, for example, rely on `within-task` in order to report the progress of a program back to the user. A simplified version of this procedure is shown in Figure 3.2. It expects two arguments, a task and a thunk, and forces the task to execute the thunk before continuing with whatever processing it is currently doing. The operation of this procedure is intertwined with the scheduler (so the details are deferred to Chapter 4) but the overview demonstrates a different use of the global interrupt mechanism.

`Within-task` works by testing whether the task is currently executing on one of the processors. If so, it marks the task indicating that it must

[2]The user interface normally used for MultiScheme does not allow the user to select a particular task for interaction. Rather, it chooses randomly from the currently executing tasks.

```
(define (within-task task thunk)
  ...
  (with-task-locked task
    (lambda (task-still-runnable?)
      (if (eq? (task.status task) 'RUNNING)
          (begin
            (set-task.status! task 'WITHIN-TASK)
            (set-task.code! task thunk)
            (global-interrupt high-priority
             (lambda () (if (eq? (current-task) task) (reschedule)))
             (lambda () #T)))
          ...))))
```

**Notes:** Description of the task data structure as well as the utility routines `with-task-locked`, `current-task`, and `reschedule` are provided in Chapter 4.

Figure 3.2: Simplified Code for `Within-Task`

be forced to execute the specified code and then initiates an interrupt on all processors, forcing the one running the chosen task to reschedule the task for later execution. The procedure `reschedule`, described in Section 4.4, stores the state of the task in such a way that the specified code is executed before the task resumes its current computation.

Unlike the earlier uses of global interrupts, this one does not need any synchronization of the processors. The purpose of the interrupt is merely to get the attention of a particular processor (without knowing in advance which one). All of the processors are briefly interrupted from their duties to process the interrupt, but they can resume immediately and independently.

## 3.2 Minor Extensions

There are minor additional extensions needed to convert extended sequential Scheme into MultiScheme:

- Programs can ask the number of Scheme interpreters in the system (using `N-Interpreters`) and subdivide work accordingly. They can also ask which interpreter they are currently running on (using `My-Interpreter-Number`). MultiScheme also provides a `My-Processor-Number` for use in reporting hardware problems[3].

- MultiScheme provides an abstraction of an underlying task distribution mechanism, described in Chapter 4 along with the details of implementing the MultiScheme scheduler. User programs are not expected to deal with this set of primitives unless they modify the scheduler to provide specialized scheduling policies.

- MIT Scheme's mutation operations (`set!`, `set-car!`, etc.) were already defined to return the previous value stored in the cell they modify. In MultiScheme these operations are performed atomically by the interpreter. The compiler may detect the fact that the value is not used and convert them into pure write operations instead of atomic swaps.

- MultiScheme supports operations derived from Multilisp[24] for performing atomic conditional modify operations. In MultiScheme, these operations are `set-car-if-eq?!`, etc.

## 3.3 Per-Task Storage

In a system that supports concurrently executing tasks there is some amount of information which constitutes the state of the task from the point of view of the underlying system. But in addition to this system-imposed state, the program structure itself may require storage on a per-task basis (see, for example, the Uniform System example of Section 5.4.3). Providing this storage is closely related to issues of variable naming, and MultiScheme has extended the fluid variables of MIT Scheme to provide per-task storage.

Section 3.3.1 describes Scheme's mechanism for providing localized storage in a sequential program through the use of lexically scoped variables —

[3] "Interpreter numbers" are assigned sequentially from 0 through `N-Interpreters`$-1$. "Processor numbers" are assigned by the underlying hardware and are not necessarily sequential.

a technique familiar as "block structure" in other languages. One frequently-encountered problem with this structure, however, is the difficulty of allowing a single name to denote different values based on the control flow of the program, the core of the difficulty encountered with per-task variables in MultiScheme. Section 3.3.2 describes the traditional Lisp dynamic binding approach to solving this problem, while Section 3.3.3 describes the fluid variables used in Scheme. Section 3.3.4 describes the implementation changes required to provide fluid variables in a parallel system, and Section 3.3.5 concludes by pointing out a set of additional options that might be pursued if the block structure of these per-task variables is not an important concern.

### 3.3.1 Packaging and Lexical Scoping

The problem of controlling a program's name space was one of the important motivations in the original design of Scheme. This led Scheme to depart from the dynamic binding of variables, which had been one of the primary underpinnings of Lisp for over a decade, in favor of lexical scoping of variable names. One of the hallmarks of MIT Scheme has been its insistence on using *only* lexical scoping for variable reference. In this regard it is like the Algol family of languages. Because Scheme also includes operations with side-effects, programs written in Scheme are analyzed using an environment model in which procedures contain both the code for executing the procedure and the environment in which free variables are to be resolved[4].

One well-known and desirable property of lexical scoping stems from the fact that the scope of a variable is textually localized. Thus a compiler for a lexically scoped language can make deductions about the ways in which variables are referenced even when presented with small parts of the system. Unlike a compiler for a dynamically scoped language, it does not need to see an entire program to know that a certain variable's value never changes, or that it is unreferenced. The compiler need only see the code that comprises the scope of the variable. Thus separate compilation of modules can be performed with optimization of variable references in a lexically scoped language without imposing unnatural rules about referencing free variables, or introducing unwarranted assumptions about the behavior of components

[4]The compiler is often able to deduce that some of this information is not needed at run time and is free to choose more efficient representations based on these deductions.

of the program external to the compilation unit.

This same ability to subdivide the code aids programmers and designers in building and evolving large systems. The scoping, or block structure, provides a simple and effective mechanism for isolating independent parts of a system from one another. At the same time, it provides an easily used and easily detected form of communication through the use of free variables. Combining these in different ways provides a great deal of power and flexibility. One common pattern found in MIT Scheme comes from the ability to treat environments as first-class objects. With this ability it becomes possible to create a package of related procedures, similar in spirit to a Clu cluster[37].

A Scheme package is a single environment frame which contains a number of variables that are global to the package. Most packages include in these "package variables" a number of procedures that perform activities using the variables to guide their behavior. Some of these procedures, containing the package as the environment to use for resolving free variable references, are also exported from the package to serve as interface procedures with the rest of the Scheme system. Figure 3.3 shows a typical piece of MIT Scheme code to construct a hypothetical package used for printing numbers. This has one explicitly declared package variable (`radix`) in addition to all three of the procedures that are created using `define`. Two of these three procedures are exported to the external environment (using the `access` special form of MIT Scheme), and one of them is "renamed" in the process (i.e. it is referenced by one name within the package and by a different name from outside the package).

### 3.3.2 Dynamic Binding

The package introduced in Section 3.3.1 also demonstrates a common difficulty with this organization. Suppose we discover a frequent need to execute an existing program that prints some numbers using `print-number`, but with output in octal or hexadecimal rather than decimal. Our first inclination might be to write a new interface procedure as part of the package:

```
(define number-package
  (let ((radix 10))
    (define (number->list-of-digits number)
      ...)
    (define (print-number number stream)
      (for-each
       (lambda (digit) (display digit stream))
       (number->list-of-digits number)))
    (define (change-radix! new-radix)
      (let ((old-radix radix))
        (set! radix new-radix)
        old-radix))                    ; Return old radix
    (the-environment)))                ; (1)

;; Now export the interface procedures
(define print-number
  (access print-number number-package)) ; (2)
(define set-print-radix!
  (access change-radix! number-package))
```

**Notes:**

1. `The-Environment` supplies the current environment frame as an ordinary Scheme object.
2. `Access` performs a (lexical) variable lookup in a specified environment frame.

Figure 3.3: A Simple Package

```
(define (with-new-radix procedure new-radix)
  (let ((old-radix (set-print-radix! new-radix)))
    (let ((result (procedure)))
      (set-print-radix! old-radix)
      result)))
```

We change the radix and remember the old radix. Next we call the procedure, and finally set the radix back to its old value. This works unless somehow the procedure we are calling manages to return back past the call to `with-new-radix` without executing the code to set the radix back. This can only happen if during the execution of that procedure a continuation made earlier in the computation is used. This might at first appear to be a fairly unlikely occurrence; but it is precisely what happens if the user interrupts the computation using `within-task` as described in Section 3.1.4.

By allowing first-class continuations into the language, we have obliged MIT Scheme to provide some mechanism for handling this kind of multi-level return. Many languages provide an "exception handling mechanism" to handle this situation, and MIT Scheme provides a very general mechanism that serves this purpose, described in Appendix D. The problem here, however, is a simple one that illustrates a problem with lexical scoping compared to dynamic binding. In a dynamic binding discipline the values of variables are located with respect to the execution stack at the time of the reference. So, if we used dynamic binding for the value of `radix` we could safely use the following piece of code:

```
(define (with-new-radix procedure radix)
  (procedure))
```

Introducing "just a little" dynamic binding into Scheme requires the careful examination and resolution of a number of potential problems between the interactive environment and tools with a more static view of programs, such as a compiler. The goal of the MIT Scheme interactive environment is to support the very simple model of a user positioning an interaction loop inside an environment corresponding to a specific procedure invocation. This allows users to enter an arbitrary Scheme expression and find its value at that position within the program. But in order to do this, there must be some way of knowing which free variables in the expression are to be referenced using dynamic binding (and hence use the call stack) as opposed to those that are to be referenced lexically (using the chain of frames of definition).

In general this information must also be known at compilation time[5], and might be introduced in a variety of ways.

1. The source text could require explicit declarations stating which way to treat variables for which no binding is found at compile time. Unfortunately, at run time there is no information available that specifies what parts of a program were compiled as a single unit, or which declarations were in effect for what variables. As a result it is not possible to determine which references entered by the user were to variables determined by the compiler to be dynamic as opposed to lexical. Of course, the compiler could leave around such information, but this places an otherwise unnecessary burden on the compiler and runtime system.

2. A default assumption can be used, as in ZetaLisp. In this case referencing any variable for which no binding is provided by the compilation unit returns the dynamic binding of the variable[6]. But this has precisely the same problems as the earlier solution, since this information is also not available at run time.

3. A new special form could be created for introducing dynamic bindings. This is an interesting compromise and is the basis for the actual choice made in MIT Scheme and extended in MultiScheme. This option is discussed in Section 3.3.3, below.

4. Special forms could be introduced for both referencing and introducing a dynamic binding of a variable. This option is so burdensome that it would make the use of dynamic variables virtually non-existent. It also opens up the possibility of program bugs arising from inadvertently referencing the dynamic binding of a variable when the lexical value is desired, and *vice versa*.

MIT Scheme adopted an intermediate solution to this problem. Because of the difficulties of building an interactive system consistent with two distinct

[5]By "compilation time" I mean the time when Scheme source code is converted into an executable program. Thus, compilation occurs even when Scheme expressions are typed in or loaded interactively.

[6]The actual rules of CommonLisp, on the other hand, are quite complicated here, and even permit the possibility of changing *at run time* the type of binding to be used. To the author's knowledge no implementation of CommonLisp actually supports this option.

variable referencing and binding mechanisms, it provides a mechanism that does not affect the lexical nature of bindings in the language. Instead, it provides a mechanism allowing the existing lexical lookup to effectively reference different locations at different times, based on the dynamic execution of the program.

### 3.3.3 Fluid Variables

Rather than introduce dynamic binding, MIT Scheme provides the special form fluid-let, originally introduced by Sussman, to alter the value of an existing variable for use during the execution (time) of a particular segment of code. All variable references remain completely lexical, but the *locations* where these "fluid variables" are stored is altered. Syntactically, `fluid-let` is exactly the same as `let`. The solution to the earlier problem can be expressed by adding the following procedure to the package:

```
(define (with-new-radix procedure new-radix)
  (fluid-let ((radix new-radix))
    (procedure)))
```

`Fluid-let` has a slightly unusual semantics. It *does not* introduce a new binding (as does `let`), although the syntactic terminology refers to the variables in both cases as a "binding list.". Rather, it converts the existing bindings into what are termed fluid variables (hence the name). The execution of a `fluid-let` expression separates the dynamic execution of a program into two parts. Before entering the body of the `fluid-let`, and after exiting, the variables specified in the binding list (in this case `radix`) refer to a certain set of locations in the store. While execution is within the body, however, they refer to a new set of locations and the old locations are inaccessible. The new locations are created when control initially enters the body of the `fluid-let`, and they are initialized to the values specified in the binding list (in this case the value of the expression `new-radix`).

Because of the existence of first-class continuations in the language, the implementation of `fluid-let` is not trivial. In MIT Scheme it is based on the mechanism described in Appendix D which guarantees that any attempt to create a continuation during the execution of the body of the `fluid-let` correctly captures the locations of the fluid variables. When such a continuation is used, whether from inside or outside the execution of the body, those locations are restored.

Fluid variables are used in the MIT Scheme runtime system for maintaining the state of a large number of packages within the system. These variables include the primary input and output streams, current syntaxer tables for parsing programs, information about the current error or breakpoint context, and recursive state of the compiler. They provide a convenient encapsulation of an important construct whose power is needed in any large system. The use of fluid variables has completely replaced dynamic binding within MIT Scheme.

### 3.3.4 Fluid Variables in a Parallel System

Unfortunately, both the semantics and the mechanism used to implement `fluid-let` in MIT Scheme run into problems when the system is extended to parallel execution. Unlike a sequential system where the entire system is either inside or outside the dynamic extent of the body of a `fluid-let` form, MultiScheme as a system can be simultaneously inside *and* outside the body. Furthermore, it can instantaneously be inside of *many* bodies that change the locations of the same variables. Thus each task must somehow contain the information needed to locate the particular cell where the value of each of its fluid variables is currently stored.

MultiScheme resolves this situation by providing a run time mechanism that marks individual entries in an environment frame for special handling by the variable lookup and side-effect (`set!`) code. Since the MIT Scheme interpreter already examines the values of a variable that is referenced or modified to detect unbound and unassigned variable errors, this same test was extended into a more flexible mechanism. A type code (called trap on reference or trap for short) was assigned for this purpose. Every interpreted variable reference or `set!` operation examines the contents of the (lexically located) environment frame to check for special handling. If this trap code is found the slot also contains the information necessary to complete the reference. In many cases the compiler can omit this check, since it can be determined statically (at compile time) to be unnecessary.

In order to implement `fluid-let` the trap code is associated with a small block of data. This block includes a flag indicating that the variable in question is fluid as opposed to unbound or unassigned or some user-specified type. It also contains the cell where the value of the fluid variable is stored when outside of the influence of any `fluid-let` form. Furthermore, associated

with each task is a fluid binding list, a standard Lisp A-List mapping trap objects to cells for the value of the corresponding fluid variable. In order to allow the introduction of a `future` form within the code executed as part of the body of a `fluid-let` to function as it would in MIT Scheme, a newly created task inherits the fluid binding list of the task that spawns it, and it can therefore communicate with that task and its own "siblings" using the cells (and the fluid variables they represent) that are part of that list. Finally, a continuation also contains the fluid binding list that was in existence when it was created, and invoking that continuation in the normal manner restores those fluid bindings.

To summarize the mechanism that supports `fluid-let` in MultiScheme:

- When a `fluid-let` form is entered, each of the variables in the binding list is located using ordinary lexical lookup.
  - If the variable is not yet marked as fluid (by the presence of a `trap` type code in the value cell) create a new trap object that contains the current contents of the value cell as the contents of the "outside" cell. Store this object into the lexical value cell.
  - In either case, make a new entry on the task's fluid binding list. This new entry contains the trap object as key and the value of the expression in the `fluid-let` binding list as value.
- On every variable reference or `set!` operation, look up the variable using ordinary lexical lookup. If the value of the variable is a trap object for a fluid variable, look on the task's fluid binding list.
  - If the trap object is found in the fluid binding list, then the corresponding value cell is used in place of the cell in the lexical environment frame.
  - If the trap object is not on the fluid binding list, then the "outside" cell of the trap object is used in place of the cell in the lexical environment frame.
- When a new task is created, it inherits the fluid binding list of the creating task. The initial task has an empty fluid binding list.
- When a continuation is created, it contains the current fluid binding list as part of the saved computation state.

- When a continuation is applied to an argument (i.e. when a task `throws` to the continuation), the fluid binding list from the continuation becomes the task's fluid binding list.

Because fluid variables are referenced only rarely and the fluid binding lists tend to remain fairly short this mechanism has proven effective in MultiScheme. The mechanism is closely related to the deep binding mechanism originally used to implement dynamic binding in Lisp, and suffers from the same problems. If the fluid binding lists become long (typically because a few variables are being bound by a recursive procedure) then references to early entries becomes slow[7]. For this reason, most Lisps now implement dynamic binding using a technique known as shallow binding that multiplexes the value cell within the environment directly. An implementation of `fluid-let` using shallow binding was considered for use in MultiScheme, but was rejected because it would dramatically increase the time required to switch tasks.

### 3.3.5 Task-Private Storage

Fluid variables provide an excellent way for tasks to create their own storage. `Fluid-let` can be used to provide a value for a variable without altering the scope of the variable. Yet its accessibility is limited to the task that evaluated the `fluid-let` and any sub-tasks it spawns within the body. Furthermore, since the mechanism is supported by a uniform variable reference mechanism there is no need to modify existing code to cope with the fact that some of its free variables may become fluid variables at any time during the execution of the program. The mechanism is inherently dynamic, not static.

Yet this very flexibility can be a problem. The speed of a variable reference is governed by the usual issues relating to lexical lookup. For fluid variables this time must be augmented by the time required for a deep search of the task's fluid binding list. If a task references a variable that has a fluid binding by some *other* task, yet is not itself within the body of any `fluid-let` for that variable, it must find the "outside" binding of that variable. Unfortunately this requires searching the task's entire fluid binding list to discover that the task has no such binding. Then, of course, the correct binding slot

[7] By providing a cache for recently referenced fluid variables (flushed when a `throw` occurs and when a `fluid-let` body is exited), the cost of referencing fluid variables can often be significantly reduced.

is easily located from the trap object which was originally located. Thus the time for referencing such variables is proportional to the length of the task's fluid binding list (but see the footnote on page 74). There is, therefore, a desire to keep this list as short as possible.

At the same time, there is always a need for a task to contain data relevant only to its own workings and for which the ability to share that data with other tasks is minimal. For example, in MultiScheme running on the BBN Butterfly, each task has its own I/O stream that it uses by default. While the task itself must be able to reference these streams no application has yet been found in which any other task need be concerned with them. This task state can be supported directly using the fluid variables mechanism by placing a `fluid-let` into the code in the scheduler for creating tasks. This guarantees that every task has its own location in which to store the streams when they are needed.

While this solution works, it guarantees that each task's fluid binding list contains an entry for each of these task state variables. Furthermore, since each task inherits the fluid binding list of the task that spawned it, it contains an entry for each of these variables for each task in the chain of parent tasks back to the initial task created by the system. Thus the very act of creating a task will force that task to take a longer time to reference fluid variables than the task that created it. This observation is really just the same one that initiated the conversion from deep binding to shallow binding in traditional Lisp systems.

This difficulty is a very real one in MultiScheme, but there has been no single solution. Instead, the existing mechanisms support three different solutions employed for various parts of the task state.

1. For items that must be referenced by the lowest levels of the system and possibly the microcode of a machine, the task object itself contains a (fixed) number of slots. Since the task object can be manipulated from within MultiScheme, it is possible to view the task object itself as a data structure containing certain pieces of state. This, of course, requires that the state variables are fixed and known in advance. Furthermore, it requires that any code that manipulates this state be aware that it is stored in a particular fashion. The task data structure is described, along with a number of routines that deal with it, in Section 4.1.2.

2. One of the slots in the task structure can contain a standard lexical

environment frame. This frame contains the extensible portion of the task state. This solves the problem of allowing the task state to be conveniently extended from within MultiScheme, but it must still be manipulated explicitly. This mechanism is not, in fact, used in the current implementation.

3. The trap mechanism described for `fluid-let` is quite general. By placing different flags (actually a procedure) into the trap object it is possible to specify an arbitrary mechanism for resolving variable references. One other way to use this power is to specify that references to a particular lexical variable are to be resolved into references to the task state object, as implemented in either of the earlier two solutions. This permits the structuring of the name space via lexical scoping to be utilized without either fragmenting the task state or distributing through the code the knowledge of which variables are task state variables.

Providing a convenient syntax and interface to these mechanisms is an interesting area for further investigation. The mechanisms themselves are quite powerful and have easily satisfied all of the current demands for task state. Unfortunately they are rather hard to use and the difficulties that have arisen in this area so far do not need to utilize their full capabilities. As more complex applications are built, these mechanisms will almost certainly be employed to help control the modularity of the program design.

## 3.4 Introducing the Task

The history of MultiScheme includes a long series of experiments with the notion of a `task`. Steele[52] introduced the notion of a continuation (which originated in the area of denotational semantics) as a means of analyzing Scheme programs in the process of compilation. This "continuation passing style" of programming proved sufficiently interesting and powerful that the notion of using continuations as part of ordinary programming practice was expanded and demonstrated in a series of papers by Friedman and others at Indiana University[17, 20, 28]. The use of continuations as first-class data objects was added to the language, and they were hailed as a replacement for the standard `catch` and `throw` constructs of earlier Lisp systems. These continuation objects, since they encapsulate the interesting state of the system,

were the first candidates for tasks.

Using them it is easy to construct a multiprocess*ing* system, sharing a single processor among a number of "active" continuations. For example, the concept of an engine introduced by Haynes and Friedman[27] is one way of capturing this notion. At this time a continuation was purely a control object. It was noticed that a continuation is completely interchangeable with a procedure making it is easy to extend the continuation seen by a programmer to include other pieces of state. As described in Section D.5, the MIT Scheme system includes the dynamic state information as part of the continuation objects manipulated by programs. Again, these extended continuation objects are good candidates for the notion of a task.

In multiprocessing systems there is, however, a notion that is not explicitly visible. Not all continuations are treated in the same way. Some of them are candidates for execution by the system and others are merely objects created and manipulated by programs. The notion of a task refers to these former kinds of continuations. A continuation embodies work passively, the same way a procedure does (in fact, some Scheme systems implement continuations as procedures). A task, on the other hand, embodies work which for one reason or another should actually be undertaken. A task is active if a processor is actually working on it. It is suspended if it could use a processor if one were available but no processor is actually working on it. The system maintains a data structure that contains all of the suspended tasks so that when the processor chooses to cease executing the current (active) task it can choose a new task to execute. With the addition of placeholder objects and an event driven system, there is a third status for tasks. They can be inactive if some event must occur before it can be active again — typically the event will be the arrival of a value for a particular placeholder.

The next stage in the evolution of a MultiScheme task was motivated by the desire to make the system demand driven. With this goal comes the need to garbage collect unnecessary tasks, and this in turn makes the task distinct from the continuation object. The continuation embodies work to be performed. But the task embodies both work to be performed (a continuation) and the *reason* why the work should be performed. Should this reason become invalid, then the task can be removed from consideration for processor resources, i.e., it can be garbage collected. The task is a way to represent a continuation that is requesting processor resources, and the set of active and suspended tasks are what constitute the demanding agents of

the demand-driven system.

Maintaining knowledge about reasons for performing work and scheduling based on these reasons is the core of much work in artificial intelligence. Clearly it is not yet within the state of the art to use these techniques for scheduling and garbage collecting tasks at the lowest level of a programming system. Instead, a very simple operational definition can be (and was) used. The reason for performing a task was not, in fact, explicitly specified. Instead, the standard Lisp garbage collector was used as a model. The Scheme system has some designated objects (the root of the garbage collection) that are considered to be vital to the existence of the system. These must be preserved at all costs and are their own reason for existence. But they in turn depend upon the existence of (or point at or contain) other objects. The reason for existence of these other objects is the fact that the vital objects depend upon them. But these in turn depend upon yet other objects, and so forth. Thus every object that is preserved after a garbage collection remains because it is either a vital object, or there is a dependency chain from a vital object to the object. The same is true of tasks — a task has a valid reason to exist if it is either one of the vital objects or there is some path from the vital objects to reach the task.

But in a very real sense this begs the question. What are the vital objects in the system? In MIT Scheme there are a large number of them, but only two typically lead to retaining tasks. The first is the global environment since the values of variables in this environment can always be referenced[8]. Introducing a global name for a task is tantamount to declaring that the task must always be available, and hence it has a permanent reason for existence. Similarly, storing a task in a data structure or variable that can be located by a chain of ordinary accessors from the global environment gives the task a reason to exist.

The second object of interest is the task executing at the time of the garbage collection. Since the garbage collection primitive operation is invoked as any other primitive function it runs as part of the task that calls it. But the fact that a task was active at the time of a garbage collection is hardly a good reason to maintain the task. As a result, the interrupt handler that

[8]The access special form in MIT Scheme can be used in any lexical environment to reference variables in the global environment. The global environment is the only environment for which this is true, since it is the only environment that can itself be referenced without the use of a name.

initiates a garbage collection (see Section 3.1.1) guarantees that all processors switch to a task that exists only for the purpose of garbage collection.

The addition of placeholders to Scheme puts a new complexion on this problem. If we use tasks primarily as part of the `future` macro or a similar construction, then we always create a placeholder for the value of a task at the same time the task is created. In a very real sense, the purpose of the task is to calculate a value for that placeholder. This is reinforced by the fact that in a parallel- or multi-processing system the placeholder propagates through the system even while the value it represents is being calculated. Thus we need to represent the fact that a task's purpose is the calculation of the value for a specific placeholder. This is easily done by making the placeholder point to the task calculating its value. In this way if the garbage collector maintains any references to the placeholder it will also retain the task itself.

The final step in the evolution of the task comes from the decision to firmly link the notion of task to placeholders. Associating *every* task in the system with a placeholder and making that placeholder the "goal" for the task provides a simple linkage between tasks and placeholders that makes it considerably easier to grasp the notion of task. Tasks exist for the purpose of computing some value, and the placeholder associated with the task allows this value to propagate through the system even while the task is doing the calculation.

This simplification is nice, but it is important to keep in mind that a task may do other things in the process of calculating the value of its associated placeholder. There is an important analogy between tasks with a purpose and procedures. We may think of a procedure as existing for the purpose of computing some value. Yet some very useful procedures loop forever and produce no value. Similarly, a task may include an infinite loop and hence its associated placeholder may never receive a value. Similarly, a procedure may deliberately side-effect a variable or data structure or invoke a continuation thereby producing a result in a place other than the one indicated by its ostensible purpose. Likewise, a task may execute code that contains an explicit call to `determine!` and thereby contribute a value to a placeholder other than the one it is apparently intended to compute. The association between the task and the placeholder should be thought of as providing a form of ordinary behavior chosen to support the expected common pattern of usage rather than an inherent and unshakeable condition.

To summarize this model, here is a recap of the three main MultiScheme

objects relevant to (potentially) simultaneous computations.

**Placeholders**

A placeholder represents a value that is not yet computed. Placeholders associated with a task (or a set of tasks) provide the primary "computational motive force" for those tasks. There are typically no references to tasks except through the placeholders for the values they are calculating. `Touch`ing a placeholder may cause a task to be generated, providing a lazy evaluation mechanism.

**Continuations**

A continuation represents a control state, a set of fluid variable bindings, and a corresponding point in the system-state-space[9] (see Appendix D). These three parts can be restored independently, or they can be restored jointly by applying the continuation to an argument. This corresponds to transfer of control (`goto`) in other languages. There is a special continuation, used when a task is generated, that corresponds to task termination. It stores the value supplied to the continuation as the value of the placeholder associated with the task that is terminating.

**Tasks**

A task represents a line of computation that is expected to ultimately lead to the computation of a value for a particular placeholder. When a task is created the placeholder that is its goal must be specified, and that placeholder forms the primary source of "computational motive force" that will keep the task running. Typically, if the placeholder is no longer needed the task will be removed from the system (along with the placeholder), although it is possible to retain a task explicitly if necessary. Tasks are created with the special "task termination" continuation which will cause the value supplied to the continuation to be placed in the task's placeholder and then terminate the task.

[9] Continuations also contain other information relevant to the control state of the system, such as the interrupt mask and compiler register set. The fact that this is not encoded in the control state itself is an implementation detail.

## 3.5 Demand-Driven Computation

One of the important design factors in MultiScheme is that computation be demand driven — not in the sense of using "call by need" parameters or a "lazy evaluator," but rather in the sense that a computation continues to run only as long as it is serving a useful purpose. The notion of a task, introduced in Section 3.4, captures the essence of this drive by providing an explicit goal for each computation in the system. One way of stating this demand-driven computation mode is by saying

> A computation continues only as long as the value it is computing is still needed.

By reversing the statement, it can provide the answer to a problem raised when converting a standard sequential Scheme program into a parallel version.

> As long as the value of a computation is still needed that computation must continue.

The parallel between memory allocation and task allocation is very strong. Just as Lisp allows memory to be allocated and invisibly handles the release of that memory when it is no longer needed by the computation at hand, MultiScheme allows tasks to be created and will remove them when they are no longer needed. Thus the need to abort a task is minimized, just as the need to deallocate memory is minimized. Furthermore the technique used in both cases to force the deallocation is the same. To release memory the programmer must find *all* objects that reference the memory being released and guarantee that they no longer reference the object in question (typically this means storing `'()` or some other non-pointer as the value of a variable that is normally the only entry point to a data structure). Similarly, in MultiScheme the programmer must guarantee that the goals of all the unneeded computations have been provided a value by the explicit use of `determine!`[10].

This item of philosophy becomes most obvious when considering the use of `call-with-current-continuation` (creation of a continuation object)

[10] For more drastic cases it is possible to use `within-task` to force a task to halt, but this can leave the system in an inconsistent state if not done with care.

and `throw` (the use of such an object). In a single processor system there are a number of ways to understand this interaction, and a number of styles for their use. In Scheme, continuations are merely a convenient way of writing certain procedures. The decision to explicitly write out a procedure or to make use of a continuation is purely one of user convenience and should imply no deep consequences. Based on this line of reasoning, continuations in MultiScheme are treated exactly as any other procedure object. They have the effect of changing the implicit continuation being used by the task that calls them, but the task and its goal remain intact.

```
(define (f y)
  (call-with-current-continuation
   (lambda (send-answer)
     (define (transform fn)
       (let ((fn-of-y (fn y)))
         (if (= 0 fn-of-y)
             (send-answer 'no-good)
             (hairy-computation fn-of-y))))
     (+ (transform sin) (transform cos)))))
```

Figure 3.4: Multi-level Exit using `Call-With-Current-Continuation`

This decision is not without ramifications and is not taken lightly. By adopting a demand-driven model and the notion of goal-oriented tasks it is no longer true that `(touch (future <exp>))` is equivalent to `<exp>` in functional programs. Consider the program of Figure 3.4. This is a fairly common use of `call-with-current-continuation` where it is used to provide a way to return a result from a nested procedure call. This use is similar to the way `catch` is used in CommonLisp. If `transform` detects a problem condition it immediately returns an answer of `NO-GOOD` to the program that called `F` and neither the hairy computation nor the addition operation is performed. Thus, for example, `(future (f 0))` will ultimately return an answer of `NO-GOOD`.

If we wish to introduce parallelism into this program, one obvious way to do it would be to perform the calls to `transform` simultaneously. Thus we

would change the final line of the program to

```
(+ (future (transform sin))
   (future (transform cos)))
```

The program now results in an error since `(future (transform sin))` creates a new task whose goal is the computation of the transform. This task then uses the `send-answer` continuation to deliver a result of `NO-GOOD` as the result of the call to `(f 0)`. Since this is the end of a task (the one created by `(future (f 0))`, the answer is stored in the goal of the current task — and this is then added to the other transform, resulting in the error.

---

```
(define (alternate-answer body)
  (define final-result
    (delay (body (lambda (early-answer)
                   (determine! final-result early-answer)
                   (kill-task)))))
  (touch final-result))
```

Figure 3.5: A Replacement for `Call-With-Current-Continuation`

---

What is the basis of this problem? In terms of the demand-driven computing model the difficulty here is that one particular task has the goal of computing a value for `(f 0)`. In order to achieve this goal, additional tasks are created and the goals for these tasks would ordinarily be combined by the original task to form the value of its goal. The use of a continuation here is intended to allow *any* task to provide a value for this goal.

This can be easily expressed by replacing the call to `call-with-current-continuation` with a slightly different procedure, `alternate-answer` shown in Figure 3.5. This works by creating a new task to execute the body of code that, in the normal case, is executed by the originating task. This new task computes as usual, but the procedure that replaces the continuation in the original version explicitly determines the placeholder for the generated task. Thus, regardless of which task tries to use this continuation, it is the original placeholder that receives the value. In this example, the task that

generates the answer then "commits suicide," leaving its own placeholder without a value[11] on the assumption that there can be no need for the task's value if the alternate answer is being returned. If the task should continue computing then the call to `kill-task` can be omitted.

This solution also demonstrates one important consequence of the demand-driven approach. Using continuations for multi-level exit on a parallel processor leaves open the question of how to kill other tasks that were spawned after the `call-with-current-continuation` was executed by the original task. This "spawning tree" oriented approach to programming can be completely avoided in MultiScheme. The problem is non-existent: those tasks that continue to compute after a value is available for the placeholder created by the `call-with-current-continuation` are precisely those that are attempting to compute values needed by the system even when that value is known. To have killed these tasks merely because of the spawning relation would lead to deadlock later — a problem avoided by allowing the garbage collector's traversal of active memory to detect and remove tasks that are unreferenceable.

## 3.6 Summary

Extending MIT Scheme for use on a parallel processing system required four major areas to be explored: processor coordination, data storage for tasks, dynamic state and exception handling, and the distinction between tasks and continuations.

Section 3.1 introduced the mechanisms used to coordinate the processors in the system: `synchronizer`s which act as a "starting gate" to guarantee that all processors are at a given part of the computation, and `global-interrupt` to get the attention of all processors and demand their cooperation in initiating some event. These were used to support three important system operations: initiating garbage collection, pausing the system for observation, and reacting to user interrupt requests.

The separation of the name space in a way which enables tasks to access information that they may not necessarily share with others is discussed in

[11] The procedure `kill-task` isn't specified here, so it can arrange to place an error value in the placeholder if desired. Otherwise, the procedure `next` discussed in Section 4.2.2 would be a reasonable choice.

Section 3.3. The problem is divided into two different jobs. The fluid variables of MIT Scheme are extended into the parallel processing domain to provide one form of support. In addition, three different techniques (see page 75) for providing totally private task information were discussed.

Finally, the distinction between a continuation and a task in MultiScheme was introduced in Section 3.4. The motivation for this distinction and the important notion of demand-driven computation was introduced. The use of task goals to support the garbage collection of useless tasks created for speculative computation was described in Section 3.5.

# Chapter 4

# Implementing the Scheduler

The MultiScheme scheduler provides a convenient interface, in the form of a package of procedures, between MultiScheme programs and the underlying virtual machine. Some of the procedures are invoked by programs written in MultiScheme while others are invoked as part of the trap or interrupt handling of the virtual machine. The scheduler is itself written in MultiScheme and is relatively small (20 pages of code including utility routines). This has proven to be an important factor in the development of MultiScheme, providing a localized and flexible base for a number of experiments with the nature of event-driven computing.

This chapter serves four purposes. First, it constitutes a "proof by example" that the MultiScheme language as described in Chapters 2 and 3 is a powerful systems programming language for a parallel processor. It also serves as the first extended example of programming in this language. Third, it provides an English description of a number of the constructs used in the implementation and thus acts as a form of documentation for people dealing with the actual program. Finally, the implementation of `disjoin` in Section 4.3.2 and Figure 4.9 fulfills the promise made in Section 2.5 of demonstrating an event-driven mechanism for implementing the `amb` and `fair-merge` operations.

This chapter discusses each of the major operations supported by the scheduler: task creation (Section 4.3), task suspension and task switch (Section 4.4), storing a value into a placeholder (Section 4.5), and transition from parallel processing to single task execution (Section 4.6). The rough outline of the scheduler (the services it supports and the inter-relationship between

these services) has proven quite robust over time. Even as the system grew to support more kinds of event driven computation, the core of the scheduler as described here has remained almost constant. The scheduler was originally intended to be, and remains, a highly flexible body of code. The scheduler described here is the "standard" scheduler as it currently exists. As new applications are developed, driving the system toward new modes of computation, the data structures of the scheduler are modified to accommodate the new requirements. Users are encouraged to examine and understand the scheduler, and feel free to modify it for their own needs. Naturally, such modifications must be undertaken with a good deal of care. But these modifications have proven useful in the past and have in some cases been formalized and added to the standard MultiScheme scheduler.

By its very nature, the discussion in this chapter is more closely focused on implementation details than are the earlier chapters. The presentation is roughly bottom up, describing the data structures in Section 4.1, general utility routines in Section 4.2, and then the user-visible routines. In order to avoid an overwhelming amount of detail the examples included in this chapter are simplified versions of the actual procedures in the scheduler. These simplified versions present the important core of each procedure, and should be considered more closely related to pseudo-code than to fully worked out implementations. In many cases the versions presented here will not work correctly in the actual implementation of MultiScheme. This comes from a variety of reasons, including race conditions and name changes introduced to be more consistent with the terminology of this document. Readers interested in the complete versions of these procedures should contact the author for a current version of the scheduler code.

## 4.1 Scheduler Data Structures

Much of the work of the scheduler procedures revolves around the correct maintenance of the data structures that implement placeholders (see Section 2.1), tasks (see Section 3.4)[1], and a queue of tasks that are ready to

[1]At this time, the task and placeholder data structures are actually implemented as a single Scheme object. This works, but is both a conceptually poor idea and implementationally clumsy. The work of splitting the two apart has been planned and scheduled, but is not yet underway. This chapter describes the system as it is expected to be built after

run.

For this description, all access to data structures is assumed to be through mutators and selectors for each part of the structure. Thus, corresponding to the goal slot of a task data structure there are two procedures: `task.goal` returns the goal of a given task and `set-task.goal!` stores a new goal into the task data structure.

### 4.1.1 Placeholders

Placeholders are the primary vehicle connecting the scheduler (and hence programs written in MultiScheme) with the underlying support for parallel processing. Placeholders are created by a scheduler procedure, normally as part of the task creation process (see Section 4.3). Supplying a value for a placeholder (through `determine!` and `mutably-determine!`) is also supported by scheduler procedures (see Section 4.5). Detection of placeholders and automatically forcing them is built into the primitive operations and the underlying machine itself, as described in Section 4.4.

| Name | Notes |
|---|---|
| Determined? | |
| Lock | |
| Value or Waiting Queue | *See text* |
| Motivated Task | |

See text for complete description

Figure 4.1: Placeholder Data Structure

The placeholder data structure is shown in Figure 4.1. Each of the fields is described below.

**Determined?**
: A tri-state flag that indicates whether the placeholder: (a) has no value yet; (b) has an immutable value; or (c) has a mutable value. This flag is used by the underlying machine to test whether a `touch` of this

the next release of the standard scheduler.

placeholder should trap into the scheduler (as discussed in Section 4.4) or extract the current value and continue.

**Lock**

A standard mutual exclusion lock used to indicate that the placeholder is currently being modified by MultiScheme code. The underlying machine uses this lock to implement the primitive mutual exclusion procedures `lock-placeholder!` and `unlock-placeholder!`. In addition, when a placeholder is locked the underlying machine will not convert a placeholder into its value during garbage collection or variable look-up. Under ordinary circumstances this conversion (called splicing) is an important optimization. The lock permits MultiScheme code that explicitly manipulates placeholders (typically within the scheduler) to suppress this splicing. This assures that a placeholder will not suddenly transform into another object while the data structure itself is being examined.

**Value**

Stores the value of the placeholder if it is either mutably or immutably determined.

**Waiting queue**

A queue of tasks currently waiting for this placeholder's value to be determined. This queue is built using weak cons cells, as described earlier in Section 2.2.2, since membership in this queue does not constitute a reason for the task to continue computing. The underlying machine does not reference this information (it is handled only by MultiScheme code, typically within the scheduler). Since this queue must be empty when the placeholder already has a value, the implementation overlaps the storage space for these last two items. There is no inherent reason why this structure is a queue (rather than, say, a stack) since all of the items are released for execution at the same time.

**Motivated task**

The task that has the computation of a value for this placeholder as its goal. As with any item other than a weak cons cell, the garbage collector *does* trace through this link. Thus it serves to retain the

task that is computing the value of this placeholder as long as the placeholder itself is needed.[2]

### 4.1.2 Tasks

The task data structure contains a variety of information, but is not directly referenced by the underlying machine. Tasks represent work that has been requested to be performed, and they are the objects that the scheduler has the underlying machine store on its work distribution queue (see below).

In order to support garbage collection of no longer useful tasks, the root used by the garbage collection algorithm contains a particular set of tasks whose continued existence is required by the user interface to MultiScheme (those that can be reached directly using the ordinary keyboard interrupt characters). Other tasks are retained only if they can be reached either from this initial set of tasks (because one of these initial tasks is waiting for (see below) the value of a placeholder and that placeholder's motivated task references another task) or from the global environment.

| Name | Notes |
|---|---|
| Goal | Placeholder associated with this task |
| Lock | |
| Code | Work to be performed when task is next run |
| Status | *See text for details* |
| Original Code | For debugging purposes |
| Task-Private Data | See Section 3.3.5 |
| Waiting For | Placeholder(s) for which this task is waiting |
| Wake-up Value | *See text for details* |

See text for complete description

Figure 4.2: Task Data Structure

The task data structure is shown in Figure 4.2.

[2]The motivated task could be extended to a list of tasks, but the code shown in this chapter does not support this option.

**Goal**
: The placeholder that is the goal for this task. When a task is actively computing, this placeholder is known as the current placeholder for the processor doing the computation. When a task executes the termination continuation (see Section 4.3.3) it stores the computed value into this placeholder.

**Lock**
: A standard lock to serialize access to the task description.

**Code**
: The code to run in order to re-activate this task. If the object stored here is not applicable (i.e. neither a procedure nor a continuation) then either the task is already active or for some reason it cannot be re-activated (it may have finished computing and not yet been garbage collected, for example).

**Status**
: The current state of this task. This is one of:

| | |
|---|---|
| **created** | Task is newly created |
| **delayed** | See `delay-policy`, Section 4.3.2 |
| **determined** | Task is finished |
| **disjoin** | Waiting for the first of several placeholders |
| **paused** | Stopped by `pause-everything` |
| **runnable** | Available for execution |
| **running** | Actually in possession of a processor |
| **waiting** | Waiting for a specific placeholder |
| **within-task** | Running, but see Section 3.1.4 |

**Original code**
: For debugging purposes this contains the expression that the task was created to evaluate.

**Task-private data**
: See Section 3.3.5.

**Waiting for**

If the task has status waiting or disjoin this specifies the placeholder (or list of placeholders) for which it is waiting. This is an ordinary (strong) list, since the value of these placeholders is necessary for this task to continue its own computation. Hence this task represents a reason for the tasks that are computing a value for each of these placeholders to continue their computation.

**Wake-up value**

When the task is awakened, the code is passed this value as its argument. It is primarily used in the implementation of `disjoin` (see sections 4.3.2 and 4.4).

In addition, it has long been expected that some information might be stored here for use by user supplied scheduling policies. This information could indicate task priority or estimated time required for the value to be computed. This has not been implemented since no applications have yet required sophisticated scheduling policies.

### 4.1.3 Runnable Task Queue

The subject of task distribution is one on which a great deal of work has been done, both theoretical and practical. The MultiScheme scheduler is built using three primitive operations (`put-work`, `get-work`, and `drain-work-queue`) that encapsulate the choice of task distribution mechanism implemented by the underlying machine. The model employed by the scheduler is that it can announce that a task needs processing resources using `put-work`, and when it needs more work to do it will retrieve a task using `get-work`. Because the shared heap contains all of the task, placeholder, stack, and other structures needed for a computation, no guarantee is made that a task will be run on the same processor which announced that it needed resources.

In addition, there are times when the system must retract work that has been declared to be available (such as during garbage collection initiation). The primitive operation `drain-work-queue` returns a standard data structure (composed of weak cons cells) containing all items that were available for work at the time the primitive was called. It leaves the work queue of the underlying machine empty. As usual, synchronization and serialization are the responsibility of the MultiScheme code, not the primitive itself. Thus,

it is possible for one task to drain the work queue while other tasks are still active and creating new entries on the queue. The results will be consistent (a task that is available for work will be removed either by a call to `drain-work-queue` or a call to `get-work` but not both), although it may not represent an instantaneous snapshot of the internal data structures.

## 4.2 Overall Concepts and Utility Routines

The scheduler is organized around the data structures described in Section 4.1 and two additional notions. The first, described in Section 4.2.1, is atomicity and critical sections of code. These are supported through a system of priority interrupts (within a single processor) and a set of data object locks (between processors). The second is the task state and task switch operations, described in Section 4.2.2, supported through the use of Scheme's continuations.

### 4.2.1 Atomicity

As with any operating system scheduler, most of the routines in the scheduler must appear to occur without interruption. The fact that these routines are written in Scheme, however, does not permit them to run completely uninterrupted: the garbage collector cannot be suppressed for even short intervals without serious consequences. As a result, most of the operations are written to raise their own interrupt level to prohibit any kind of interrupt *except* garbage collection, and the garbage collection code guarantees that any task that is running at a raised interrupt level will continue to run after the garbage collection. This notion is embedded in the macros `atomic` and `define-atomic` which are used liberally throughout the scheduler implementation. To make the code more easily understood, however, these have been omitted from the simplified versions described here.

A second standard problem, exclusive access to certain data structures, also exists in the scheduler. As Halstead[24] shows, users can ordinarily use placeholders and certain primitive atomic operations to implement semaphores. These solutions can be extended to include other standard multi- and parallel-processing interlocks. Unfortunately, the scheduler is in large part responsible for implementing the semantics for placeholders that these solutions exploit.

The scheduler, therefore, is built using two major utility routines that are in turn based on spin-locks.

The primitive operation `lock-placeholder!` is used to implement the more complicated of the two utility routines. It is called with one argument that is normally a placeholder. `Lock-placeholder!` either immediately returns #f (if its argument is not a placeholder) or it waits until it is able to acquire the lock which is part of the placeholder data structure and returns a value of #t[3].

Using this primitive, the following scheduler utility routine can be written:

```
(define (With-Placeholder-Locked Placeholder Procedure)
  (atomic
   (if (lock-placeholder! Placeholder)
       (let ((result (Procedure #t)))
         (unlock-placeholder! Placeholder)
         result)
       (Procedure #f))))
```

As can be seen, `With-Placeholder-Locked` runs a procedure with a given object (expected to be a placeholder) locked. The procedure receives an argument that indicates whether the object is in fact a placeholder. Notice that when the object is not a placeholder (and hence the procedure is given the argument `#f`) the object is *not* locked while the procedure is running, since MultiScheme does not provide any standard way of locking arbitrary objects.

A similar utility routine, `With-Task-Locked` is also supplied. It locks a task (its first argument) and then calls a procedure (its second argument) indicating whether the task is actually runnable. When the procedure returns the task is unlocked. Unlike `With-Placeholder-Locked`, the task *is* always locked when the procedure runs since the race condition that exists for placeholders is not a problem with tasks.

[3] A primitive is included for this purpose only because of the complexity of writing the *correct* MultiScheme code to deal with a potential race between one processor setting the value of a placeholder and another processor attempting to lock it.

### 4.2.2 Task Switch

When a processor changes tasks it is really performing into three separate operations. The first operation captures the current state of the task in a way that allows it to be restarted later. The second chooses a new task for execution, and the third activates a chosen task. Task termination (as described in Section 4.3.3) is nothing more than performing the last two steps but not the first. Task creation (see Section 4.3.2) may include the first and third steps with a standard choice for the second.

The first operation, capturing the current state of a task, is done using a variant on MIT Scheme's `call-with-current-continuation` procedure. Once a task is suspended it will be resumed only once, and then that state will later be suspended and so forth. Thus, unlike an ordinary continuation object, the object that denotes a suspended task state need not be able to be invoked multiple times. In implementation terms, this means that a certain amount of copying of continuation stack entries can be avoided with task suspensions. While this detail is an important efficiency concern, it has not been called out in the code shown here.

While acquiring a representation of the current state of the computation is simple, actually storing it in the task data structure is not as straightforward. In Section 3.1.4 an important user operation, `within-task`, was introduced (the relevant code is shown in Figure 3.2 on page 64). Calling `within-task` to modify the operation of a task which is already running marks the task data structure to indicate the work that must be performed, and it is the responsibility of the task when it next saves its state away to arrange to perform that work when the task is next activated.

The routine `store-my-state`, shown in Figure 4.3 is provided to support this operation. It allows scheduler routines to specify the state to be used when the task normally regains control (the argument `state`), and additional work to be performed on the task data structure while it is locked (`while-locked`). As you can see, it locks the current task data structure and then stores either the specified state or a procedure that first executes the work specified by a call to `within-task` as the work to be performed when the task is next activated.

One other common packaging of this first operation is provided by `release-task`, shown in Figure 4.4. In this case, the intention is to release the current task for (possibly parallel) execution and then execute some other code while the

```
(define (store-my-state state while-locked)
  (let ((my-task (current-task)))
    (with-task-locked my-task
      (lambda (am-I-runnable?)              ; (1)
        (if am-I-runnable?
            (begin
              (set-task.code! my-task
                (if (eq? (task.status my-task) 'WITHIN-TASK)
                    (let ((within-task-code (task.code my-task)))
                      (lambda (wake-up)     ; (2)
                        (within-task-code wake-up)
                        (state wake-up)))
                    state))                 ; (3)
              (while-locked my-task)))))))  ; (4)
```

**Notes:**

1. Find and lock the current task data structure.
2. If the task is expected to continue running but has been marked for special handling by `within-task` (see Figure 3.2 on page 64), then when the task next awakens it must first execute the code specified in the call to `within-task` and then continue on to its ordinary computation.
3. Under ordinary circumstances, the state to be stored is just the state specified by the caller.
4. If the task will continue to run, call the user-specified procedure while the task data structure is still locked.

Figure 4.3: Saving state for future execution: `Store-My-State`

```
(define (release-task thunk)
  (call-with-current-continuation
   (lambda (my-state)
     (store-my-state my-state
       (lambda (my-task)
         (set-current-task! 'STATE-SAVED)
         (set-task.status! my-task 'RUNNABLE)
         (put-work my-task)))
     (thunk))))
```

Figure 4.4: Relinquishing the processor: `Release-Task`

processor is temporarily not performing an existing task. The only non-obvious part of `Release-task` is why it must be provided with the code to be executed. Notice, however, that the code is executed as part of the procedure called by the `call-with-current-continuation` operation. Thus it is executed by the calling task, but *not* when that task is resumed by calling the `my-state` continuation.

The second operation, choosing a task to perform, is most often deferred to the underlying machine, using the primitive `get-work` to select the task:

```
(define (next)
  (Set-Current-Task! 'WAITING-FOR-WORK)
  (run (get-work)))
```

The third operation, activating a chosen task, is the most complicated. This job is handled by the procedure `run`, shown in Figure 4.5. It consists mostly of routine housekeeping activities. The task being activated is first locked and tested to see if it is actually runnable. If so, the status is changed to running and the code and wake-up value are extracted from the task data structure. The task is then unlocked, and either the code is activated with the appropriate wake-up value as its argument or (if the task turned out not to be runnable) an alternative task is chosen using `next`.

```
(define (run task)
  (define what-to-actually-do
    (With-Task-Locked task
     (lambda (Still-Runnable?)
       (if Still-Runnable?              ; (1)
           (let ((code-for-new-task (task.code task))
                 (wake-up-value (task.wake-up-value task)))
             (set-task.status! task 'RUNNING)
             (set-task.wake-up-value! task '())
             (Set-Current-Task! task)
             (lambda ()                 ; (2)
               (code-for-new-task wakeup-value)))
           next))))                     ; (3)
  (what-to-actually-do))                ; (4)
```

**Notes:**

1. Test the task to see if it is actually runnable.
2. If the task is runnable, this procedure will restart it.
3. If the task is not runnable, this procedure will select an alternate task and start it instead.
4. Actually call the procedure chosen in steps 2 and 3 above.

Figure 4.5: Activating a chosen task: the `run` procedure

### 4.2.3 Other Utility Routines

There are a handful of other utility routines that are referenced later in this chapter.

- A task data structure and its related placeholder can be created using `(Make-Task)`, which makes the pair simultaneously and supplies a standard set of default values for all of the information required.

```
(define (immutable? placeholder)
  (eq? (placeholder.known? placeholder) #t))

(define (undetermined? placeholder)
  (eq? (placeholder.known? placeholder) #f))

(define (determined? placeholder)
  (not (undetermined? placeholder)))

(define (mutable? placeholder)
  (and (determined? placeholder)
       (not (immutable? placeholder))))
```

Figure 4.6: Tri-state Flag Representation

- The three possible states of the determined? slot of a placeholder are: `#t` indicating that the placeholder has an immutable value; `#f` indicating that it has no value at all; and anything else indicates that the value is mutable. This is captured in the three procedures shown in Figure 4.6. Notice that the placeholder must be locked in order to safely perform these operations.

- A task that has been waiting is activated using the procedure `activate` shown in Figure 4.7. This procedure tests whether the task is still runnable and is in fact waiting for the condition that has occurred (as indicated by the `test`). It then updates the task data structure

```
(define (activate task test wake-up-value)
  (With-Task-Locked task (lambda (task-runnable?)
    (if (and task-runnable? (test (task.status task)))
        (begin
          (set-task.waiting-for! task '())
          (set-task.status! task 'RUNNABLE)
          (set-task.wake-up-value! task wake-up-value)
          (put-work task))))))
```

Figure 4.7: Activating a waiting task

and releases it for distribution using the underlying machine operation `put-work`.

- The procedure `Saving-State`, shown in Figure 4.8 was mentioned earlier in Section 3.1.3. It allows a task to save its state away and then execute a selected piece of code (`thunk`). The code is run in a continuation that is part of the root of garbage collection and not as part of the task that called `saving-state`. This permits the garbage collector to reclaim the originating task if necessary. When the code finishes execution, another task is selected for execution rather than returning to the original task. A good way of thinking about `saving-state` is that it performs a task switch (into a non-existent task) and executes the `thunk` in the new task.

- Two "ugly" procedures, `Current-Task` and `Set-Current-Task!`, are provided to keep track of the task that is currently executing. These maintain data that is private to the process*or* and are used to implement the higher level concepts of task-private data as described in Section 3.3.5. The author is too embarrassed to reveal the implementation.

- `Weak-list->list` converts a list composed of weak cons cells into one composed of ordinary cons cells.

```
(define (saving-state thunk)
  (release-task (lambda ()              ; (1)
    (within-control-point               ; (2)
     the-error-continuation
      (lambda () (thunk) (next))))))
```

**Notes:**

1. Release the current task for potential parallel execution.
2. Begin execution within `the-error-continuation` which was created at system boot time, and does not reference any other continuations.

Figure 4.8: Code for `Saving-State`

- A package of queue manipulation routines, including `enqueue`, `dequeue`, and `queue-contents`. Using these the operation `add-to-waiting-queue!` is implemented:

```
(define (add-to-waiting-queue! placeholder task)
  (enqueue (placeholder.waiting-queue placeholder) task))
```

## 4.3 Task Creation and Termination

Creating a task in MultiScheme really has four steps: create a continuation, create a task (i.e. the task data structure), create a placeholder, and schedule the running and newly created tasks for (possibly parallel) execution. Each of these can be performed independently and then combined to provide specialized handling of unusual cases. Tasks are normally created, however, by using the `future` macro.

This macro has one required argument, the expression to be executed in parallel, and an optional policy used to schedule the parent and child tasks.

Thus, the macro expands

```
(+ (future expression policy) (future expression2))
 into
(+ (spawn-task (lambda () expression) policy)
   (spawn-task (lambda () expression2) parent-gets-priority)
```

The remainder of this section consists of a description of the `Spawn-Task` procedure (Section 4.3.1), alternatives to this standard method of task creation (Section 4.3.2), and finally the handling of task termination (Section 4.3.3).

### 4.3.1 Ordinary Task Creation

As described above, most of the work of creating a task is ordinarily carried out by `Spawn-Task`. This is merely a standard way of using the four steps mentioned above.

**Create a continuation.**
One of the parts of the task data structure, described above, is the `code` it is to execute. `Spawn-Task` creates a continuation for this purpose (see the discussion below). The continuation could be expressed as the procedure:

```
(lambda (ignored-argument)
  (termination-continuation (thunk)))
```

In this procedure, `thunk` is the first argument to `Spawn-Task`, the procedure created from the `expression` by expanding the `future` macro. The `termination-continuation` is a specific (primitive) continuation that indicates the end of a task. The handling of this continuation is described in Section 4.3.3.

**Create a task data structure.**
The space for the data structure that was described in Section 4.1.2 must be allocated and initialized. The `code` slot is filled with the continuation created in the previous step. The `goal` slot is filled with the placeholder created in the following step.

**Create a placeholder.**
This acts as the goal of the task being created. It will receive the value computed by that task when the termination continuation is reached. The task created in the previous step is made the motivated task of the placeholder.

**Schedule the tasks.**
`Spawn-Task` calls the user-supplied policy routine to schedule the current (spawning) task and the newly created task. The default routine, `Parent-Gets-Priority` (shown in Section 4.3.2), releases the new task for potentially parallel execution. The value returned by `Spawn-Task` to the task that called it is the newly created placeholder.

The decision to use a continuation (rather than a procedure) for the initial code of a task is not completely arbitrary. If a procedure is used the task switch code in `run` (see Figure 4.5) that activates the newly created task would be nothing more than a procedure call. But procedure call includes passing an implicit continuation for use when a value is returned, and this continuation will in some way reference the task that made the procedure call. This prevents the garbage collector from reclaiming that task as long as the newly created task is in existence.

In implementation terms, which may be easier to understand, procedure call is handled by using a single stack to hold continuations (return addresses). If the initial code for a task were simply a procedure then the stack used for the new task when it first runs would be the same as the stack of the task that was relinquishing the processor. This works perfectly well but leads to a form of cactus stack implementation that has the garbage collection problem mentioned above.

By explicitly building a continuation, however, task switch becomes the same as invoking a continuation that does *not* implicitly reference the old task. The continuation created when the task is created is an initial stack frame and task switch (now `throw`) causes the stack to be switched as well.

### 4.3.2 Alternative Ways to Create a Task

The task creation code is modularized into the four steps described above. Actually utilizing these individual components is unusual since the flexibility

available using the second (policy) argument to `Spawn-Task` is sufficient for most problems. To make this power easily available, two alternative policies are included in the scheduler package along with the default policy.

The standard policy, `Parent-Gets-Priority` is very efficient:

```
(define (Parent-Gets-Priority new-task)
  (put-work new-task)
  'CHILD-QUEUED-FOR-EXECUTION)
```

This policy gives processing priority to the parent task. That is, the task that calls `Spawn-Task` continues to run after the call, while the task which is created is scheduled for (possibly parallel) execution. Since the task and its associated placeholder have been made and initialized by `Spawn-Task`, all that must be done is to make the new task available for computation. This is done using the underlying task distribution mechanism, implemented by `put-work`. (In this, as in the other policies, the value returned by the policy is ignored by `Spawn-Task` but is useful in debugging the scheduler itself.)

Halstead argues, in his overview of Multilisp[24], that this standard policy can lead to undesired performance characteristics as a system reaches saturation. He suggests a strategy in which the parent task is deferred while the child task immediately begins execution. This is implemented using the `Child-Gets-Priority` policy.

```
(define (Child-gets-priority new-task)
  (release-task (lambda () (run new-task))))
```

The third policy, `Delay-Policy`, marks the spawned task as delayed and does *not* release it for parallel execution.

```
(define (Delay-policy new-task)
  (set-task.status! new-task 'DELAYED)
  'OK-I-DELAYED-IT)
```

Instead, the first task that touches the goal placeholder associated with the newly created task will release that task for execution (see Section 4.4 below).

In addition to these three policies, there is one other case that occurs sufficiently often to be provided standardized support. This is the ability to wait for the first of a number of placeholders to return a value. This ability, implemented by the procedures `disjoin` and `await-first` of the scheduler, is the key to the `amb` and `fair-merge` procedures discussed in section 2.5.

```
(define (disjoin . Placeholders)        ; (1)
  (let ((My-Task (Make-Task)))
    (let ((My-Placeholder (task.goal My-Task)))
      (set-task.status! My-Task 'DISJOIN)
      (set-task.waiting-for! My-Task Placeholders)
      (set-task.code! My-Task
        (lambda (awakened-value)        ; (2)
          (determine! My-Placeholder awakened-value)
          (next)))
      (for-each                         ; (3)
       (lambda (Placeholder)
         (add-to-waiting-queue! Placeholder My-Task))
       Placeholders)
      My-Placeholder)))                 ; (4)
```

**Notes:**

1. As explained in the text, this code does not deal with a number of important possibilities.
2. Code to be run when this newly created task is activated (i.e. when one of the placeholder receives a value). This is one of two major race conditions that the complete version handles. If more than one task completes, `My-Placeholder` may already have a value when this code is run.
3. Enqueue this task on the `waiting-queue` of each of the placeholders. This is the second of the major race conditions. In the process of enqueuing, it may be discovered that one of the placeholders already has a value which must then be returned instantly.
4. The value returned by `disjoin` is the placeholder that will ultimately receive the value of the first computed placeholder.

Figure 4.9: Simplified Code for `disjoin`

The actual code for these procedures is complicated because it must deal with the possibility that one of the placeholders has already received a value before the operation has been completed, and because more than one of the placeholders may eventually receive a value. Figure 4.9 shows a much simpler version that does not deal with these problems (the footnotes to the figure explain the most important omissions). Recall that `disjoin` itself returns a placeholder rather than actually waiting for the value to be known.

This simplified version works by creating a task and its corresponding placeholder (goal) using `Make-Task`. The purpose of this new task is to propagate the value of the appropriate placeholder (the first one that receives a value) out to its own goal. The `code` to be performed when this new task is awakened is supplied as an explicit procedure[4]. This is very similar to the processing of the normal case, except that the task has a list of placeholders (rather than a single placeholder) for which it is waiting and it is enqueued on each of these placeholders.

When any of the placeholders for which this task is waiting receives a value (see Section 4.5), that placeholder is stored in this task's `wake-up value` slot and the task is made available for execution. When the task is activated (using the `run` procedure described in Section 4.2.2) the procedure stored in the `code` slot will be passed this wake-up value. The procedure will propagate it to the placeholder created by the call to `disjoin`, and then call `next` (see Section 4.2.2) to release the processor and find another task.

The use of a task to propagate the value of the appropriate disjunct may seem unusual, but it improves the modularity of the scheduler code itself. This work could have been made part of the `determine!` code, but this organization allows `determine!` to simply awaken tasks in a standard manner. `Determine!` is never required to do any specialized processing on behalf of the tasks it awakens.

### 4.3.3 Task Termination

As mentioned in Section 4.3.1, there is a primitive continuation that denotes the termination of a task. As with any continuation, it receives a value that

[4]The choice of a procedure rather than a continuation here is somewhat arbitrary. It is easier to write as shown. Furthermore, the procedure will relinquish the processor and be itself garbage collected almost instantly so that the garbage collection problem mentioned earlier is not an issue.

is treated as the value for the goal of that task. To make modifications to the system simpler, the handling of this continuation (unlike most primitive continuations) is reflected back into the scheduler as a call to a procedure. The standard procedure is quite simple since it runs as part of the task that is terminating:

```
(define (end-of-computation-handler value)
  (determine! (task.goal (current-task)) value)
  (next))
```

This merely stores the final value into the goal and then locates and activates the next available task. Notice that by simply calling `next` without saving its own state, this task relinquishes the processor and will not be reactivated[5].

## 4.4 Suspending a Task

There are three ways in which a task can relinquish the processor. It can explicitly relinquish the processor using the `reschedule` procedure (mentioned earlier in Section 3.1.4). An interrupt can occur and cause the processor to be relinquished (for example, the initiation of a garbage collection or a clock interrupt). Finally, and most commonly, the task can attempt to `touch` a placeholder that does not yet have a value.

With the utility procedures described earlier it should be easy to see how the first operation is performed:

```
(define (reschedule)
  (release-task next))
```

Garbage collection initiation was described in Section 3.1.1. The actual code to implement it is very similar to that of `pause-everything` discussed in Section 3.1.3. Timer interrupts are handled by calling `reschedule` as part of the interrupt handler.

The remainder of this section is devoted to the third problem, `touch`ing a placeholder. When placeholders were introduced in Section 2.1 it was stated

[5]It is possible, of course, to write code that saves the state of the task prior to ending the computation. The results of such an action can be predicted easily enough, and might even be useful in some cases. But this is so far out of the ordinary as to be almost certainly an error.

that they "can be used to denote an object whose value is not yet known," and that the scheduler is responsible for handling an attempt to `touch` a placeholder which does not yet have a value. This mechanism has two parts, one implemented in the machine underlying the MultiScheme system, and the other as part of the scheduler.

The underlying machine is responsible for both detecting and handling the simple cases related to placeholder objects. There are three different ways in which a placeholder can be initially noticed:

1. The "microcode" for certain operations explicitly `touch`es objects that they manipulate. If the object is not a placeholder, or the placeholder has a value, the microcode will retrieve the correct value and the operation proceeds unimpeded. The operations are all carefully written, however, so that if a placeholder is encountered that does *not* have a value the operation can be stopped and restarted at a later time. The operation gracefully backs out and returns the state of the system to what it was before the operation began. It then performs a call to the scheduler's `await-placeholder` operation. The result is as though the user had written a call to `touch` (which is implemented using `await-placeholder`) of the appropriate placeholder immediately prior to the call to the operation.

2. Many primitive operations normally type-check their arguments for validity before doing any processing. For those operations that do not permit an operand to be a placeholder (for example, arithmetic operations restricted to numeric data types), the normal error handling mechanism of MIT Scheme would cause the primitive to gracefully back out (just as in the previous case) and then invoke an error handling procedure. In MultiScheme, the microcode for these error handlers tests for placeholders and restarts the primitive automatically (i.e. without any form of trap into Scheme code) if the erroneous argument is a placeholder that has a value. If the argument is a placeholder that does not yet have a value then instead of invoking one of the Scheme error handling procedures it invokes the scheduler's `await-placeholder` procedure.

3. Compiled code contains calls to the primitive operation `touch` whenever it must ensure that an object (argument to an in-line coded primitive,

predicate of a conditional, or function to be applied) is not a placeholder and cannot, at compile time, deduce this. `Touch`, which is one of the operations described in case 1, merely tests its operand to see if it is a placeholder. If it is not, the operand is returned. If it is a placeholder with a value, that value is returned. The net result is that an attempt to use a placeholder whose value is already known will proceed unimpeded, but one whose value is still undetermined will cause a call to the `await-placeholder` procedure. (The exact placement of these calls to `touch` is a topic for further investigation. Placing them earlier in the code can frequently make the code more efficient but it reduces the potential for parallelism.)

In each case, the underlying machine handles placeholders that have already received a value but calls the `await-placeholder` procedure to handle placeholders that do not have a value. The job of `await-placeholder`, then, is to save the state of the current task (if it needs to continue running) and enqueue this task on the `waiting-queue` of the placeholder. It then releases the processor by calling `next`.

The code for `await-placeholder` is shown in Figure 4.10. The following description of its operation is keyed to the numbers in the figure.

1. Create a continuation, `me`, that holds the state of the current task. Attempt to lock the placeholder for which the task is waiting.

2. If the placeholder couldn't be locked, just resume the current task. This can occur if the placeholder has received an immutable value prior to reaching this point in the code. The placeholder would be subject to the splicing operation (mentioned in Section 4.1.1) during variable reference or garbage collection. After the lock is acquired this splicing will no longer occur.

3. If the placeholder has a value, unlock the placeholder and resume the current task. This can occur if the placeholder has received a mutable value before reaching this point in the code. In this case, the placeholder is not subject to splicing so it will have been locked, but there is no need to await the arrival of a value.

4. Activate the task that is calculating the value for the placeholder if it is inactive. This arises either because the placeholder was created by

```
(define (await-placeholder placeholder)
  (call-with-current-continuation
   (lambda (me)                              ; (1)
    (With-Placeholder-Locked placeholder
      (lambda (waiting-for-a-placeholder?)
        (cond ((not waiting-for-a-placeholder?)
               (me 'RESUME-COMPUTATION))     ; (2)
              ((determined? placeholder)
               (unlock-placeholder! placeholder)
               (me 'RESUME-COMPUTATION)))    ; (3)
        (activate                            ; (4)
         (placeholder.motivated-task placeholder)
         (lambda (status)
           (or (eq? status 'DELAYED) (eq? status 'PAUSED)))
         'NO-RELEVANT-WAKE-UP-VALUE)
        (store-my-state me (lambda (My-Task) ; (5)
          (set-task.status! My-Task 'WAITING)
          (set-task.waiting-for! My-Task placeholder)
          (add-to-waiting-queue! placeholder My-Task)))))
    (next))))                                ; (6)
```

**See text beginning on page 110 for footnotes.**

Figure 4.10: Code for `Await-Placeholder`

the `delay-policy` (hence the task has status `delayed`), or because the task has been suspended by `pause-everything` (see Section 4.6), and the task has status `paused`.

5. At this point, the task will definitely be releasing control of the processor. The state of the task, `me` from step 1, is saved away in the current task data structure. Mark the task as `waiting` and add it to the `waiting-queue` of this placeholder.

6. Unlock the placeholder (by exiting the `with-placeholder-locked` procedure). Find another task and start executing it. Notice that this call to `next` is *not* executed when control returns to the original task using the `me` continuation created in step 1 (count the parentheses...).

## 4.5 Storing the Value of a Placeholder

Storing a value into a placeholder is a straightforward operation, although the details are somewhat complicated. The essential work is to store the value into the placeholder data structure and activate any tasks that may have been waiting for this value to appear. The detailed code is shown in Figure 4.11. The following notes describe the fine details of its operation. They are geared to the numbers appearing in the figure. Most of the complexity comes from the need to keep the placeholder locked for as short a time period as possible, and the possibility of a race if two tasks attempt to supply values to the placeholder nearly simultaneously.

1. The auxiliary procedure `update-placeholder!` actually makes the changes necessary to the placeholder data structure to reflect the fact that it now has a value.

2. `What-to-do` will contain one of three procedures to be performed after the placeholder is unlocked, in step 7. The three procedures are (a) an error procedure, step 3; (b) awaken the waiting tasks, step 5; and (c) do nothing, step 6.

3. With the placeholder locked, test whether it already has an immutable value. If so, return a procedure that will cause an error in step 7.

```
(define (determine! placeholder value allow-mutations?)
  (define (update-placeholder!)           ; (1)
    (set-task.status! (placeholder.motivated-task placeholder)
                      'DETERMINED)
    (set-placeholder.value! placeholder value)
    (set-placeholder.determined?! placeholder
      (if allow-mutations? 'MUTABLE #t)))
  (define what-to-do                      ; (2)
    (With-Placeholder-Locked placeholder
      (lambda (still-a-placeholder?)
        (cond ((or (not still-a-placeholder?)
                   (immutable? placeholder))
               (lambda ()                 ; (3)
                 (error "Immutable Placeholder" placeholder)))
              ((undetermined? placeholder)
               (let ((waiters
                      (queue-contents
                       (placeholder.waiting-queue placeholder))))
                 (update-placeholder!)    ; (4)
                 (lambda ()               ; (5)
                   (for-each
                    (lambda (task)
                      (activate task
                                (lambda (status)
                                  (or (eq? status 'WAITING)
                                      (eq? status 'DISJOIN)))
                                placeholder))
                    waiters))))
              (else (update-placeholder!) ; (6)
                    (lambda () 'OK))))))
  (what-to-do)                            ; (7)
  value)                                  ; (8)
```

**See text beginning on page 112 for footnotes.**

Figure 4.11: Code for `determine!`

4. If the placeholder did not previously have a value, remember the tasks that are waiting for the value of this placeholder (in `waiters`) and then update the placeholder.

5. Since the placeholder didn't have a value before, in step 7 we must `activate` (see Figure 4.7) each task that is waiting for this placeholder. The activation test permits only tasks that are waiting for a placeholder (status waiting or disjoin) to be awakened.

6. If the placeholder previously had a mutable value, then it can't have a queue of tasks waiting for its value to appear. Thus, in step 7 we don't need to take any special action.

7. Now that the placeholder is unlocked, perform whatever work is necessary.

8. The value returned by `determine!` is (arbitrarily) the value that has been given to the placeholder.

## 4.6 Single Task Interludes

The final, and most complicated, operation supported by the scheduler is the ability to switch from a parallel processing mode where many tasks are simultaneously active to one in which only a single task is active. This operation, embodied in the procedure `pause-everything`, has already been discussed in Section 3.1.3, and the code is shown in Figure 3.1 on page 62. This section presents the underlying support routine that was omitted in that earlier version[6].

As mentioned in the earlier discussion of `pause-everything` its job is to suspend all other tasks on the system. It returns as its value an object that encapsulates these other tasks through a "message passing" interface. The returned object (implemented in MIT Scheme as a procedure of one argument, the message) accepts three messages.

**Any-Tasks?**

Returns a boolean answer of `#t` if there were other tasks running at

[6] A procedure with structure very similar to `pause-everything` but without the elaborate returned object is used to initiate garbage collection.

```
(define (make-returned-object the-queue)
  (lambda (message)
    (cond ((eq? message 'ANY-TASKS?)    ; (1)
           (and (not (eq? the-queue #t))
                (not (eq? the-queue '()))))
          ((eq? message 'RESTART-TASKS) ; (2)
           (if (eq? the-queue #t)
               (error "Attempt to re-use a pause object!")
               (begin
                 (for-each              ; (3)
                  (lambda (task)
                    (activate task
                              (lambda (status) (eq? status 'PAUSED))
                              'NO-RELEVANT-WAKE-UP-VALUE)
                    the-queue))
                 (set! the-queue #t))))
          ((eq? message 'THE-TASKS)     ; (4)
           (if (eq? the-queue #t) '() the-queue))
          (else (error "Pause: unknown message" message)))))
```

**Notes:**

1. Code to handle the Any-Tasks? message.
2. Code to handle the Restart-Tasks message.
3. If this is the first time the restart-tasks message is received, any tasks that are still paused are activated. Notice that the code for `await-placeholder` shown in Figure 4.10 will have activated any of these tasks that were `touch`ed after the call to `pause-everything`. Hence, the test here.
4. Code to handle the The-Tasks message.

Figure 4.12: `Make-Returned-Object`, support for `pause-everything`

the time of the call to `pause-everything` and they have not yet been restarted.

**The-Tasks**
Returns a list of the tasks that were suspended by the call to `pause-everything` provided they have not yet been restarted.

**Restart-Tasks**
Activates the tasks that were suspended by the call to `pause-everything` by releasing them to the underlying task distribution mechanism. It can be called only once.

The procedure `Make-Returned-Object`, shown in Figure 4.12, creates the message accepting object that will be returned by `pause-everything`.

## 4.7 Summary

The scheduler is the "heart" of the MultiScheme system, and is designed as a flexible and extensible mechanism for supporting a variety of experiments. The primary data structures of the scheduler, the placeholder and the task are described in Section 4.1. Using these data structures, a variety of ways of creating, suspending, and managing tasks are described.

Most of the material presented is very much "nuts and bolts engineering" but serves to demonstrate the ease with which the parallel processing support can be described within the existing language framework.

# Chapter 5

# Examples

The earlier chapters have described the structure of the MultiScheme system and a number of the important design choices. This chapter consists of three programs that exploit parallelism and demonstrate several of the unique features of MultiScheme. The purpose of these examples is to introduce some novel ways of using the tools provided by the underlying MultiScheme system.

Section 5.1 concentrates on speculative parallelism — using parallelism to increase the speed of programs that try multiple approaches to solving the same problem. It shows a simple procedure (`first-value`) which can be used as the base for several parallel control structures, and demonstrates its use in the core of a simple "expert system" (rule-based interpreter). Section 5.2 shows the evolution of a serial program for solving the $n$-body problem of computational dynamics into a parallel implementation by the addition of `future` constructs. The judicious addition of `future`s, coupled with measurements and a small amount of rewriting, ultimately yields a program that solves the 8-body problem 6.6 times faster than the serial version on a 16 processor machine. Section 5.3 (along with some details in Appendix B) reviews this solution to the $n$-body problem and provides a reformulation using a system similar to a pipelined computer architecture or a dataflow computation model. Section 5.4 describes how the fork/join, Uniform System[56], and QLambda[21] parallel programming methods can be expressed in MultiScheme.

There is no claim that these programs provide optimal solutions to the initial problems — a carefully crafted Fortran program coupled with an optimizing Fortran compiler can certainly generate faster programs for a sequen-

tial computer than the current interpreter for MultiScheme on a 32-processor Butterfly computer. The programs presented here, however, *do* scale with the size of the problem and the number of processors. Furthermore, there is reason to believe that an optimizing compiler for MultiScheme can be built and that the resulting compiled programs will both perform well and exploit the parallelism specified in the program[1].

## 5.1 Speculative Computation

There are three important ways to use multiple processors of a given type to solve a problem faster than it would be solved with a single processor of the same type:

- Many problems can be divided into loosely coupled smaller problems so that combining the solutions to the smaller problems produces a solution to the original problem. If the coupling is sufficiently loose, the smaller problems can each be assigned to a separate processor and the time required to solve them can be overlapped.

- In a probability-based program (such as a Monte-Carlo method), performing the same operation more often in a given amount of time leads to an improved answer. So for a desired precision, using more processors will lead to an answer in a shorter period of time.

- If many different methods are known for solving a single problem each processor can pursue a different approach to the problem. If we are looking for *any* solution to the problem, then we can select the solution found by the first processor to announce success.

This third approach is known as speculative parallelism. Using the other two approaches we can hope that $n$ processors would run a program $n$ times faster than one processor. By contrast, in speculative computations we hope to have $n$ processors run as fast as the one processor that *always* picks the

[1]The hope for an optimizing compiler is more than sheer speculation. An optimizing compiler for Scheme[36] has already been implemented. Work on a similar compiler for MIT Scheme is underway. Needless to say, since MultiScheme forms the heart of the BBN Butterfly Lisp product, BBN is pursuing compilation technology in the parallel-processor domain with considerable vigor.

best method to solve the problem at hand. Clearly, this approach is justified only if there is no simple way of deciding, in advance, which approach works best on the particular problem at hand.

### 5.1.1 Parallel Control in a Rule Interpreter

Speculative parallelism may prove useful in a variety of cases. Many "AI search" problems can be conveniently expressed using speculation; in fact, this was one factor motivating the design of the Connection Machine[31]. The example chosen here, a rule interpreter that forms the core of an expert system, exhibits one such search. The purpose of the example is to demonstrate how speculative computations can be expressed in MultiScheme. The example does not address other important issues such as the ability to analyze the rule base to find a better control structure than the one described here.

The example is based on a program originally developed for teaching an introductory undergraduate course in signal processing[1]. It is a rule-based expression simplifier supplying enough power to handle a set of rules for simple symbolic algebra. Like most rule interpreters it consists of a pattern matcher, a rule firing mechanism, and a control structure component. Both the pattern matcher and the control structure can utilize speculative parallelism, but for simplicity this example deals only with the control structure.

The rule interpreter has an inner loop that takes an expression and tries to match it against all of the rules. If a rule matches the expression, the rule firing mechanism is invoked to produce a modified (presumably simpler) expression. If no rule matches, then the expression is already in its simplest form. The problem we are attempting to solve, then, is converting the sequential control structure of "multiple solution techniques augmented with a default result" into a corresponding speculative (parallel) control structure. In doing so, we will assume that rules may be fired *in any order* that terminates and will still arrive at a correct final result — the only restriction is that we will not "give up" on simplifying an expression until all rules have failed to simplify it. This assumption is *not* valid in general for rule sets and poses an important practical limitation to the solution described here.

MultiScheme provides two forms of support for speculative programming, and both are used in solving this problem: the procedure `disjoin` introduced in Section 2.5, and the garbage collection of useless tasks discussed in Sec-

tion 3.5. The former is used as the most primitive operation for requesting speculative computation — it takes a set of placeholders and returns a placeholder that eventually receives the value of the first one to be given a value. The latter is invoked implicitly rather than explicitly, and serves to free up processors that are pursuing approaches to an already-solved problem.

Before proceeding, a little utility procedure will come in handy. The procedure `disjoin` is a nice primitive mechanism, but for this example a slightly different interface will prove more convenient. Rather than receive a placeholder for the first value to be computed, we will want to receive the value itself. Furthermore, we will be creating a list of placeholders separately from the call to `disjoin` so we need a procedure that accepts a list of placeholders (`disjoin` expects the placeholders as arguments). This is easily expressed:

```
(define (await-first list-of-placeholders)
  (touch               ; Wait for a value to appear
   (apply disjoin      ; "Spread" the arguments
          list-of-placeholders)))
```

With only this to help us, it is slightly tricky to express the notion of a default value to be used only when all other approaches fail. If we merely use `disjoin` to try all of the approaches we risk getting a premature answer from the default method, causing us to abandon the other techniques. On the other hand, if we don't include the default value in the set of methods tried, we will never succeed in simplifying an expression since we won't be able to simplify the most primitive components. The gist of the solution is to provide an explicit indication that a rule has failed. Then, rather than just wait for `disjoin` to return an answer, we look at the answer we get back. If it is a failure indication we wait again, until either a successful result is returned or no other rules are available — only in this last case do we supply the default value.

To implement this idea, we begin by introducing the special value used to represent the failure of a rule (`failure-tag`), and a procedure to test for a not-yet-failed placeholder:

```
(define failure-tag (list 'failure))

(define (not-failed object)
  (or (placeholder? object)             ; No value yet
      (not (eq? object failure-tag))))  ; Value, not failure
```

Using these it is easy enough to write the procedure that waits for a non-failure answer or returns the default value:

```
(define (first-value placeholders default-value)
  (if (null? placeholders)
      default-value                    ; No alternative
      (let ((result (await-first placeholders)))
        (if (eq? failure-tag result)
            (first-value               ; Failed: try again
             (filter not-failed placeholders) default-value)
            result))))                 ; Success
```

This mechanism is sufficient to solve our problem[2]. The rule system discussed in [1] takes a set of rules and produces a procedure for simplifying expressions based on these rules. It consists of three major procedures, corresponding to the pattern matcher, rule firing mechanism, and control structure. For our purposes, the first two components can be treated as "black boxes" whose internal operation is not relevant to us. The pattern matcher and rule firing components have the following interfaces (these differ slightly from the version in [1]):

`(matcher rule expression fail succeed)`
: Matches the `rule` against the `expression`. If there is no way for them to match, `fail` is called with no arguments. Otherwise `succeed` is called with a dictionary (containing the values assigned by the successful match to variables in the `pattern`) and a procedure that can be used to resume the matching process if the values in the dictionary are unacceptable (`fire`, below, is the base for a typical success procedure as shown in Figure 5.1).

`(fire rule dictionary simplifier)`
: Fires the specified `rule` using `dictionary` to guide construction of a new expression. The resulting new expression is simplified using `simplifier` and returned.

[2]The version of `first-value` shown here has order $n^2$ performance in the number of placeholders, because of the call to `filter`. More efficient versions could be constructed using, for example, a counter of the number of placeholders whose value is not yet known.

```
(define (simplifier the-rules)
  (define simplify-exp                        ; (1)
    (memorize
     (lambda (exp)
       (try-rules (if (compound? exp)
                      (map simplify-exp exp)
                      exp)))))
  (define (try-rules exp)                     ; (2)
    (define (match-rule rule rule-fail)       ; (3)
      (matcher rule exp rule-fail
               (lambda (dictionary keep-matching)
                 (fire rule dictionary simplify-exp))))
    (define (scan rules)                      ; (4)
      (if (null? rules)
          exp
          (match-rule (car rules)
                      (lambda () (scan (cdr rules))))))
    (scan the-rules))
  simplify-exp)
```

**Notes:**

1. `Simplify-exp` simplifies all the components of an expression, then tries all of the rules on the result. `Memorize` implements the standard dynamic programming trick of consulting a table of known simplifications before attempting the recursive tree walk shown here.
2. `Try-rules` tries to match and fire each rule against the incoming expression.
3. `Match-rule` tries to match and fire a single rule against the incoming expression.
4. `Scan` sequentially tries to match and fire each rule against the incoming expression.

Figure 5.1: Recursive Control Structure

The control component — the lone component we are setting out to modify — consists of the routine `simplifier` shown in Figure 5.1. This simplifies any sub-expressions, then tries the rules sequentially until one fires. The firing process then creates a replacement expression and (recursively) simplifies it. The only part of `simplifier` that we will need to change is `scan`, which in the original version (reproduced below) sequentially scans the rules:

```
(define (scan rules)
  (if (null? rules)
      exp
      (match-rule (car rules)
                  (lambda () (scan (cdr rules))))))
```

In this sequential version, `scan` tries the first rule — using `match-rule` — and specifies that if this rule fails to fire the process should repeat with the remaining rules. Instead of this, we want to use the `first-value` procedure we defined above by creating a task for each of the rules. In order to do this, we must arrange matters so that when a rule fails to match it causes the task that is running to return the failure signal:

```
(define (scan rules)
  (define workers                        ; List of placeholders
    (map (lambda (rule)                  ; ...one per rule
           (FUTURE (match-rule rule
                               (lambda () failure-tag))))
         rules))
  (first-value workers exp))
```

Introducing this form of speculative parallelism into the rule interpreter was quite simple. But the same technique allows more complicated sequential control structures to be converted to parallel speculation as well. By only a small change to `first-value` it can be converted into the basic support needed for handling any number of priority-based solution methods — instead of returning a specific value, it can call a procedure that starts the next lower priority set of methods and calls `first-value` to wait for all of these methods to complete, and so forth. Since there need be no restriction on the mechanism used to generate the next set of methods to be tried, quite complicated systems of dependencies can be expressed using the same general technique.

### 5.1.2 Speculative Computation: Summary

As an example, this rule system was run using two sets of rules — one for symbolic algebra and one for derivatives — and these two rule sets were passed as arguments to the `simplifier` procedure to create two simplification functions. These two functions were themselves composed to produce one large procedure, `psimp`, that first simplifies its input using the derivative rules and then simplifies the result using the algebra rules. The algebraic rule set consisted of twenty rules including constant elimination, commutativity, distributivity, and conversion of products into exponents. The derivative rule set had eight rules.

The `psimp` procedure was used to find the derivative of a standard cubic equation ($ax^3 + bx^2 + cx + d$ with respect to $x$). Using ten processors, the derivative calculation required half the time of the same calculation on a single processor. One property of the rule sets (unfortunate for this example) is that a single expression rarely matches the pattern of more than one rule. As a result, most tasks terminate very quickly by failing to match, so the garbage collection of tasks played no noticeable role. The ability of the garbage collector to remove useless threads of computation remains important in general, although this rule set and parallel control structure do not show it to advantage.

The particular choice of speculative control structure used here is only one of a range that deserves more exploration. An interesting property of this structure is that it allows the parallel system to arrive at answers even in cases where a sequential implementation would fail because of an infinite loop in the rule set. If, for example, we include commutativity as one of rules of algebra it is essential in the serial system that this rule be chosen only once for any given expression — otherwise we might start with $a + b$, convert it to $b + a$, then convert it back to $a + b$ and so forth[3]. By contrast, the parallel control structure here *will* terminate even in this case — and then the garbage collector would remove the remaining tasks. Without performing such an experiment, however, it is not clear just how well such a rule set would perform.

This experiment, while demonstrating a factor of 2 performance improve-

[3]The rule base used in the serial program of [1] solves this problem by the use of restrictions on the applicability of the rule. A lexical ordering on expressions is exploited to guarantee that the rule is fired only once for a given expression.

ment, leaves open a number of questions related to the use of speculative parallelism in rule-based systems. But, as mentioned at the outset, answering these questions was not the purpose of the example. Far more importantly, this example has shown that the underlying `disjoin` operation does provide the starting point for building systems that exploit parallelism to explore alternative methods for solving a single problem. Using this operation it was a straightforward task to modify the serial control structure of an existing system into one more suited for a parallel processor.

## 5.2 Object Interaction Simulator

One class of problem in which parallel computation can be expected to provide significant performance improvement is the simulation of systems composed of interacting components. The plan is to utilize the multiple computation units so that each simulates a single object or group of objects, periodically updating its own state based on the state of all of the other objects. This kind of simulation is quite common, and has been a target of investigation in the parallel computing community for some time. One classic example of this kind of problem is the "n-body problem" of classical physics:

**Given:** The mass ($m_i$), initial position ($\vec{x_i}(0)$), and initial velocity ($\vec{v_i}(0)$) of a group of $n$ particles.

**Given:** The force law ($\vec{F}(\vec{x_i}, \vec{x_j})$) governing the interaction of a pair of particles.

**Predict:** The position ($\vec{x_i}(t)$) and velocity ($\vec{v_i}(t)$) of the particles into the future (possibly ignoring collisions).

The problem is easily formalized, but the computational task for even relatively small systems is immense. Because the interparticle force calculations require $n^2$ steps a serial processor requires time proportional to $n^2$. Since the problem inherently divides into $n$ independent parts (one per particle), it is clear that a program can be coded that will solve the problem in time proportional to $n$ if there are $n$ processors available.

The description and analysis of this single example is quite long (it spans both Sections 5.2 and 5.3) and provides information at a number of levels. At

one level it provides a largely accurate chronology of the conversion of a working serial program into a parallel program that achieves the desired speed increase. At another level, the measurements that motivated each step in the process are themselves aided by modifications to the MultiScheme scheduler that provided the data needed to graphically present the parallelism profile of the program, illustrating the value of a flexible scheduler implementation in MultiScheme. At yet another level, the changes from each version of the program to the next demonstrate uses of the constructs described earlier. And finally, the last transformation (described in Section 5.3) of the program demonstrates a technique that has long been the strong point of Lisp languages: the ability to construct a system that provides the external behavior of a completely different model of computation based on primitives that exist within (in this case) MultiScheme.

### 5.2.1 Simple Sequential Implementation

To begin implementing a parallel program that solves the n-body problem, let us first study an elegant sequential solution. This program is adapted from a version written by Sussman, based on his work on orbital mechanics[3]. The program is structured around the notion of a system of particles and operations on such systems. It operates by constructing differential systems representing changes to a system over some small time interval. Just as a system consists of particles, a differential system consists of differential particles whose state is computed based on the state of the original particles and a particle interaction (force) law. Using a Runge-Kutta integration technique to combine the initial system with related differential systems the program predicts the position and velocity of the particles into the future. The solution makes use of a library of general-purpose mathematical routines, shown in Figure 5.2.

A system consists of a simulated time, and the set (list) of particles in the system. The particles have their own state information, represented for simplicity as a vector of quantities. Each particle has a name, primarily for documentation purposes. It also has a mass (a scalar quantity) and two vectors representing its position and velocity. Particles are constructed using the procedure `make-particle`, systems by `make-system`. Selectors are `time` and `particles` for a system, and `name`, `mass`, `position` and `velocity` for particles.

| Name | Description |
|---|---|
| `(group op ll)` | Re-groups objects from `ll`, a list of sets of objects, into sets based on a selection criterion (`op`). Each output set consists of those input objects for which `op` returns the same value. |
| `(reduce op vals)` | Combines all of the `vals` using the associative operator `op`. |
| Scalar Operations | `cube`, `square` and `sqrt` (square root). |
| `(scale n)` | Returns the procedure that multiplies a vector by `n` (a number). |
| `(scale-by n v)` | Scale vector `v` by amount `n`. More convenient to write than `((scale n) v)`. |
| Vector Operations | `add` and `sub`. |
| `zero-vector` | A pre-computed three dimensional vector of zeros. |

Figure 5.2: A Simple Mathematics Library

There are two operations that can be performed on particles. `Scale-Particle` (used to construct differential systems) multiplies the state information of a particle by a scalar factor (representing the time step). `Add-Particles` adds together the state information from a number of particles to produce the state information for a new particle. See Figure 5.3.

Finally, there are two operations on systems of particles. These correspond to the particle operations: `scale-system` multiplies a system by a constant factor (time step), and `add-systems` sums a number of systems to produce a resultant system. See Figure 5.4.

The most complicated step in our simulation is to write a procedure that will construct a "differential system" from an initial system and a particle interaction force law (see Figure 5.5). A differential system consists of a differential of time ($dt/dt$, clearly always 1), and a set of differential particles. Just as a particle consists of a mass, position, and velocity, a differential particle consists of a differential of mass ($dm/dt$, always 0 since there is assumed to be no loss of mass over time), a differential of position ($dx/dt$, which is of course the velocity of the original particle), and a differential

```
(define (scale-particle factor)
  (define factor* (scale factor))          ; vector scale
  (lambda (particle)
    (make-particle
     (name particle)                       ; name
     (* factor (mass particle))            ; mass
     (factor* (position particle))         ; position
     (factor* (velocity particle)))))      ; velocity

(define (add-particles . particles)
  (make-particle
   (name (car particles))                  ; name (same for all)
   (reduce + (map mass particles))         ; mass
   (reduce add (map position particles))   ; position
   (reduce add (map velocity particles)))) ; velocity
```

Figure 5.3: Operations on Particles (serial)

```
(define (scale-system scale-factor)
  (define sp (scale-particle scale-factor))
  (lambda (system)
    (make-system (* scale-factor (time system))
                 (map sp (particles system)))))

(define (add-systems . systems)
  (make-system (reduce + (map time systems))
               (map (lambda (bunch) (reduce add-particles bunch))
                    (group name (map particles systems)))))
```

Figure 5.4: Operations on Systems of Particles (serial)

of velocity ($dv/dt$, or acceleration). This last component is the core of the computation. Acceleration is just the sum of the inter-particle forces acting on a given particle (divided by the mass of the particle), computed using the specified `force-law`.

Given this framework, we can describe the overall n-body computation. All that needs to be done is to select an integration procedure sufficiently accurate for our purposes. One could, for example, merely use the differential system to produce a linear prediction of the next state. This would suffice for either a very small value of $dt$ or a low accuracy calculation. There are many more sophisticated (and accurate) techniques that have been developed for numerical integration, and for this example a $4^{th}$ order Runge-Kutta integrator[4] was used. The code for this specific integration technique (`runge-kutta-4`) along with the procedure that produces a stream of future values of an initial system (`integrate-system`) are shown in Figure 5.6.

The serial program is now complete. It can be tested out by creating a system of particles and predicting the state of the system for three time steps. Using the 9-particle solar system[5] as an example, we have (time in days, distance in AU, mass in units of the mass of the sun, center of mass coordinate system):

```
(define sun
  (make-particle
   'sun+mercury (+ 1 (/ 1 6000000))          ; name and mass
   (-4.09433e-3 -5.62963e-3 -2.2635e-3)      ; position
   (6.666e-6 -5.75161e-6 -2.6353e-6)))       ; velocity
...
(define system (make-system 0 (list sun+mercury ...)))
(define system-stream (integrate-system system 0.5))
(nth-stream system-stream 3) ⇝
  (1.5                                       ; time
   (SUN+MERCURY                              ; name
    1.                                       ; mass
    (-4.08432e-3 -5.63825e-3 -2.26745e-3)    ; position
```

[4] A Runge-Kutta integrator works by using interpolation of the system based on derivatives of the system state up to some chosen number. A $4th$ order integrator, thus, uses the first through fourth derivatives of the system. It is a generalization of the more familiar (but less accurate) trapezoidal technique for integration.

[5] This treats the sun and Mercury as a single particle, a standard simplification used in astrophysics.

```
(define (particle-force force-law)
  (define (accelerations bodies)           ; (1)
    (define (my-acceleration me)           ; (2)
      (reduce add
              (map (lambda (other) (force-law me other))
                   bodies)))
    (map my-acceleration bodies))
  (lambda (system-state)                   ; (3)
    (make-system 1                         ; dt/dt
      (map (lambda (particle acceleration)
             (make-particle
              (name particle)
              0                            ; dm/dt
              (velocity particle)          ; dx/dt
              acceleration))               ; dv/dt
           (particles system-state)
           (accelerations (particles system-state))))))

(define (gravitation me it)                ; (4)
  (if (eq? me it)
      zero-vector
      (let ((dx (sub (position me) (position it))))
        (let ((rcube (cube (sqrt (reduce + (map square dx))))))
          (scale-by (/ (* -G (mass it)) rcube) dx)))))
```

**Notes:**

1. Return a list of the accelerations of all $n$ particles.
2. Calculate the acceleration of one particle.
3. Procedure returned from `Particle-Force`, that takes a system at time $t$ and produces the differential system $dt$. See text beginning on page 127.
4. A sample force law returning the acceleration of `me` as a result of gravitational interaction with `it`.

Figure 5.5: Differential System Generator

```
(define (runge-kutta-4 f dt)                ; (1)
  (define dt* (scale-system dt))
  (define 2* (scale-system 2))
  (define 1/2* (scale-system (/ 1 2)))
  (define 1/6* (scale-system (/ 1 6)))
  (lambda (system)                          ; (2)
    (define k0 (dt* (f system)))
    (define k1 (dt* (f (add-systems system (1/2* k0)))))
    (define k2 (dt* (f (add-systems system (1/2* k1)))))
    (define k3 (dt* (f (add-systems system k2))))
    (add-systems system
      (1/6* (add-systems k0 (2* k1) (2* k2) k3)))))

(define (integrate-system initial-state dt) ; (3)
  (let ((integrator (runge-kutta-4 (particle-force gravitation) dt)))
    (define (next state)
      (cons state
            (delay (next (integrator state)))))
    (next initial-state)))
```

**Notes:**

1. Produce a procedure that takes an input system at time $t$ and produces a system at time $t + dt$ using a $4^{th}$ order Runge-Kutta integrator.
2. This is the actual integration procedure returned from `Runge-Kutta-4`.
3. Produce a stream of values representing the state of the system beginning with the `initial-state` and separated by intervals of $dt$.

Figure 5.6: Integration Procedure (serial)

```
    ( 6.67693e-6 -5.73931e-6 -2.63031e-6)))) ; velocity
    ...))
```

### 5.2.2 Parallel Particle Interaction

The system as described above works well, and it is time to speed it up with the judicious addition of `future`. Because the computation is asymptotically dominated by the time required to compute particle interactions, it seems logical to rewrite the `particle-force` procedure of Figure 5.5. This is easily done, by changing only the procedure `my-acceleration` within `particle-force`:

```
(define (my-acceleration me)
  (FUTURE
   (reduce add (map (lambda (other) (force-law me other))
                    bodies))))
```

Now, in order to perform the inter-particle force calculations the program creates a task for each particle. That task calculates the acceleration of its associated particle based on the state information of all of the particles. Installing this change and running the program yields precisely the same results as before, but faster.

Our initial expectation, of course, was that for a 9-particle system we should see a factor of 9 improvement (since the experiment was run on a machine with more than 9 processors). Unfortunately, the program runs only a factor of 1.8 faster than the original — the kind of result often encountered when adding parallelism to a serial program. At this point, the project shifts into a "performance debugging" phase, not unlike the ordinary debugging phase of a program. At this stage the ability to visualize the parallelism being attained by the program is essential.

Fortunately, MultiScheme running on the Butterfly computer had already been augmented by Seth Steinberg (of the BBN Butterfly Lisp project) to provide a good deal of help with this task. He modified the scheduler described in Chapter 4 to send a monitoring packet out whenever a task is created, suspended, or completes its processing. These packets are collected and can be displayed (unfortunately only after a run is completed) using a set of tools that are part of the "Butterfly Lisp User Interface" (BLUI) developed by Laura Bagnall of the BBN project. A glance at these displays (see Figure 5.7 and the accompanying notes) reveals a consistent pattern:

- A short stretch of 9-fold parallel computation.
- A medium length stretch of serial computation.
- The previous two items repeat a total of four times.
- A long period of serial computation.

Clearly, the 9-fold parallel unit that repeats four times arises from the four calls to `f` within the procedure `runge-kutta-4`. The medium length serial computation between them corresponds to the work of scaling and adding

**Key to Performance Diagrams:**
Performance diagrams consist of two regions, with a common time axis running from left to right. The lower region is a graph of processor activity showing the number of processors actually engaged in running tasks. The upper portion is a detailed accounting of all tasks in the system during the time of interest. Each task is represented by a horizontal bar which, in enlarged drawings, will be seen to consist of segments in three different patterns: fully black indicating a running task, dark grey indicating a task that is ready to run but for which no processor is available, and white (very light grey) indicating a task waiting for a placeholder's value to become available. The vertical arrows indicate either a task being spawned (arrow pointing down from the parent, which is black, to the child), a task waiting for the value of another's goal placeholder (arrow from the task that is waiting — and thus changing from black to white — to the task that will deliver the value), or a `determine!` operation (typically terminating a horizontal bar).

The range of times shown on these performance diagrams is not the same on every diagram.

Figure 5.7: Parallelism in Particle Force

the resulting system. The final long serial computation corresponds to the final call to `add-systems`.

These results were not overly surprising, although there seemed to be no reason for the final call to `add-systems` to be so very much (more than a factor of 4) longer than the earlier calls. Re-examining the procedure in Figure 5.4 shows that one possible problem is in the `group` procedure of the math library. Indeed, examining that code (not included here) reveals that it employs an algorithm with a worst case $n^2$ performance. A minimal speed increase can be found by rewriting `group` to itself operate in parallel, but a more dramatic increase results from understanding what purpose it is serving.

In general, when two systems of particles are to be added together, there is no guarantee that the system contains precisely the same particles. Furthermore, we have implemented systems using unsorted lists to represent the sets of particles. As a result, there is no guarantee that even if the systems contain the same particles they will appear in the same order in each system. Thus, a general purpose procedure for adding together systems of particles must use some operation like `group` to match corresponding particles in the systems. In the problem at hand, however, the particles in each system are in fact always both the same and in the same order. Thus, rewriting `add-systems` with this assumption will save considerable time during the computation. At the same time, we can operate on each set of corresponding particles in parallel:

```
(define (add-systems . systems)
  (define (transposer particle-set)
    (define (compute particles) (reduce add-particles particles))
    (if (null? (caar particle-set))
        '()
        (cons (FUTURE (compute (map car particle-set)))
              (FUTURE (transposer (map cdr particle-set))))))
  (make-system (reduce + (map time systems))
               (transposer (map particles systems))))
```

With this new version of `add-systems` performance is significantly better, but only by a factor of 3.7 over the original. (See Figure 5.8 for a close-up of one of the four system calculations. This figure corresponds to the region between the points marked A and B in Figure 5.7.) The next advance

This figure corresponds to the area between the points marked A and B in Figure 5.7. See the notes to Figure 5.7 for a general description of performance diagrams. In this figure, the numbers visible in the horizontal bars show the number of the processor running the task.

Figure 5.8: Parallelism in Adding Systems

comes from realizing that the procedure `scale-system` can be improved by processing the particles in parallel:

```
(define (scale-system scale-factor)
  (define sp (scale-particle scale-factor))
  (define (scale-it x) (FUTURE (sp x)))
  (lambda (system)
    (make-system (* scale-factor (time system))
                 (map scale-it (particles system)))))
```

See the notes to Figure 5.7 for a general description of performance diagrams.

Figure 5.9: Parallelism in Scaling Systems

This was not, perhaps, obvious at first since the work entailed in scaling a particle seems small — small enough that it isn't likely to compensate for the work necessary to create a new task. On reconsideration, however, the cost of the two vector operations involved in scaling a particle (one for position and the other for velocity) is sufficient to overshadow this cost. With

This figure is an enlargement of the area between the points marked A and B in Figure 5.9 See the notes to Figure 5.7 for a general description of performance diagrams. Unlike the earlier examples, the amount of parallelism demanded exceeded the 16 processors available. Hence some of the bars are grey, indicating queuing of tasks waiting for processors.

Figure 5.10: Details of Scaling Systems

this change we begin our final approach to the speed-up we had originally expected. The figure now stands at a 6.6-fold increase over the original serial version. Further work along the same lines doesn't seem warranted, since examination of the activity graphs (see Figure 5.9 for the total computation, and an enlargement in Figure 5.10) indicates no significant areas of serial computation — we are seeing the overhead of creating the tasks on demand combined with small amounts of truly serial computation (such as the sequencing of execution of the `defines` in `runge-kutta-4`).

### 5.2.3 Particle Interaction: Summary

As we have seen, the simple methodology of adding `future` constructs to a straightforward serial solution of the n-body (particle interaction) problem does, indeed, succeed in producing a faster program by exploiting the inherent parallelism. Unfortunately, intuition is not a perfect guide to where this parallelism can be used to best advantage. A set of tools for visualizing where the parallelism is being exploited and (more importantly) where computation remains serial is almost essential to the "performance debugging" task.

But this simple methodology has some drawbacks. This example has demonstrated two of them. The first, perhaps a failing of the author rather than the methodology, comes from a poor intuition. Even knowing the cost of a `future` and applying good engineering trade-off concepts, the initial parallel version missed a significant use of parallelism. This was corrected only by using the visualization tools to actually measure the time spent during the `scale-particle` calculations. This kind of engineering estimation can be improved, perhaps, by the use of automated program analysis tools. This approach has been explored to a limited degree by Sharon Gray[23].

The second drawback, however, is harder to correct. After some effort a general purpose utility procedure was found to use a large amount of serial computation to solve a very simple problem. Just as in a serial program, recoding the general algorithm using specialized knowledge of the application produces a significant improvement. Unlike the first drawback, attempting to automate this process brings us dangerously close to the automatic programming area. In fact, the effort to discover and classify algorithms that exploit parallelism is a major area of research in the theory of computation. While the actual `future` construct itself has nothing whatsoever to do with

the difficulty, its very simplicity encourages a style of operation where this kind of algorithmic exploration is easily overlooked.

## 5.3 Parallel Pipelining

The solutions to the n-body problem described in Section 5.2 were arrived at by incremental changes to a working sequential solution. This section takes a different approach to solving the same problem. First a new general framework for expressing parallel programs is created. Then this specific problem is solved within that framework. This section will serve to highlight two important points. First, one of the most powerful aspects of MultiScheme is the ease with which alternative programming models can be expressed and the resulting ability to experiment with new techniques. Second, this section introduces some uses of placeholders and tasks that are less evident than the ones in earlier examples.

The speed increase of a factor of 6.6 out of a possible 9.0 that was achieved in Section 5.2 seems about the maximum we can reach using the simple technique of adding futures to a sequential program. And yet with one final, less obvious change, we can reach a speed-up of 7.9 over the serial version. This final change occurs in `runge-kutta-4`. Notice that the two intermediate constants `k1` and `k2` appear in the final line scaled by 2. This scaling computation can begin earlier, providing some additional speed, by merely introducing two new intermediate variables:

```
(define (runge-kutta-4 f dt)
  ... definitions of dt*, 2*, 1/2*, and 1/6* as in Figure 5.6 ...
  (lambda (y)
    (define k0 (dt* (f y)))
    (define k1 (dt* (f (add-systems y (1/2* k0)))))
    (define k1*2 (FUTURE (2* k1)))      ; ***
    (define k2 (dt* (f (add-systems y (1/2* k1)))))
    (define k2*2 (FUTURE (2* k2)))      ; ***
    (define k3 (dt* (f (add-systems y k2))))
    (add-systems y (1/6* (add-systems k0 k1*2 k2*2 k3))))) ; ***
```

This new way of using parallelism to speed up the program is reminiscent of the arguments put forth in favor of the dataflow model of computing. In fact, this problem seems ideally suited for a data flow computation model. With

this idea in mind, we can explore an alternative implementation technique for this problem that *maintains the same modularity* in the solution.

The new solution to the n-body problem takes a sort of vector pipeline approach to the problem. Imagine a pipeline that consists of five kinds of building blocks:

1. Straight segments where computations on individual elements of the vector occur.

2. Forks where the vector proceeds in multiple directions simultaneously.

3. Joins where several vectors originating from one initial input reunite and are combined to form an output vector.

4. Interactions where each element of a vector does a computation based on the state of all of the elements of the vector as they arrive at this point in the pipe.

5. Outputs where all of the elements of a vector are gathered together and bundled back into Scheme format.

To use this kind of pipeline in solving the n-body problem it must be able to perform three operations: scaling a single vector (`scale-system`), adding several vectors (`add-systems`), and interparticle force calculations (`particle-force`). Scaling a vector can be accomplished by examining only the individual elements of the vector, and so a straight pipeline segment can be used. Adding several vectors is also done component by component, but requires several different vectors and hence corresponds to a join in the pipeline. Finally, interparticle force calculations require that each component of a vector know the state of all the other components of the vector and, thus, is implemented using a pipeline interaction segment.

The pipeline solution to the n-body problem will be presented in three parts. First, in Section 5.3.1, the procedures used to construct a pipeline are described (the details of the implementation are presented in Appendix B). Using these procedures we can then describe the pipeline based implementation of the procedures `scale-system`, `add-systems`, `particle-force`, and `runge-kutta-4` used in the original solution to the n-body problem. But in order to use the pipeline, we must also be able to create the objects that propagate through the pipeline. This is a rather complicated procedure,

and is explained in Section 5.3.2. Finally, we can combine these operations together to construct the remaining major part of the n-body program, `integrate-system`. The implementation of this procedure is described in Section 5.3.3, along with some observations about its behavior compared to that of the previous version.

### 5.3.1 Building the Pipeline

There are any number of ways in which a pipeline could be constructed to solve the n-body problem. The method described here is motivated by two considerations. The first is that the overall modularity of the original solution should not be changed. That is, given the choice of construction technique, the code for `runge-kutta-4` and `integrate-system` should be nearly unchanged. The second consideration is that the method should be clearly applicable to many other problems as well. In fact, if the effort is not too great, it would be nice to create a convenient abstraction for constructing pipelines with arbitrary structure and operations.

The particular abstraction described below seems to meet these requirements quite well. Unfortunately, the amount of detail required to explain the implementation is large enough that it merely distracts attention from the more important points that this example is intended to demonstrate. The implementation, therefore, is described separately in Appendix B. What follows is a description of the overall nature of the pipeline and the construction operations. The description provided here should be sufficient for the purposes of understanding the remainder of this example.

There are two distinct ways to visualize the pipeline, and both are essential to understanding its operation. The first way, which corresponds to the "view from outside" is that it has an input port, one or more output ports, and an overall physical interconnection. Specially constructed vectors of objects (called pipe vectors) can be presented at the input port. They progress through the pipeline and the objects in them are modified by operational portions of the pipe (straight segments and interaction segments), and ultimately arrive at output ports. The output ports appear to be a list of the pipe vectors that have arrived at the port in the order of their arrival at this output port. Naturally, these output lists will appear to become longer and longer as more and more pipe vectors arrive at the output port — in fact, an attempt to examine the entire list will always fail to terminate (take an

infinite amount of time) since a new pipe vector might arrive at any time.

The complementary view, "from inside," is that a pipe vector of objects arrives at the input port. Each object in the pipe vector progresses independently through the pipeline — the pipe vector is largely a polite fiction maintained for the benefit of outsiders. From the object's point of view the pipeline is nothing but a list of messages that tell it how to mutate its own state information as it passes through the pipe. The total existence of a pipe vector consists only in the fact that each object knows the pipe vector to which it belongs and each pipe vector knows how many objects compose it and to which pipeline input port it has been assigned. Section 5.3.2 describes the messages that each object must be capable of handling, and the implementation of the objects themselves. For the purposes of this section, we need only know that operations to be performed on vectors passing through the pipe are actually received by the individual objects within the pipe vector. Thus, for any given pipe vector, it is critical that all of the objects in that vector handle the same set of messages. It is these messages that the pipeline construction procedures allow the user to specify, and that consequently tailor the pipeline to a particular application.

The work of constructing the pipeline is supported by a package of procedures whose implementation is discussed in Appendix B. The following paragraphs document the interface to this package, but the overall game plan must be explained first. A pipeline is constructed by making an input port (using `make-input`). This is then extended step by step: at each step a procedure takes the previous end of the pipeline, creates a new pipe segment, joins them together, and returns the newly created segment (the new end of the pipe). Outputs are created in a way similar to other segments (by extending an existing pipe segment using `add-output`), although the result is not itself a pipe segment. You can imagine the pipeline "growing" from its input port toward its output ports with loose ends being extended at each step, leading to a new set of loose ends until finally these ends are "capped" with an output port.

`(make-input)`

> Creates a pipe input segment. This is the way in which a pipeline construction project normally begins. Vectors are injected into the pipeline at this segment using `start-pipeline`, discussed in Section 5.3.2.

`(extend p m)`

Given an existing pipe segment (`p`) that is currently a "dangling end" of the pipeline, `extend` creates a new straight pipe segment that will cause operation `m` to be applied to vectors passing through it. This new pipe segment is now a dangling end, and is returned to the caller. The messages that can be used for `m` are defined by the objects that comprise the pipe vectors passing through this segment of pipe.

`(fork n p)`

Given an existing (dangling end) pipe segment (`p`), return a list of `n` new dangling ends for the pipeline. As a pipe vector exits segment `p` it is duplicated `n` times, and one copy of the pipe vector will continue along each of these output pipes.

`(join m ps)`

Given a set of existing pipe segments (dangling ends, `ps`), create a new dangling end segment. Each of the incoming pipe segments is extended into the new segment. As pipe vectors arrive from the input segments they are combined using operation `m` to produce a single outgoing pipe vector that proceeds down the newly created pipe segment. As in `extend`, the operations that may be used are defined by the objects which comprise the pipe vectors arriving at this pipe segment.

`(add-interactor p m)`

Similar to the operation of `extend`, except that the object arriving at this pipe segment will receive a record of the state of all the objects in the same pipe vector as they arrived at this point. This permits each object in a pipe vector to compute a new state based on the states of all the other objects in the vector. In this example, `add-interactor` is used for force calculations.

`(add-output p)`

Extend an incoming dangling end by creating an output port to "cap" that segment. The value returned by `add-output` is a list of all of the pipe vectors that arrive at the output port. In fact, this value is at all times a list whose final element is an undetermined placeholder. It initially contains no elements at all (i.e. it is itself the undetermined placeholder). As each pipe vector arrives at the output port the undetermined placeholder is determined to be a `cons` of the new output

vector and a new undetermined placeholder.

Using these operations we can define our own specialized pipeline construction procedures that correspond to components of the earlier solution. The first one, `add-systems`, is quite simple once we realize that it is used only in the process of pipeline construction. Just as in the earlier solution it received an entire system as input, in this version it receives existing "dangling ends" of the pipeline. Just as it used to return a new system, now it returns a new dangling end of the pipeline:

```
(define (add-systems . pipes) (join 'add pipes))
```

The next two construction operations are `scale-system` and `particle-force`. These were originally higher-order procedures that produced output procedures which transform systems. Correspondingly, the new versions are higher-order procedures that produce pipeline constructors:

```
(define (scale-system amount)
  (lambda (pipe) (extend pipe (list 'scale amount))))

(define (particle-force force-law)
  (lambda (in-pipe)
    (add-interactor in-pipe (list 'DERIV force-law))))
```

Notice that we have already begun placing requirements on the objects that will comprise the vectors passing through the pipeline in our application. We are requiring that the objects support three messages:

1. (`scale` *amount*) corresponds to the procedure `scale-particle` of the earlier solution. The object is expected to multiply its state information by the specified *amount*.

2. `add` corresponds to `add-particles`. The object will receive a list of states that it must add together to produce its new state.

3. (`deriv` *force-law*) has no direct counterpart in the earlier solution. The object will receive a list of states of other particles in its vector. It must compute the state of its differential particle as was done in the code for `particle-force` in Figure 5.5 on page 130.

In addition to these messages, which are specific to the application of the pipeline, the pipeline construction procedures themselves require some additional messages. These are described in Appendix B.

The final pipeline construction procedure is `runge-kutta-4`. As stated above, the overall goal was that this procedure should not change. This was not, unfortunately, possible:

```
(define (runge-kutta-4 f dt)
  ... definitions of dt*, 2*, 1/2*, and 1/6* as in Figure 5.6 ...
  (lambda (y)    ; y is a PIPE
    (define ys (fork 5 y))
    (define k0s (fork 2 (dt* (f (1st ys)))))
    (define k1s (fork 2 (dt* (f (add-systems
                                 (2nd ys) (1/2* (1st k0s)))))))
    (define k2s (fork 2 (dt* (f (add-systems
                                 (3rd ys) (1/2* (1st k1s)))))))
    (define k3 (dt* (f (add-systems (4th ys) (1st k2s)))))
    (add-systems (5th ys)
                 (1/6* (add-systems (2nd k0s) (2* (2nd k1s))
                                    (2* (2nd k2s)) k3)))))
```

The changes, however, are syntactic. If you compare this procedure with the one shown in Figure 5.6 on page 131 you will notice that the only differences arise from the need to copy the vector (using `fork`) when it arrives at the incoming pipe segment. Subsequent references must specify an appropriate branch. The procedures `1st`, `2nd`, etc. are used here (as well as later) for the names of the appropriate selectors from the list of branches. This requirement arises from the fact that a single variable can be referenced in many places in a program, but the pipeline requires a separate copy for each reference.

Notice that, just as the original version of `runge-kutta-4` was higher-order, so the new version is also. That is, `runge-kutta-4` returns a procedure that will extend a dangling pipe end (`y`) into a newly constructed pipe segment and return the dangling end of this new pipeline.

### 5.3.2 Pipe Vectors and Objects in the Pipe

Having labored to understand the construction of the pipeline we can turn to the second task: making things to put into the pipeline. There are, in fact, three separate problems to address here. We must construct the

pipe vectors (`make-pipe-vector`), construct the objects that comprise the vectors (`make-object`), and we must start the vectors down the pipeline (`start-pipeline`). Of these, the most complicated one, and most informative, is the construction of the objects themselves. This operation is described in detail in this section, while only an overview of the others is provided here (the implementation and details are part of Appendix B).

As mentioned earlier, vectors in the pipeline are for the most part a polite fiction maintained for the benefit of the "outside view" of the pipe. The real work is performed by the objects that make up the vector. A pipe vector, in fact, is a message receiving object that accepts two messages: `size` returns the number of objects in the vector, and `messages` returns the list of messages (i.e. the pipeline input port) to which that vector has been assigned. These messages are used only internally to the pipeline itself, as shown in Appendix B. Once the objects belonging to a vector have been created using `make-object`, the pipe vector itself can be created by passing a list of the objects to `make-pipe-vector`. `Make-pipe-vector` does just a tiny bit more than create the message receiving object: it informs each object that it now belongs to this vector by a mechanism that will be described along with the details of constructing objects.

Once a pipe vector has been constructed, it can be injected into a specific pipeline input port using `start-pipeline`. As shown in the discussion of Appendix B this amounts to nothing more than telling the vector which message list its objects are to examine.

With this background and overview, it is time to discuss the star players in the example: the objects that flow through the pipe. As has been mentioned already several times, the objects are expected to accept a message from the pipe, process it, and continue on to the next message. All objects must be able to handle a set of messages that are used by the pipeline construction operations of Appendix B. In addition, they can be customized to accept additional messages appropriate to a particular application of the pipeline. In the n-body problem, for example, these messages are `add`, `deriv`, and `scale` as discussed in Section 5.3.1. In general, an object consists of some internal state information and a group of message handlers that mutate this state in response to particular messages received by the object.

So far, nothing has really changed from the original solution, except to state that we are dealing at each stage with a vector representing a system of particles flowing through a pipeline. But now for the new twist: each object

in a pipe vector is represented not by a static procedure object (as would be the ordinary practice in Scheme) but rather by an active task. Thus, the pipeline actually operates in parallel on all objects within the pipe at any instant. The objects proceed independently along the pipe until they reach a fork, join, or output segment. At a fork, each object is replicated (once for each output of the fork) creating a task for each output direction of the fork — thus the degree of parallelism increases at these points. At a join, objects wait until all of their partners arrive on the other input paths and then a single object is created to travel along the output path. At an output, all of the objects from a single input vector are gathered together and then are released as a unit. The pipeline system is, in fact, built out of active objects flowing through a statically created pipeline.

This new approach has several advantages over the original. First, by creating processes only when an object is created or at a fork in the pipeline we can amortize the cost of process creation over the straight line length of a pipe. Thus operations that would not merit the creation of a task for each particle in the original implementation can be executed in parallel without remodularizing the program to combine them into larger (hence longer duration) operations.

Furthermore, since each object is an independent task traveling along the pipeline, the system need not idle while one object is undergoing lengthy processing. The objects do not travel in lock-step through the system as they would in a traditional vector processor, and it would be simple, in fact, to allow the pipeline to contain conditional branches or loops that would affect only some of the objects in any given input vector.

To make the job of creating active objects for inclusion in pipeline vectors simpler, `make-object` creates the appropriate task and handles the standard messages required of every object (see Appendix B for a description of these messages). In order to make an object that has its own state and specialized messages we merely specify the initial state of the object, a procedure that will handle object-specific messages, and a procedure that will copy the state of the object. As explained in Section 5.3.1 the objects in our example must handle three kinds of messages.

Before describing the code for `make-object` (shown in Figure 5.11) it will help to see how it is used in this particular example. This will sharpen our understanding of its specification, and thereby make the actual implementation considerably easier to absorb. Because the *pipe vector* (which propagates

through the pipeline) corresponds to the *systems* of the earlier solution, we will require two kinds of objects. These correspond to the elements of a system: the time (whose state is a simple number), and particles (with a more complex state including name, mass, position, and velocity). Time objects are simple to create, given `make-object`:

```
(define (make-time-element time)
  (define (time-messages time m arg)
    (cond ((eq? m 'ADD) (apply + arg))
          ((eq? (1st m) 'DERIV) 1) ; DERIV: dt/dt
          ((eq? (1st m) 'SCALE) (* time (2nd m)))))
  (make-object time                ; Initial State
               (lambda (x) x)      ; Copy State
               time-messages))     ; Message Handler
```

Notice the way the message handler is expected to operate. It will receive three arguments: the current state of the object (`time` in this example), the message to be processed (`m`), and a single additional argument (`arg`). It is expected to update the state information and return the new state. It may do this by side-effecting a data structure or by creating a new one depending upon implementation considerations.

The additional argument, `arg`, has three different meanings depending upon which operation was used to create the pipe segment sending the message:

**Extend**
: The argument is always `'()`. Any information needed to process the message must be included in the message itself and the state of the object. For this reason `scale-system` (on page 145) includes the scale factor as part of the message.

**Join**
: The argument is a list of the states of corresponding objects arriving at the junction.

**Interaction**
: The argument is a list of the states of all the objects in the same vector as this object.

For particles, the state is the same as the data structure used in the serial implementation of particles (a standard Scheme vector). The definition of `scale-particle!` and `add-particles!` (used in the code below)are omitted; they are simply side-effecting versions of the procedures `scale-particle` and `add-particles` shown in Figure 5.3[6].

```
(define (make-particle-element particle)
  (define (interact force-law me all)
    (make-particle (name me) 0          ; name and dm/dt
     (velocity me)                      ; dx/dt
     (reduce add                        ; dv/dt
       (map (lambda (other) (force-law me other))
            (cdr all)))))               ; Ignore time!
  (define (particle-messages state m arg)
    (cond ((eq? m 'ADD) (add-particles! state arg))
          ((eq? (1st m) 'SCALE) (scale-particle! state (2nd m)))
          ((eq? (1st m) 'DERIV) (interact (2nd m) state arg))))
  (make-object particle                 ; Initial State
               vector-copy              ; Copy State
               particle-messages))      ; Message Handler
```

The core of the procedure `Make-Object` is shown in Figure 5.11. The code to handle most messages has been omitted since it is quite straightforward. Only three messages are handled by the code shown in the figure:

- Procedural messages are used in the implementation of the pipeline, as shown in Appendix B. An object (i.e. task) responds to a procedural message by calling the procedure and supplying the object's own message handler. This allows the procedure to effectively send multiple messages to the object without requiring the creation of a complicated and lengthy pipe segment.

- Halt indicates that the object (task) should terminate immediately. It is used both for the output port of a pipeline and for pipeline joins. See Appendix B for examples.

[6]The use of `(cdr all)` to ignore the time of the system when doing the interparticle force calculation corresponds to the construction of the pipe vector in the procedure `serial->pipe-system`, shown on page 154.

- User messages are those not required of all objects. They are handled by calling the user supplied message handler (for example, the procedure `time-messages` for time objects created by `make-time-element`). They pass along the current state of the object, the message, and any additional argument that may have been specified. The user's message handler is expected to return the new state of the object.

One final note before diving into the implementation details. In addition to the job of creating a task for the object, `make-object` returns a Scheme procedure that can be used to activate the task. The task is created in such a way that it will `touch` an undetermined future almost immediately after creation, and hence will become inactive. The procedure returned by `make-object` allows the task to become active once the object is made part of a pipe vector and the pipe vector is placed in the pipeline.

The following items are keyed to the implementation of `make-object` shown in Figure 5.11.

1. The procedure `make-placeholder` creates a placeholder with no associated task. As a result, any task that touches it will hang until some other task explicitly gives it a value (using `determine!`). It is used here as the initial list of messages to be processed by this object so that the task associated with this object will wait until it is inserted into a pipeline. This conveniently implements a "start gate" to hold off execution until a selected time. It receives a value somewhat indirectly, but the start of that process is in step 7.

2. This clause handles procedural messages by providing them with the message handler itself. The task that implements the object executes the procedural message and thus sends itself messages before proceeding on to the next message in the pipeline.

3. (See also step 5.) The procedure `kill-task` is not shown. It removes the task that implements the object from the list of active objects in the system and then releases the processor to other tasks, "committing suicide." This list of active objects is essential to the correct operation of the pipeline. As described in Section 3.4, a task continues to compute only as long as the placeholder which is its goal is referenced by the system. But the pipeline works by using tasks that have no

```
(define (make-object state copy-state user-code)
  (let ((my-cell)
        (vector)
        (object-number)
        (the-messages (make-placeholder)))     ; (1)
    (define (loop message-list)
      (define (standard-handler m #!optional arg)
        (cond
         ((procedure? m) (m standard-handler)) ; (2)
         ((eq? m 'HALT) (kill-task my-cell))   ; (3)
         ... more standard message handlers ...
         (else                                 ; (4)
          (set! state
           (user-code state m (if (unassigned? arg) '() arg))))))
      (standard-handler (car message-list))
      (loop (cdr message-list)))
    (set! my-cell                              ; (5)
     (make-task (lambda () (loop the-messages))))
    (lambda (the-vector number messages)       ; (6)
      (set! vector the-vector)
      (set! object-number number)
      (determine! the-messages messages))))    ; (7)
```

**See text beginning on page 151 for footnotes.**

Figure 5.11: Creating Vector Elements

goal placeholder. Thus the pipeline operations themselves maintain a globally visible data structure (a doubly linked list) of objects still in the pipe in order to guarantee that they will remain in the system after garbage collection. The variable `my-cell` is initialized when the task is created.

4. Messages other than the standard ones are handled by invoking the user's procedure `user-code`.

5. (See also step 3.) The procedure `make-task` is not shown. It creates a task with no goal placeholder and places it on a globally accessible data structure so that the task will not be garbage collected. A pointer to the appropriate part of this global data structure is stored in the variable `my-cell` for use when the object receives the `halt` message. This task begins immediately by calling `loop` to process messages from `the-messages`. Recall from step 1 that this is initially a placeholder with no value, so that the task runs for a very short time before becoming inactive while waiting for the messages to arrive.

6. This is the procedure actually returned from the call to `make-object`. It is used only by `make-pipe-vector` (see Appendix B), and specifies the vector to which this object belongs, the offset within that vector, and the messages which that vector is to process.

7. The task associated with this object will be able to proceed after the call to `make-pipe-vector` since this `determine!` operation supplies a value for the placeholder described in steps 1 and 5.

At this point we have implemented almost all of the pieces we need to make the example work. We can create objects representing particles (`make-particle-element`) and times (`make-time-element`), combine them into pipe vectors representing systems (`make-pipe-vector`), and start them through the pipeline (`start-pipeline`). The only remaining job is to put these pieces together to provide the simplified interface, `integrate-system`.

### 5.3.3 Integration, Pipeline Style

The procedure `integrate-system` is responsible for initializing the pipeline and keeping it busy. Notice that `integrate-system` is defined (in Sec-

tion 5.2) to return a standard stream of results, beginning with the initial state of the system. The solution presented here retains this interface, although doing so is slightly more difficult than in the earlier implementation.

Before proceeding to describe the new implementation of `integrate-system` there are two small jobs to perform. First, review the implementation of `integrate-system` shown in Figure 5.6 on page 131. The new version will be similar, but differs in some significant ways. Second, let's review the pieces we have in our newly constructed pipeline arsenal.

From Section 5.3.1 (and Appendix B) we have the standard pipeline constructor procedures `make-input` (for creating an input port) and `add-output` (for extending a pipe into an output port that appears as an ever-growing list). Using these, we implemented (in Section 5.3.1) some procedures that construct pipeline segments specific to our needs. `Particle-force` takes a force law (in our case we will use the same `gravitation` law used in the solution of Section 5.2) and produces a procedure that will extend an existing pipeline by fitting on a new segment that performs the appropriate interaction calculation. Similarly, `Runge-kutta-4` takes such a pipeline extender procedure and a time step and returns a procedure that will extend an existing pipeline into an integrator pipeline.

We also have some procedures from Section 5.3.2. We will use these to create two new utility routines. The first takes a system of particles in the form used in Section 5.2 and constructs a pipe vector from them:

```
(define (serial->pipe-system serial-system)
  (make-pipe-vector
   (cons (make-time-element (time serial-system))
         (map make-particle-element (particles serial-system)))))
```

This is simple enough. It converts the time and particles from the serial system into the corresponding time and particle objects for the pipeline (using `make-time-element` and `make-particle-element` of Section 5.3.2). These are then combined together into a single pipe vector using `make-pipe-vector`. Notice that it is at the time when this procedure is called that most of the active objects (tasks) are created (the remainder are created when these objects arrive at the forks in the pipeline created by `runge-kutta-4`).

The inverse operation is even simpler. The entry that appears in the list at the output port of a pipeline when a pipe vector arrives at the port is just a list of the state information of the objects that were part of the pipe

vector. Since at the input we made the time object be the first entry in the pipe vector, it will be the first entry in the output as well:

```
(define (pipe->serial-system pipe-system)
  (make-system (car pipe-system) (cdr pipe-system)))
```

The only other operation we need was described in Section 5.3.2 and shown in Appendix B. This is `start-pipeline` which requires a pipeline input port and a pipe vector, and merely inserts the vector into the pipeline.

With these two jobs completed we can discuss the implementation of `integrate-system` shown in Figure 5.12. Notice that the outline of the pipeline version of `integrate-system` and the version in Figure 5.6 are roughly the same. The new version is more complicated, but it still consists of two parts. The first constructs the pipeline and starts it operating. The second is a procedure to compute subsequent values of the system state.

The differences between the two versions arise from two sources. In the pipeline version, it is necessary to construct the actual pipeline when `integrate-system` is called, since it is only then that all of the components are available. Furthermore, two different parts of the pipeline must be located for future use (its input and output ports). These make the construction somewhat more cumbersome than in the earlier version. It is further complicated by a technical point: it would be undesirable to allow the original output port of the pipeline to be visible once the value of the system after the first time step has been extracted. If this were done, then the garbage collector would "see" the potentially infinite list of states of the system over time — instead it is maintained in a local variable that becomes inaccessible after the first call to `next`. This is a standard precaution used in Scheme when dealing with potentially infinite streams.

The second difference is that the pipeline must be started and kept busy. This is done with the two calls to `start-pipeline`. When `integrate-system` is called, the initial state of the system is fed into the input port of the pipeline. Every time a result is extracted from the system (by a call to `next`) the result is also fed back into the pipe so that the pipeline can begin computing the system state at the next time step if it is called for.

This implementation, in some sense, has a bit of speculative computation. It computes one time step ahead of what has been requested. Notice that this is not true of the earlier solution, and as a result performance of the two

```
(define (integrate-system initial-state dt)
  (define integrator-maker              ; (1)
    (runge-kutta-4 (particle-force gravitation) dt))
  (define input (make-input))
  (define (next output-list)
    (let ((next-out (pipe->serial-system (car output-list))))
      (start-pipeline                   ; (2)
       input (serial->pipe-system next-out))
      (cons next-out                    ; (3)
            (delay (next (cdr output-list))))))
  (let ((pipeline                       ; (4)
         (add-output (integrator-maker input))))
    (start-pipeline                     ; (5)
     input (serial->pipe-system initial-state))
    (cons initial-state                 ; (6)
          (delay (next pipeline)))))
```

**Notes:**

1. `Integrator-Maker` will take an input pipeline "dangling end" and extend it into a newly constructed integration pipeline.
2. As each result is pulled from the stream returned by `integrate-system` the output system is turned around and started back down the pipeline. This allows the pipeline to overlap computation of the next answer with the handling of the current answer. See also comment 5.
3. Create the actual item to be added to the stream of answers. As in most stream processes, this is a `cons` of the currently available answer and a promise to compute the next one.
4. Create the complete pipeline by taking the input port (`input`), extending it with an integration pipeline, and capping that with an output port.
5. Start the pipeline computing the state of the system one time step into the future. See also comment 2.
6. The result returned by `integrate-system` is the current state and a promise to compute the next state.

Figure 5.12: Pipeline Version of `Integrate-System`

systems is somewhat hard to compare. To further complicate the measurement problem, notice that `integrate-system` in the pipeline approach is responsible for constructing the pipeline. Since the pipeline is a static structure it seems hardly fair to count this cost as part of the computation (since it is amortized over the total number of time steps ultimately performed).

If the intention of the program is to continue the calculations indefinitely into the future (rather than explicitly by demand), we could have constructed a looping pipeline by placing a fork in the pipeline after the integrator segment. We would then feed one end of this fork into the output port and "weld" the other back to the pipe segment leading into the integrator. This would lead to unbounded speculative computation, since the objects in the pipeline would then be able to loop forever. The implementation shown here would need some modifications to support this possibility.

### 5.3.4 Parallel Pipeline: Performance

Unlike the $n$-body programs in Section 5.2, this pipeline implementation has performance characteristics that are difficult to compare directly with the serial version. In attempting to make this comparison, a number of issues arise that are interesting in their own right. The results reported in this section reflect only a cursory attempt to understand the performance of this pipeline emulation — they are intended (in the same vein as the examples themselves) to illustrate how such measurements can be made. This highlights the more important point that MultiScheme provides a good base for emulating other systems, as discussed in Section 5.4.

First, the good news. The program presented here was able to solve the problem a factor of 6.9 times faster than the original serial version — running somewhat faster than the final version of the program of Section 5.2, although slower than the modified version that motivated this approach[7]

By adding measurement hooks to the pipeline implementation of Ap-

[7]Because of machine availability, the measurements in Section 5.2 were taken separately from those in Section 5.3. The systems used to collect the measurements differed in the total number of physical processors and amounts of memory, although the experiments were performed using the same numbers of active processors. The speed-up measurements are ratios of measurements, both of which are taken from the same system. These ratios, therefore, should be comparable even though the raw data from which they are calculated is not.

pendix B, it is possible to gain more detailed insight into this approach. For example, running one sample data set through the pipeline to perform a single integration step yields the following breakdown of processor time[8]:

| Time *(kticks)* | Description |
|---|---|
| | **Pipeline Message Handling** |
| 424.1 | Handling the pipeline join message. |
| 12.0 | Handling the pipeline fork message. |
| 112.6 | Handling the pipeline interact message. |
| 0.3 | Handling the pipeline output message. |
| | **User Message Handling** (includes time spent waiting for value of placeholders) |
| 535.2 | Calling user supplied message handlers from straight pipe segments. |
| 419.1 | Calling user supplied handler as part of the join operation. |
| 110.6 | Calling user supplied handler as part of the interact operation. |
| | **Pipeline Internal Utilities** |
| 5.8 | Searching and inserting into tables, part of the join, output, and interact operations. |
| 7.3 | Busy-waiting on locks, part of the table operations and process creation and removal. |

In addition, the real time required to compute the new system state was 16.5 kiloticks. Since the computation was done using 16 processors, this indicates that a total of 264 kiloticks of CPU time were available. The measured idle time, totalled across all processors, was 154.1 kiloticks. Thus the actual amount of time spent computing was $264.0 - 154.1$ or 109.9 kiloticks.

[8]The timing units here are "kiloticks," 1000 times the amount of time required to call and return from the procedure `x` defined by `(define (x) 3)`. This time unit is relative insensitive to the actual machine on which the measurements are taken. For comparison, see Appendix C.

We can condense this information to gain an estimate of the cost of implementing the actual pipeline operations. For example, to compute the percentage of time spent in the overhead of a join operation, we take the 424.1 kiloticks required for all joins and subtract the amount of time spent in the user's code that actually combines the incoming data, a total of 419.1 ticks. The pipeline overhead is thus 50 kiloticks, or a total of 4.5% of the CPU time. Performing this calculation for the other operations yields the following picture of processor utilization:

| % Time | Description |
|---|---|
| 10.9 | Pipeline fork operations |
| 4.5 | Pipeline join operations |
| 1.8 | Pipeline interact operations |
| 0.2 | Pipeline output operations |
| 82.6 | User Program |

The 82.6% of processor time available to the user program can be further subdivided. While actual measurement of the amount of time spent in user code (as opposed to, for example, the MultiScheme scheduler and the cost of message dispatching) is a difficult task, we can still make a reasonable estimate. Since the serial version took 732.1 kiloticks to execute (if we disregard garbage collection which did not occur in the parallel case), we can assume that roughly the same amount of time was required by the pipeline execution of the program. This 732.1 kiloticks comes to 66.6% of the total CPU time spent in the program, leaving 16.0% for unallocated overhead.

Finally, as can be seen from this table, 17.4% of the total time was spent in the pipeline operations themselves. Yet from the previous table (under the heading "pipeline internal utilities) only 5.3% of the total time is spent manipulating the tables used for matching objects arriving from different branches of the pipe (at joins and interaction points). At the same time, 6.6% of the total time goes into serializing access to data structures, indicating that the creation and destruction of tasks represents a more serious problem than the matching itself.

### 5.3.5 Parallel Pipeline: Summary

This final example has served to demonstrate three points. It clearly shows the independent utility of tasks, placeholders, and the `determine!` operation.

But notice that tasks (without goal placeholders) are used to implement the objects that propagate through the pipeline (step 5 of `make-object`, on page 153). On the other hand, placeholders (without tasks) are used to function as "gates" that allow tasks to be conveniently activated using the `determine!` operation (steps 1 and 7 of `make-object`).

Second, while the details have not been shown, notice that the decision to use tasks without specific computation goals makes the garbage collector a danger to the computation. Special arrangements (hidden inside `make-task` and `kill-task` in steps 3 and 5 of `make-object`) are required to maintain a global data structure that specifies the tasks that are in the pipeline lest they be removed at the next garbage collection. Unfortunately, this data structure can retain tasks that are no longer useful. For example, if the user stops all tasks using `pause-everything` and then discards the tasks which were running (an operation that is particularly convenient since it corresponds to the standard "abort" interrupt handler) the data structure will still contain references to the tasks. These tasks will not drain processor resources (they are no longer candidates for processor time) although the data structures that support them will remain in memory.

But most importantly this example shows that MultiScheme's packaging of parallel computation allows a radically different approach to parallel programming to be embedded within the language. To emphasize this point, it is worth reviewing this solution to the n-body problem from this particular point of view. Bear in mind that the pipeline procedures of Appendix B together with `make-object` of Section 5.3.2 represent the interface to the embedded computation model.

Using these, we easily created four procedures that construct a pipeline suitable for the particular problem under consideration (Section 5.3.1). In order to tailor the operations of the pipeline to our own application, we added two more procedures that construct objects with properties specific to our application (Section 5.3.2). We created two procedures to convert between the serial representations of particle systems that the user normally deals with and the pipe vectors which propagate through the pipeline (Section 5.3.3). Finally, we wrote the only complicated procedure (`integrate-system`) which serves to keep the pipeline full and convert its results into a more standard representation[9]. In fact, the total amount of code needed to solve the problem

[9]In fact, this final procedure is probably a sufficiently common type of operation that

using this interface amounts to less than two pages of text, and much of this is nearly unchanged from the original implementation.

This work, once the pipeline interface is specified, is not difficult. In fact, it seems likely that anyone with a reasonable familiarity with Scheme and the application itself (i.e. the use of a Runge-Kutta integrator to solve the n-body problem) could be expected to construct the solution. The embedding of the pipeline is a more significant project as is only to be expected. Yet, as can be seen from the code in Appendix B, even this project is not hard once the interface is defined.

## 5.4 Other Methodologies

This section shows how three additional parallel-programming methodologies can be expressed in MultiScheme. Section 5.4.1 briefly reviews the extensions to Lisp proposed by Gabriel and McCarthy[21] for introducing parallelism, and then presents a set of syntactic rewrite rules (macros) that provide these same constructs within MultiScheme. Section 5.4.2 outlines a common approach for command-based (as opposed to expression-based) languages, the `fork` and `join` construct. Again, this is easily embedded within MultiScheme, although the "linguistic fit" of these constructs is rather poor in an expression-based language like Lisp. Finally, Section 5.4.3 discusses the approach to parallelism advocated by BBN for the C language[56].

### 5.4.1 QLambda

In the summer of 1984, prior to the work on MultiScheme, Gabriel and McCarthy suggested a trio of extensions to Lisp[21] intended to support parallel programming. In their language, the special forms `lambda` and `let` have matching counterparts, `qlambda` and `qlet`, that allow the user to specify how eagerly the body of the expression is to be evaluated. The basic model is one of independent tasks communicating by a queue of input arguments. Evaluation of a `qlambda` expression leads to a new task running independently. A call to this task is performed by evaluating the arguments to the task and putting them on the task's queue. Evaluation of a `qlet` allows the

it would be provided as a part of the interface to the pipeline package in a more developed implementation.

values of the bound variables to be calculated in parallel with the execution of the body of the `qlet` expression.

Each of these constructs can operate at three eagerness levels (my names for the levels, not theirs): `none`, providing behavior identical to the standard (sequential) special forms; `normal`, creating a task that waits for all of the argument values to arrive before proceeding on to execute the body in parallel with the call to it; or `eager`, creating a task and executing the body in parallel with the task pausing only when it attempts to reference arguments that have not yet arrived. Gabriel and McCarthy proposed a similar, but more complicated, extension for `catch` and `throw`.

In March of 1985, with the help of Byron Davies (a member of the group at Stanford that was developing QLisp), a set of macros was developed for converting programs written using `qlet` and `qlambda` into standard Multi-Scheme programs. No support was provided for `qcatch`, since its definition was still under active development at the time.

The results of this work, somewhat revised, are described below. Each of the six cases (two special forms, each with three eagerness values) is described. In QLisp, the amount of eagerness may be specified either syntactically (by using a particular constant for each of the three values) or at runtime (by using a non-constant expression). The rewrite rules expand as shown if the amount of eagerness is constant. Otherwise, they expand into a runtime case dispatch to the correctly expanded version.

### Converting `qlet` Expressions

A `qlet` expression has the form

`(qlet` <*e*> `(`<*bindings*>`)` . <*body*>`)`

where <*e*> specifies the amount of eagerness and each <*binding*> consists of a variable and an expression. To make the explanation simpler, we will expand a particular `qlet` expression:

```
(qlet e ((a (f 3))
         (b (g 4)))
  (cons a b))
```

If the eagerness is `none`, this converts to the standard `let` special form:

```
(let ((a (f 3))
      (b (g 4)))
  (cons a b))
```

If the eagerness is normal, then the intent is to evaluate the expressions in the <*bindings*> in parallel. When all of these values are available, the body can then be evaluated:

```
(let ((a (future (f 3)))
      (b (future (g 4))))
  (touch a)
  (touch b)
  (cons a b))
```

Finally, an eagerness of eager indicates that execution of the body should proceed concurrently with evaluation of the expressions in the <*bindings*>:

```
(let ((a (future (f 3)))
      (b (future (g 4))))
  (cons a b))
```

Notice that in this eager case the value (a `cons` cell) may well be returned to the caller long before the values of `(f 3)` and `(g 4)` are computed. Since `cons` does not `touch` its arguments the cell will contain two placeholders that ultimately receive these values.

### Converting `qlambda` Expressions

A `qlambda` expression resembles a standard `lambda` expression:

`(qlambda` <*e*> <*formals*> . <*body*>`)`

again, <*e*> is the eagerness. An eagerness value of none leads to operation just like the standard form:

`(lambda` <*formals*> . <*body*>`)`

The remaining cases are considerably more complex, and the published description ([21]) is somewhat less clear about the exact semantics. The description given here corresponds to the best understanding we were able to arrive at after consultation with members of the Stanford project.

(qlambda normal $<formals>$ . $<body>$) $\rightsquigarrow$

```
(let ((argument-queue (make-empty-queue)) ; (1)
      (the-task))
  (define (thunk)                         ; (2)
    (let ((arglist (dequeue argument-queue)))
      (determine! (car arglist)
                  (apply (lambda <formals> . <body>)
                         (cdr arglist))))
    (thunk))
  (set! the-task (future (thunk)))        ; (3)
  (lambda <formals>                       ; (4)
    (let ((answer (make-placeholder)))
      (enqueue argument-queue
               (list answer . <formals>))
      answer)))
```

**Notes:**

1. `Argument-Queue` is filled by callers and emptied by the free-running task. Each entry has a placeholder for the result to be returned as well as values for the arguments.
2. Code for the free-running task: dequeue a set of arguments and result placeholder, calculate, store the result, repeat forever.
3. Start the task running when `qlambda` is evaluated.
4. Code for caller: create a placeholder for the answer, enqueue arguments and placeholder, return the placeholder.

Figure 5.13: `QLambda` with normal Eagerness

For an eagerness of normal, the intent is to create a free-standing task when the `qlambda` expression is evaluated. This task repeatedly tries to read a set of arguments from an input queue and produce a corresponding result. When the call occurs, all of the arguments are evaluated and their values are placed on the task's input queue. The caller receives a placeholder for the value to be computed by the task and continues on past the call, pausing only if it `touch`es the result of the call before the task has computed the answer. This can be implemented using atomic queue operations `enqueue` to add an item to an existing queue and `dequeue` to remove an item or suspend the caller if no data is available[10]. The code required is shown in Figure 5.13.

This code deserves a little explanation. The work of a `qlambda` occurs at two different times. When the `qlambda` expression itself is evaluated to form a "process closure" the argument queue is created and the independent task begins running. The process closure itself is created (by evaluating the `lambda` expression in the above expansion). When this process closure is applied to arguments both the arguments and a location (placeholder) where the result should be stored are added to the queue of work for the task.

The task, of course, is just a simple infinite loop. It attempts to read an item from its queue. When it succeeds, it will receive both the arguments and a placeholder into which to store the result. It then computes the required function and stores the result away using `determine!`. Since the placeholder associated with the task is lexically visible (as the value of `the-task`), it will continue running as long as the process closure remains accessible. The garbage collector will remove the task when the closure becomes inaccessible[11].

The final case, an eagerness value of eager, closely resembles the previous case. There are two differences: the task begins executing the body immediately when the `qlambda` expression is evaluated (rather than waiting for arguments to arrive from a call), and the task pauses only if it has produced an answer that has not been requested yet. This latter case is rather strange;

[10] Atomic `enqueue` and `dequeue` operations can in turn be written using the built-in atomic operations `set-car-if-eq?!` and `set-car!`.

[11] There is a race condition here that allows the task to disappear after it has received its final set of arguments but before computing a final value. One solution, due in part to Halstead, is to change the final line from `answer` to `(future (begin (touch answer) answer)))`. Explaining how this works is left as a challenge to the reader...*warning:* this may not work if you have a good compiler!

```
(qlambda eager <formals> . <body>) ⇝

(let ((argument-queue (make-empty-queue)) ; (1)
      (the-task))
  (define (thunk)                         ; (2)
    (let ((args (cons (make-placeholder)
                      (make-list (length <formals>)
                                 make-placeholder))))
      (enqueue argument-queue args)
      (let ((answer (apply (lambda <formals> . <body>)
                           (cdr args))))
        (determine! (car (car args)) answer)))
    (thunk))
  (set! the-task (future (thunk)))        ; (3)
  (lambda <formals>                       ; (4)
    (let ((answer (list (make-placeholder)))
          (args (dequeue argument-queue)))
      (determine! (car args) answer)
      (for-each determine! (cdr args) (list . <formals>))
      (car answer))))
```

**Notes:**

1. `Argument-Queue` is filled by the free-running task with placeholders where it will look for values of arguments and where to store a result. It is emptied by callers, who store the values of arguments and where they want the result stored.
2. Code for free-running task: make an entry on `argument-queue`, calculate, store result, repeat forever. Notice that `car` touches its argument, so this loop pauses after calculation until a call specifies where the result should be stored.
3. Start the free-running task going when the `qlambda` is evaluated.
4. "Process closure" returned to the user: dequeue an entry from `argument-queue`, fill it with values of arguments and a placeholder where the answer should go, return this placeholder.

Figure 5.14: `QLambda` with `eager` Eagerness

it can arise only if the body of the `qlambda` does not `touch` any of its arguments. This problem is handled in the code shown in Figure 5.14, leading to a somewhat unusual implementation. In the program shown here, the free-running task begins operation as soon as the `qlambda` expression is evaluated, but pauses when it attempts to use `determine!` to store its answer. It can proceed only when some task has called the closure and therefore supplied a placeholder for the result. At the same time, a caller must pause until the free-running task has created an entry on its queue into which arguments for the call can be placed. This last constraint does not permit a caller to simply leave the arguments on a queue and proceed — modifying the program to permit this, however, is simple enough: simply enclose the body of the `lambda` expression in a `future`!

### 5.4.2 Fork and Join

Another common language construct for parallel programming is the `fork` and `join` found, for example, in Simultaneous Pascal[10]. This construct is used in a command-oriented language to indicate that the commands between the `fork` and `join` (which must nest in the same way that `begin` and `end` nest) are to be executed in parallel. When they have all completed execution, control continues past the point of the `join`. Another common variation on this control construct is `do-all`, which executes a single block of code with respect to each index in a given range (for example, all the subscripts in a one-dimensional array). A more complicated construct operating in a similar manner is the `coenter` of Argus[38]. The implementation of this control structure in MultiScheme is simple enough: simply wrap each command in a `future`, gather a list of the resulting placeholders, and `touch` them all.

A slightly more interesting control structure, where precedence constraints are explicitly indicated, can be achieved by using placeholders to act as triggers. Thus, each precedence constraint is represented by a placeholder. When the constraint is satisfied, an explicit `determine!` operation is performed by the task that has satisfied the constraint. Similarly, any block of code that must wait for a constraint to be satisfied merely `touch`es the appropriate placeholder. This organization can be used conveniently in situations where it is easy to predict which block of code is responsible for satisfying each constraint. This kind of precedence handling is not directly related to the more powerful "constraint-based systems" familiar to the artificial intelli-

gence community. These systems do not possess this kind of predictability, so the trigger mechanism will not make the job of designing such a constraint-based system significantly easier.

### 5.4.3 Uniform System

One final model for writing parallel programs is the Uniform System[56], a methodology advocated by BBN for use on the Butterfly computer system. The essence of the method is to devise a "task generator" that runs as a serialized critical region and provides callers with information about what job to perform next. In addition, users supply a main program that is run simultaneously on all available processors. Typically the main program is a loop that calls a task generator, does the work specified, and asks for more work — that is, it supports precisely the polling model of synchronization discussed in Section 1.3.2.

The proponents of this methodology point out that it can be made highly efficient by carefully identifying data shared between processors (used by the task generators) and data which can be kept local to a particular processor. Indeed, the results achieved by using carefully implemented (but general purpose) task generators is quite impressive[13], leading to very nearly linear speed increases over the range from one to 256 processors. These results show that this technique can avoid many of the "hot-spot" (contention) problems found in other methods used on the Butterfly.

MultiScheme deliberately takes the opposite view and encourages users to ignore the distinction between private and shared data storage. If MultiScheme and the Uniform System were otherwise comparable, this difference of approach would be an interesting one to examine with respect to a number of architectures. Unfortunately, MultiScheme suffers in two additional ways: it is purely interpreted (which makes it far slower and at the same time reduces contention for individual memory locations), and it must strip and restore type codes as it references memory. This additional overhead makes the direct comparison of the two systems uninformative.

It is interesting to notice that programs can be written in the "Uniform System style" rather simply in MultiScheme (a fact that has aided some users in moving programs from one system to the other). The only problems are to see how to scale the number of MultiScheme tasks to the actual number of processors available, and the implementation of critical regions. Both of

these problems are easily solved, the former by using the `N-Interpreters` primitive mentioned in Section 3.2, and the latter using semaphores built from a spin lock using `set!` for an atomic swap operation on the lock cell.

In order to make this more concrete, Figure 5.15 shows one way to write a uniform system simulator. In order to use this simulator, we must first create a package containing the data used by the tasks, a well as the "core" of a task generator. As an example, consider writing a program to calculate the factorial function by parceling out multiplication tasks of 50 numbers each. The appropriate core procedure is shown in Figure 5.16.

The core should contain two variables shared by all tasks, one of which keeps track of the next number to be multiplied (`next-start`), and the other contains the final answer `result`. In addition, each task will need its own local variable to keep track of its part of the product being calculated (`task-value`) — note the use of `fluid-let` in Figure 5.16 to convert this single shared variable into a per-task variable. In addition, the `uniform-system` procedure itself requires four procedures:

**end-test**

This predicate returns `#T` when there is no more work available to be parceled out. When this happens, the uniform system enters its "termination phase." In this example, termination begins when the next number to multiply (the global variable `next-start`) exceeds the argument originally given to `fact`, `n`.

**next**

When the system is not in termination phase, this procedure generates the next task to be performed. In this example, the work is to multiply the current `task-value` (a per-task variable) by a range of numbers (handled by the procedure `multiply-range`, not shown here). The result becomes the new `task-value`.

**end**

This procedure is called, as part of a critical region, once by each task when the system is in its termination phase. It performs any final code before the task is killed, and the last task to be killed is expected to return the final value of the entire program. In this example, it multiplies this task's contribution to the final answer (`task-value`) into the system-wide final `result`.

```
(define (uniform-system core)
  (define result (make-placeholder))
  (define (make-task-generator core)    ; (1)
    (let ((terminated-tasks 0)
          (end-test (access end-test core))
          (next (access next core))
          (end (access end core))
          (n-tasks (n-interpreters)))
      (make-critical-region             ; (2)
       (lambda ()
         (if (end-test)                 ; (3)
             (let ((answer (end)))
               (set! terminated-tasks (1+ terminated-tasks))
               (if (= terminated-tasks n-tasks)
                   (determine! result answer))
               kill-task)
             (next))))))                ; (4)
  (define task-generator (make-task-generator core))
  (define (loop) ((task-generator)) (loop))
  (define workers                       ; (5)
    (make-list (n-interpreters)
               (lambda () (future ((access start core) loop)))))
  (touch result))                       ; (6)
```

**Notes (see also page 172):**

1. `Make-task-generator` takes the `core` of a user-specified task generator and creates a full-fledged handler. See Figure 5.16 for a sample core.
2. `Make-critical-region` serializes calls to its argument.
3. Termination phase: call `end` to allow the task to do any final code, then return `kill-task` as the work to perform. If there are no more tasks, announce the `result`.
4. Call the user-supplied task generator `next`.
5. Spawn off the actual tasks, one per processor.
6. Wait for an answer before returning to the caller.

Figure 5.15: Uniform System Simulator

```
(define (make-fact-core n)
  (make-package fact-core
    ((next-start 1)                      ; (1)
     (result 1)

     (task-value))                       ; (2)

    (define (end-test) (> next-start n)) ; (3)
    (define (next)                       ; (4)
      (let ((start next-start)
            (end (min n (+ next-start 49))))
        (set! next-start (1+ end))
        (lambda ()
          (set! task-value
                (* task-value
                   (multiply-range start end))))))
    (define (end)                        ; (5)
      (set! result (* result task-value))
      result)
    (define (start work)                 ; (6)
      (fluid-let ((task-value 1)) (work)))))
```

**Notes (see also page 169):**

1. Global variables used by all tasks.
2. Local variables, per-task.
3. Test for end of task generation phase.
4. Generator for next task.
5. Calculate final value of task.
6. Start task by initializing per-task variables then processing `work`.

Figure 5.16: Core of `Fact` for Uniform System

**start**

This procedure is called once by each task when it is started. It is expected to initialize any local state information (in this example, the variable `task-value`) and then call its argument which will cause it to run tasks generated by calls to `next` or `end`.

With this package created, we can examine the workings of the uniform system simulator of Figure 5.15. It works by creating a placeholder for its final `result`, then converting the core package of procedures into a serialized task generator (using `make-critical-region`, not shown, a simple procedure that uses the atomic `set!` operation to create a busy-waiting lock procedure). It then makes the tasks by calling `n-interpreters` to find out how many processors are available and creating a MultiScheme task for each processor. Nothing guarantees that each of these MultiScheme tasks is attached to a single processor, however by making precisely one task per processor we can guarantee that the number of tasks scales to the number of processors.

The final detail is the operation of the critical region in the task generator. This calls the `end-test` from the user supplied package to see whether the termination phase has begun. If not, the user's `next` procedure is called to generate a new item of work to be executed by this task. If so, the user's `end` procedure is called to let the task perform any final actions and return an answer. If this is the last task remaining, this answer becomes the value returned by the entire simulation. In any case, the task is then told to call `kill-task`, causing it to "commit suicide."

Finally, we can define a parallel factorial function:

```
(define (fact n) (uniform-system (make-fact-core n) 1))
```

### 5.4.4 Other Methodologies: Summary

As we have seen, MultiScheme can be used (with some creative thought) to emulate a large range of parallel programming methodologies. It provides a nice environment for "cobbling up" system prototypes written using these other methods, and this in turn aids the rapid exploration of a number of different styles for writing a program. All of these are similar to Lisp's well known advantages as a prototyping system in a sequential computing environment.

There are limits, however, to the information that can be gained from these emulations. These emulations are very useful in answering a variety of questions:

- How can I write a selected program in a particular methodology?
- Does the program produce the expected results? (If not, the emulator is a major aid in debugging the program.)
- Does the program utilize the additional processors?
- What mechanisms are needed to support this methodology?

On the other hand, deeper questions of resource utilization are often obscured by the emulation overhead. Furthermore, MultiScheme is deliberately designed to take certain kinds of resource allocation issues out of the programmer's hands. This is most noticeable when exploring questions related to communications models or performance estimates that are intended to take communication costs into account.

# Chapter 6

# Conclusion

Over a three-year period, MultiScheme has evolved to the point where it can serve as the system base for significant parallel programming projects. In fact, any of the examples discussed earlier could become the starting point for such an investigation. This idea is pursued in Section 6.1, in which the major accomplishments of the MultiScheme project are summarized and a number of newly-feasible research areas are described. But the work on the MultiScheme system itself is far from complete, and some obvious steps to improve it are the topic of Section 6.2.

## 6.1 What Have We Wrought?

The philosophy behind the design of MultiScheme is simple: add some straightforward extensions to Scheme and thereby produce a new language with considerable additional expressive power. For sequential computing, the addition of placeholders provides the basic support for embedding logic variables in Scheme programs and data structures (see Section 2.3) as well as call-by-need arguments (Section 2.4). Furthermore, MultiScheme programmers have a rich language for interacting with the memory management system — the `weak-cons` primitive allows an object to be referenced without altering its extent, while `finalization` allows an object's extent to be increased.

For parallel computing, the separation of placeholders from tasks allows

a variety of programming methodologies to be expressed conveniently within the MultiScheme system (see, for example, the pipeline example of Section 5.3 and Appendix B, as well as Section 5.4). In addition, the ability to *change* the value of a placeholder (using `mutable-determine!`) has been a key item in the support of work on parallel programming in programs with side-effects[34].

In fact, the importance of placeholder objects may go well beyond these examples. Historically, the addition of a new Lisp data type has served to focus the Lisp community's attention on problems of interest to the larger computer science community. The `cons` cell, for example, proved to be a simple structure for studying sharing; the first-class procedure forced attention onto the areas of lexical scoping and program structure; and the first-class continuation invited experimentation with control structure. In a similar manner, placeholders help to bring several very different programming issues into focus: parallel computation, logic programming, and normal-order languages. The MultiScheme work is certainly not the first to point out the connections here, but it is the first to point out that a single data structure can provide a uniform method for studying all of these areas.

Another positive result of the MultiScheme project is that a few simple extensions to Scheme lead to a system sufficiently robust that it can be used for substantial application programs in the parallel-programming domain. As a result of this work we can undertake a set of projects that were not previously feasible. Some of these projects include:

- Embedding a logic programming language within MultiScheme by combining a unification algorithm with the placeholder implementation of logic variables (Section 2.3) and control structures like the dependency-directed backtracking described by Zabih *et al*[61]. Not only should such a system provide a number of exciting possibilities because of the unusual control structure, but by embedding it within MultiScheme we create a laboratory for exploring the use of parallelism both within the logic programming component and within programs written using that system. This method of extending logic programming into the parallel domain may not be practical, even though it will certainly be instructive: placeholders serve to represent a form of structure sharing, but this very sharing is a problem in the parallel system where several potential values for a single logic variable are under consideration

simultaneously.

- MultiScheme provides an expressive base language, as does Scheme. But the power of Scheme derives largely from the abstractions that can easily be written in the base language and then used without reference to the implementation technology. Such abstractions have yet to be explored within MultiScheme. There are two areas ripe for exploration here: parallelism in control structures, and parallelism in data structures. Writing a number of parallel search algorithms in MultiScheme would serve as a good starting point for both of these areas. In the data structure area, providing a package of routines with an interface similar to that of the Connection Machine's *Lisp[55] language should be straightforward, and would allow explorations into data parallel programming in MultiScheme. Similarly, implementing the `frons`[19] operation should also be straightforward based on the current implementation of `disjoin`.

- MultiScheme provides two primitive mechanisms for supporting speculative computation: a garbage collector to eliminate unnecessary tasks, and a flexible scheduler to facilitate experiments in priority-based execution of tasks. The first step in understanding how this support can be used in practice is the creation of programs that exploit speculation to solve realistic problems.

## 6.2 What Remains To Be Done

While the MultiScheme system is now sufficiently stable to be used for the kinds of projects described in Section 6.1, much remains to be done to improve the system. Some of this work involves implementation details or small changes to the system, but there are two important areas that have only been touched upon during the construction of MultiScheme: performance and user interface. These are extremely promising areas for future research efforts.

### 6.2.1 Performance Enhancements

The most woefully neglected part of the MultiScheme work has been its performance. There are three major research directions here: compilation,

architectural changes, and system changes.

**Compilation**

Compilation technology for Lisp and Scheme is relatively well developed[36]. Most of the standard techniques are directly applicable to parallel as well as sequential programs, although some standard optimizations (such as common sub-expression elimination) have the potentially undesirable consequence of reducing the amount of parallelism even though they accelerate a sequential thread. A MultiScheme compiler would help answer a number of questions about memory contention due to Lisp's reference pattern, as well as providing the basic set of tools needed for program analysis.

**Architecture**

Even the best compiler can only do a limited amount when faced with an architecture poorly suited to the language and system. Recent work by Wu[60] suggests that specialized hardware aimed at supporting Lisp may well improve the cost-performance ratio of a processor within a given technology family. The major forms of hardware support for Lisp are well understood[41], and apply directly to parallel architectures as well. Beyond these support mechanisms, however, parallel architectures are critically sensitive to memory latency and bandwidth. As stated in Section 1.3.1 there are two fundamental requirements on a parallel architecture:

1. A large portion of the address space must be shared by (accessible to) all processors in the system.
2. The speed of reference to all portions of the address space must be comparable.

The closer an architecture comes to realizing constant access speed to all areas of the address space, the less will be the impact of any non-locality of reference by programs.

**System**

Unfortunately, the "equal access" goal stated in the previous paragraph is very hard to meet. Thus, system support aimed at keeping objects

close (in speed of access) to the task referencing them may have significant benefits. The simulation studies of Nuth[42] are a first step in understanding the relationship between performance characteristics of the memory subsystem and overall system performance. Using these same kinds of simulations, it may be possible to tune the memory and task management to more closely fit specific architectures.

### 6.2.2 User Interface

An important, and often overlooked, aspect of any programming system is the user interface. At present, the MultiScheme user interface is directly based on the MIT Scheme user interface, in turn derived from the standard Lisp Read–Eval–Print loop. While perfectly serviceable, this interface was never intended to help programmers *visualize* the activity of their programs (cf. Turbak[57]). This visualization is even more crucial in a parallel system than in a sequential one. Providing an easily understood model of the program's behavior while it is running serves several important purposes:

**Displayed Parallelism**

Any one of a number of techniques can be used to keep the programmer fully aware of the amount of parallelism actually exploited by her code. Not only does this help programmers understand the amount of parallelism in their programs, it also serves as a subtle reminder that the program has multiple concurrent threads of control. Even after two years of MultiScheme experience, I find it easy to forget this rather obvious fact: programming a parallel computer is *not* the same as programming a time-sliced sequential computer.

**Visualizing Dependencies**

When dealing with a running parallel program it is important to understand which tasks depend on one another. This information can be quite difficult to derive, since a `future` can be stored in a data structure that is accessible from a number of tasks, voiding most kinds of simple analysis. One vital role of a good user interface to MultiScheme must be to provide tools (visual or otherwise) to correctly answer the dependency question, and to subtly remind the programmer that the dependency question must be answered independently of the creation behavior of the program.

**I/O handling**

An important aspect of interacting with a parallel system is how tasks communicate with the user. MultiScheme provides "hooks" in the form of per-task variables that allow tasks to have independent I/O streams for communication, and the Butterfly Lisp User Interface (BLUI) uses these to provide a separate window for each task which attempts an I/O operation. But the ability to *interrupt* a specific task is missing — and implementing it will not be easy, since there are many tasks that do not themselves perform I/O and so have no window through which a user can "point" to the task to generate the interrupt.

**Debugging Tools**

A user interface can also be an extremely powerful debugging tool. This, in fact, is the source of much of Lisp's power as a program development environment: the Read–Eval–Print loop allows users to interact as though they were part of the program, so that inspecting and modifying the state can be done in a natural manner. This style becomes even more powerful when coupled with an interface that makes the computational environment visible and allows programmers to take advantage of position to direct their interactions. SmallTalk[22] popularized this form of interface, and a similar interface for Scheme was demonstrated by Eisenberg[15].

The fundamental components of a user-interface for MultiScheme are understood: Turbak's visualization of control structure, Eisenberg's visualization of binding and interaction environments, Clamen's task monitoring and dependency displays, graph traversal (garbage collection) algorithms, BLUI's input/output and performance displays. But combining these into a single, coherent interface remains a challenging project.

## 6.3 Parting Shots

MultiScheme is only a start along the long road towards a convenient and powerful system for programming parallel processors. The basic structure, largely inherited from its Lisp ancestry, allows abstractions to be layered on top of the system's kernel in order to extend the system in novel directions. But work remains both under, in, and above this kernel. The architectural

support *under* the MultiScheme system, for Lisp in general as well as for placeholders and first-class continuations (tasks), must be pursued to the point of producing a set of criteria for evaluating potential machines as a base for MultiScheme implementations. Critical to this effort is an understanding of the inter-processor communication inherent in the MultiScheme model, and work directed at the communication technology of the underlying hardware. Work *in* the system includes performance enhancements, user interface improvements, and the development of tools for debugging and "performance debugging" of programs. Work *above* the system includes the construction of abstractions for a variety of control and data structures and the implementation of significant application programs that exploit these new abstractions.

But the single overriding lesson I have learned from the MultiScheme work comes from a different direction entirely. In building the MultiScheme system, teaching students about it, and in describing it in this report, I have seen repeatedly not how different parallel computing is from sequential computing, but rather how *similar*. This lesson is perhaps best demonstrated by observing that most of the changes made to convert MIT Scheme into MultiScheme have been re-absorbed into the sequential system — because these additions have proven to have important benefits in sequential computing. The outstanding example, of course, is the placeholder itself: a data structure derived entirely from considerations of parallel programming, yet providing the base for several important facilities in the sequential system. It is my belief that future work will further strengthen this similarity. In retrospect, we will discover that the importance of programming parallel systems comes from insights applicable to both sequential and parallel computing.

I summarize this lesson in a simple analogy:

sequential : parallel (computing) :: real : complex (analysis)

# Bibliography


[1] Hal Abelson and G. J. Sussman. Procedural abstractions in lisp programming. Tentative title, submitted to *Byte* Magazine, August 1987.

[2] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.

[3] J. F. Applegate, M. R. Douglas, Y. Gürsel, P. Hunter, C. Seitz, and G. J. Sussman. A digital orrery. *IEEE Computer*, September 1985.

[4] H. Baker and C. Hewitt. The incremental garbage collection of processes. Technical Report AI Memo 454, Mass. Inst. of Technology, Artificial Intelligence Laboratory, December 1977.

[5] John Batali, Edmund Goodhue, Chris Hanson, Howie Shrobe, Richard M. Stallman, and Gerald Jay Sussman. The Scheme-81 architecture – system and chip. In Paul Penfield Jr., editor, *Proc. of the MIT Conference on Advanced Research in VLSI*, Dedham, Mass., 1982. Artech House.

[6] BBN Advanced Computers, Inc., Cambridge, MA. *Butterfly*™ *Parallel Processor Chrysalis*™ *Programmer's Manual*, August 1986.

[7] BBN Advanced Computers Inc., Cambridge, MA. *Butterfly Parallel Processor Overview*, 1987.

[8] BBN Computer Corporation, Cambridge, MA. *CMOS Manual*, June 1981.

[9] Bolt Beranek and Newman Laboratories, Inc. Multiprocessor system architectures: The monarch multiprocessor, vol. 2: Technical proposal.

Technical report, Bolt Beranek and Newman Laboratories, Inc., November 1984.

[10] Ellery Chan et al. Reference manual for simultaneous pascal. Harris Corp. internal document, 1986.

[11] Stewart Michael Clamen. Debugging in a parallel lisp environment. Bachelor's thesis, Mass. Inst. of Technology, 1986.

[12] Anthony James Courtemanche. Multitrash, a parallel garbage collector for multiScheme. Bachelor's thesis, Mass. Inst. of Technology, 1986.

[13] W. Crowther et al. Performance measurements on a 128-node butterfly™ parallel processor. In *Int'l. Conf. on Parallel Processing*, 1985.

[14] J. Edler et al. Issues related to mimd shared-memory computers: The nyu ultracomputer approach. In *Symposium on Computer Architecture*, pages 126–135, June 1985.

[15] Michael Eisenberg. Bochser: An integrated Scheme programming system. Master's thesis, Mass. Inst. of Technology, 1986. Available as MIT/LCS TR-349.

[16] Michael Eisenberg. Programming in Scheme: An introduction. In preparation for Scientific Press, 1988.

[17] Matthias Felleisen, Daniel Friedman, Eugene Kohlbecker, and Bruce Duba. Reasoning with continuations. In *Proc. of the Symp. on Logic in Comp. Sci.*, pages 131–141, Washington, DC, 1986. IEEE Computer Society Press.

[18] Robert E. Filman and Daniel P. Friedman. *Coordinated Computing.* McGraw-Hill, New York, 1984.

[19] D. Friedman and D. Wise. An indeterminate constructor for applicative programming. In *ACM Symp. Princ. Program. Lang.*, pages 245–250, Las Vegas, Nevada, January 1980.

[20] Daniel Friedman, Christopher Haynes, and Eugene Kohlbecker. Programming with continuations. In P. Pepper, editor, *Program Transformation and Programming Environments*, pages 263–274. Springer-Verlag, 1984.

[21] R. P. Gabriel and J. McCarthy. Queue-based multi-processing lisp. In *Proc. ACM Symp. LISP and Functional Progr.*, pages 25–44, Austin, Tex., August 1984.

[22] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley series in Computer Science. Addison-Wesley, Reading, Massachusetts, 1983.

[23] Sharon L. Gray. Using futures to exploit parallelism in lisp. Master's thesis, Mass. Inst. of Technology, 1986.

[24] R. Halstead. Multilisp: A language for concurrent symbolic computation. In *ACM Trans. on Prog. Languages and Systems*, pages 501–538, October 1985.

[25] R. Halstead, T. Anderson, R. Osborne, and T. Sterling. Concert: Design of a multiprocessor development system. In *Symposium on Computer Architecture*, volume 13, pages 40–48, June 1986.

[26] Christopher Hanson and John Lamping. Dynamic binding in Scheme. Unpublished paper.

[27] C. T. Haynes and D. P. Friedman. Engines build process abstractions. In *Proc. ACM Symp. LISP and Functional Progr.*, pages 18–24. ACM, 1984.

[28] Christopher Haynes, Daniel Friedman, and Mitchell Wand. Continuations and coroutines. In *Proc. ACM Symp. LISP and Functional Progr.*, pages 293–298. ACM, 1984.

[29] Peter Henderson. Can a programming system be written in a fully functional style? In John Darlington et al., editors, *Functional Programming and its Applications. An Advanced Course.* Cambridge University Press, New York, 1982.

[30] Hewlett-Packard Company, Fort Collins, CO. *HP-UX Reference Manual*, 1985. Volume 3, Section 3C.

[31] W. Daniel Hillis. *The Connection Machine*. PhD thesis, Mass. Inst. of Technology, 1985. Available from MIT Press as part of the ACM Distinuished Dissertation series.

[32] Jack Holloway, Guy Lewis Steele Jr., Gerald Jay Sussman, and Alan Bell. The Scheme–79 chip. Technical Report AI Memo 559, Mass. Inst. of Technology, Artificial Intelligence Laboratory, 1980.

[33] Kirk Johnson et al. Personal Communication.

[34] M. Katz. Paratran: A transparent, transaction based runtime mechanism for parallel execution of Scheme. Master's thesis, Mass. Inst. of Technology, May 1986.

[35] Tom Knight. An architecture for mostly functional languages. In *Proc. ACM Symp. LISP and Functional Progr.*, pages 105–112. ACM, 1986.

[36] D. A. Kranz et al. Orbit: An optimizing compiler for Scheme. In *Symposium on Compiler Construction*, pages 219–233. ACM SIGPLAN, June 1986.

[37] B. Liskov et al. *CLU Reference Manual*. Springer-Verlag, 1981.

[38] Barbara Liskov and Robert Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[39] John McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*. North-Holand, 1963.

[40] James G. Mitchell et al. Mesa language manual, version 5.0. Technical Report CSL-79-3, Xerox PARC, Systems Development Department, 1979.

[41] David Moon. Architecture of the symbolics 3600. In *Proc. of the 12th IEEE International Symposium on Computer Architecture*. IEEE, April 1985.

[42] Peter Nuth. Communications patterns in a symbolic multiprocessor. Master's thesis, Mass. Inst. of Technology, May 1987.

[43] United States Department of Defense. *Reference Manual for the Ada Programming Language*. ANSI/MIL-STD-18151A-1983. Springer-Verlag, New York, 1983.

[44] S. M. Ornstein, W. R. Crowther, M. F. Kraley, R. D. Bressler, A. Michael, and F. E. Heart. Pluribus — a reliable multiprocessor". In *AFIPS Conference Proceedings*, volume 44, pages 551–559, May 1975.

[45] Don Oxley. Embedding Prolog in Scheme. Unpublished manuscript.

[46] G. Pfister et al. The ibm research parallel processor prototype (rp3): Introduction and architecture. In *Int'l. Conf. on Parallel Processing*, pages 764–771, St. Charles, Ill., August 1985.

[47] Jonathan Rees, Norman Adams, and James Meehan. The t manual. Technical report, Yale University, January 1984. Fourth edition.

[48] Jonathan Rees and William Clinger (*editors*). Revised$^3$ report on the algorithmic language Scheme. *ACM Sigplan Notices*, 21(12), December 1986. Also available as MIT AI Memo 818a.

[49] R. D. Rettberg et al. Development of a voice funnel system, design report. Technical Report BBN Report 4098, Bolt Beranek and Newman Laboratories, Inc., August 1979.

[50] Guillermo Juan Rozas. A computational model for observation in quantum mechanics. Master's thesis, Mass. Inst. of Technology, 1986.

[51] Richard Stallman. According to Gerald J. Sussman, the `dynamic-wind` procedure was first proposed by Richard M. Stallman. No reference appears to be available.

[52] Guy Lewis Steele Jr. Rabbit: A compiler for Scheme. Master's thesis, Mass. Inst. of Technology, 1978.

[53] Guy Lewis Steele Jr. *Common LISP The Language*. Digital Press, 1984.

[54] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[55] Thinking Machines Corp., Cambridge, MA. **Lisp Programmer's Manual*, 1987. Proprietary document.

[56] R. Thomas and W. Crowther. The uniform system approach to programming the butterfly. Technical report, Bolt Beranek and Newman Laboratories, Inc., March 1986.

[57] Franklyn Turbak. Grasp: a visible and manipulable model for procedural programs. Master's thesis, Mass. Inst. of Technology, 1986.

[58] Janet H. Walker et al. Praxis language reference manual. Technical Report 4582, Bolt Beranek and Newman, Inc., 1981.

[59] A. Wang. Exploiting parallelism in lisp programs with side effects. Bachelor's thesis, Mass. Inst. of Technology, May 1986.

[60] Henry M. Wu. Performance evaluation of the Scheme86 and hp precision architectures. Master's thesis, Mass. Inst. of Technology, 1987.

[61] Ramin Zabih, David McAllester, and David Chapman. Nondeterministic lisp with dependency-directed backtracking. In *Proc. Sixth National Conference on Artificial Intelligence*, 1987.

# Appendix A

# Fundamentals of Scheme

This chapter serves a dual purpose, both introducing the Scheme language itself and describing those aspects of the language particularly relevant to the implementation of a parallel programming system based around the language.

Providing a tutorial on the language for those unfamiliar with programming, or programming in Lisp languages, would require more space than can be provided here. A number of introductory texts exist. The book by Abelson and Sussman[2] is referred to frequently in the main body since it does an outstanding job of introducing a number of important programming concepts using the Scheme language. The book by Eisenberg[16] is more accessible to people with less formal background, but does not emphasize the same range of concepts. The combination of the two should be ample to introduce anyone interested in the language to all of the ideas needed to understand Scheme as used in this report.

For those readers who are familiar with computer science but not Lisp in particular, Section A.1 serves as a fast overview of some of the important features of the Lisp languages. For those familiar with Lisps in general, but not the Scheme dialect itself, the most important special features of Scheme are summarized in Section A.2. Two of these special features are used so heavily, both by Scheme programmers generally and in the procedures presented here, that they are discussed separately in sections A.3 and A.4. The final section, A.5, provides an introduction to the notion of a programming language and its relation to a complete programming system, especially in the MIT Scheme dialect.

## A.1 Introduction to Lisp

The Lisp languages are distinguished by a number of shared attributes. For those not familiar with the Lisp culture, here is a brief (and doubtless biased) list of the items most unlike other programming cultures. The author makes no claim that these are all advantages of the language family; they do appear, however, to be facts.

**(Lack of) Syntax**

Perhaps most distressing to those unfamiliar with the language is its apparent lack of syntactic constructs. This is actually a misperception: there is a good deal of syntax, but it is all superficially similar. Lisp is an *expression* language, as opposed to the more common *statement* languages. Lisp expressions are of four kinds:

1. Constant expressions. For the purposes of this report, the constants are numerals, the two booleans `#f` (false, also written as `'()` or `()` since, for historical reasons, MIT Scheme uses the same object to denote the empty list) and `#t` (true), and strings surrounded by double quotation marks (`""`).
2. Variables. Denoted by a sequence of alphanumeric characters and certain additional characters (most importantly for this report "`!`", "`?`", and "`-`"). Historically, Lisp systems have used a system of dynamic binding for finding the value of a variable. Scheme, however, uses a strictly lexical scoping system (the same system used by the Algol family of languages).
3. Special forms. Most of the syntax of the language is embodied in what are known as the "special forms" of Lisp. There is a short list of these forms, and the ones used in this report are shown in figures A.1 and A.4 (pages 194 and 198, respectively). A special form is signalled by an expression beginning with a left parenthesis followed by a keyword (the name of the special form). The form ends with the close parenthesis matching the opening one of the form.
4. Combinations, applications, or procedure calls. These names are interchangeable, although combination tends refers to the syntactic construct, the others to the semantics. These superficially

resemble special forms, a fact that confuses people learning the language. A combination is, like a special form, surrounded by parentheses. If the first sub-expression following the left parenthesis is *not* the keyword of a special form, then the expression is a combination. In Scheme, the value of a combination is found by evaluating each of the subexpressions (the order is not specified). The value of the first subexpression must be a procedure or continuation, and it is then applied to the values of the other subexpressions.

**Storage allocation**

Programmers do not worry about allocating or releasing memory in Lisp. Operations allocate memory as needed by creating data structures or other objects. Data structures are made using `cons` to create an ordered pair, `list` to create a singly linked list out of ordered pairs, or `vector` to create a one-dimensional vector. Memory is reclaimed by recycling those parts that cannot be reached using the ordinary operations of the system. This process of recycling memory is known as garbage collection. The operations `car` and `cdr` select the first and second components of an ordered pair (respectively), and `vector-ref` selects entries in a vector (zero based). The contents of a data structure can be modified using the mutators `set-car!`, `set-cdr!`, and `vector-set!` as appropriate. Lists can be copied to make vectors using the procedures `list->vector` and vectors copied to make lists using `vector->list`. There are other data objects with their associated mutators and selectors, introduced as needed throughout the main body.

**Functional style**

Algol family languages tend to rely on sequential evaluation of commands as the primary control construct. The Lisp family of languages uses applicative order evaluation as its primary control construct: the subexpressions of a combination are evaluated before the body of the procedure is evaluated. The Scheme special form `begin` is used to force sequential evaluation of a series of expressions when needed.

**Incremental program construction**

Historically, one of the most notable features of Lisp systems has been the emphasis on an environment supporting interactive construction

of programs. In particular, the Lisp language does *not* have the notion of a program. Rather, it deals exclusively with procedures having a standard function call interface. The notion of a program in other languages is supported through an informal (human based) system of specifying certain top-level entry points into a related group of procedures. The Lisp system supplies a standard interaction environment (the MIT Scheme system has the "read – eval – print" interface) for defining and calling procedures.

**Dynamic type checking**

A common misperception about the Lisp family is that they are typeless languages. In fact, they are far from typeless, although they require runtime checking of data types. Unlike the Algol family of languages, Lisp places data types on the objects manipulated by the language, *not* on the variables of the language itself. Usually no attempt is made to statically type variables. Instead, the primitive operations themselves examine their operations to determine whether they have the appropriate data types. While the Lisp languages cannot strictly be said either to support or not support user-defined abstract data types, it is certainly fair to say that this discipline is more one imposed at the programmers' discretion than by the language or programming system.

**First-class objects**

One way of examining programming languages is to ask what types of objects are treated as "first-class citizens" of the language. A first-class object can be (a) stored as the value of a variable, (b) passed as an argument to a procedure, (c) returned as the value of a function, and (d) stored in data structures. A running theme of the evolution of the Lisp family has been an attempt to make more objects first-class. In the MIT Scheme dialect, all objects are first-class with the exception of special forms (syntax or macros). In particular, binding environments, procedures, and continuations are first-class objects. The ramifications of these latter two are explored in sections A.3 and A.4 respectively.

## A.2 Scheme as a Dialect of Lisp

Scheme evolved as a separate dialect of Lisp with three distinguishing features. Scheme was the first Lisp to use lexical scoping for its variable binding mechanism. This major break with the Lisp tradition has allowed Scheme to experiment with the block structure available in the Algol family of languages. Scheme has been influenced by a very strong minimalist tradition from formal mathematics. It has fewer syntactic constructs than most other dialects and a much smaller standard library of procedures. Finally, it emphasizes consistency and simplicity in the language, relying on compile-time analysis for translation of the simple constructs into efficient implementation.

Readers familiar with CommonLisp[53] will notice the following major differences:

- Scheme does not support "special variables." Thus, special forms such as `symbol-value` and `symbol-function` do not exist in Scheme. Section 3.3.3 discusses the use of fluid variables in MIT Scheme. These provide similar properties but are strictly lexically scoped. The `define` special form in Scheme always introduces a binding in the current lexical contour, and is thus different from the `defun` special form of CommonLisp.

- Scheme has no iteration constructs. Instead, all implementations of Scheme are required to exhibit fully reductive behavior (sometimes called tail-recursion). That is, each sub-expression of an initial expression can be syntactically determined to be either a subproblem or a reduction of the full expression. No stack space is required to handle the evaluation of a reduction sub-expression. The result of this requirement is that some (syntactically) recursive procedures may be executed in an iterative manner. This area is described in more detail in Section A.4.

- Scheme treats all sub-expressions of a combination in an identical manner. In particular, the first sub-expression is evaluated using the same evaluation rules it uses for the other sub-expressions. There is no separate function name binding environment. None of the associated special forms exist (`funcall`, `flet`, *etc.*).

| Name | Description |
|---|---|
| and, or | (`and` $exp_1$ ... $exp_n$), (`or` $exp_1$ ... $exp_n$)<br>Short-circuit (or conditional) `and, or`. |
| begin | (`begin` $exp_1$ ... $exp_n$)<br>Sequentially evaluate *exp*s, returning value of $exp_n$. |
| cond | (`cond` ($pred_1$ . $conseq_1$) ...))<br>McCarthy's conditional construct. |
| define | (`define` *name exp*)<br>Add binding for *name* to *exp* in current environment. |
| | (`define` (*name* . *params*) . *body*) expands to<br>(`define` *name* (`lambda` (*params*) . *body*)). |
| delay | (`delay` *exp*)<br>Memoized delay of *exp* until it is `forced`. |
| if | (`if` *predicate consequent alternative*)<br>If–Then–Else construct. |
| lambda | (`lambda` ($param_1$ ...) . *body*)<br>Create a procedure (closure). |
| let | (`let` (($name_1$ $exp_1$) ...) . *body*)<br>Evaluate *exp*s, then bind *name*s to these values, then evaluate *body*. Evaluation and binding order are *not* specified. |
| letrec | (`letrec` (($name_1$ $exp_1$) ...) . *body*)<br>Similar to `let`, but *exp*s can reference the *name*s. |
| quote | (`quote` *expression*) or '*expression*<br>Return *expression* without evaluating it. |
| set! | (`set!` *name expression*)<br>Assign value of *expression* to lexical variable *name*. |

**Note:** For a full definition of the standard Scheme language, please refer to [48].

Figure A.1: Standard Scheme Syntax Used in Report

- The special form `lambda` in Scheme evaluates to a procedure object — the "closure" of CommonLisp. There is no need for the `function` special form (or its abbreviation, "`#'`").

- The special form `set!` in Scheme is roughly equivalent to the `setq` form of CommonLisp but it will *not* create a binding. Scheme has no equivalent for the `setf` macro, but uses the primitive operations `set-car!`, `set-cdr!`, and `vector-set!` explicitly.

- Scheme guarantees neither the order of evaluation of operands nor the order of binding in a procedure call. For sequential evaluation, the special form `begin` must be used. For sequential binding, either the macro `let*` or an equivalent program structure must be used.

### A.2.1 The Standard Scheme Dialect

There is a standard Scheme dialect supported by a number of different implementations. This dialect is described in a short document[48] that includes a complete denotational semantics for the language. A very brief summary of the syntactic constructs used in the examples of this report is included in Figure A.1.

The procedures used in this report which either differ from CommonLisp or are not frequently encountered are shown in Figure A.2. In addition, a smattering of common Lisp procedures are used without explanation (such as `car`, `cdr`, `eq?`, *etc.*).

Figure A.3 shows a simple program demonstrating some of the syntax of the Scheme language and most of the procedures described in Figure A.2.

### A.2.2 MIT Extensions to Scheme

The purpose of the Scheme standard[48] is to provide a language base. This base is then subject to local extensions. The MIT Scheme system contains a number of extensions used throughout this report, shown in Figure A.4. These extensions provide three features not conveniently available in the standard dialect.

**First-class environments**

The MIT dialect of Scheme has been extended to allow lexical binding

| Name | Description |
|---|---|
| apply | (`apply` *procedure argument-list*)<br>Call *procedure* with the arguments in the *argument-list*. This "spreads" the argument list before calling the procedure. |
| display | Print its one argument out in a human- (rather than machine-) readable format. |
| for-each | (`for-each` *procedure list*)<br>Applies the *procedure* to each item in the *list*. Procedures are invoked "for effect only," thus the value returned is unspecified. This is MacLisp's `mapc` procedure. |
| map | (`map` *procedure list*)<br>Similar to `for-each`, but returns a list of the results produced by *procedure*. This is MacLisp's `mapcar` procedure. |
| newline | Begin a new line on the primary output device. |

Figure A.2: Standard Procedures Used in Report

environments to be treated as first-class objects. Environments are made available to a Scheme program by using the `the-environment` special form or one of a number of macros built using it. These return as their value the current environment. The value of a variable in a given environment can be found using the `access` special form, and the syntax of the `set!` special form allows the use of an `access` special form in place of a variable name. Finally, the procedure `eval` can be used as a "compile and run" operation. It receives both a program to be executed (typically represented as a list) and a binding environment for execution.

**Stream processing**

Section 2.5 introduces the notion of stream processing, a very powerful method for dealing with simulations in a functional system. MIT Scheme provides only one simple additional syntax for dealing with streams: `cons-stream` expands as shown in Figure A.4. Using this syntactic extension along with the standard `delay` special form and

```
(define (display-scaled-matrix matrix scale)
  (define (scale-row row)               ; (1)
    (map (lambda (item) (* item scale)) row))
  (define (display-row row)             ; (2)
    (define (each-item item)            ; (3)
       (display item)
       (display " "))
    (for-each each-item row)
    (newline))
  (for-each                             ; (4)
   (lambda (row) (display-row (scale-row row)))
   matrix))
```

**Notes:**

1. The procedure `scale-row` takes a row of a matrix (stored as a linked list) and produces a list of scaled values using the procedure `map`.
2. The procedure `display-row` uses `for-each` to display each item in a given input row, then uses `newline` to start a new line of output.
3. `Each-item` is called once for each item in the matrix. It displays the item followed by a space.
4. This is the main body of the procedure `display-scaled-matrix`. It uses `for-each` to display the scaled rows of the matrix. This procedure assumes that a matrix is stored as a list of lists (rather than the more traditional vector of vectors).

Figure A.3: A Simple Scheme Program

| Name | Description |
|---|---|
| access | `(access` *name* $exp_1$`)` and<br>`(set! (access` *name* $exp_1$`)` $exp_2$`)`<br>The variable *name* in the environment $exp_1$. |
| cons-stream | `(cons-stream` $exp_1$ $exp_2$`)` expands into<br>`(cons` $exp_1$ `(delay` $exp_2$`))` |
| fluid-let | `(fluid-let ((`$name_1$ $exp_1$`) ...) .` *body*`)`<br>Evaluate all of the *exp*s, then modify the bindings of the existing (lexical) variables (*name*s). Evaluate the *body*, then restore the original values of the variables. Interacts with `call-with-current-continuation` to guarantee that values are restored even when exit is performed using continuations. See description in Section 3.3.3. |
| the-environment | `(the-environment)`<br>Return the current lexical environment as a Scheme object. Used with `access`. |

Figure A.4: MIT Syntax Used in Report

primitive operations `car`, `cdr`, and `force`, the entire array of stream processing functions are easily defined.

**Fluid variable binding**
Section 3.3.3 discusses MIT Scheme's fluid variable concept. The essential idea is to allow a single lexical variable to represent different values depending on the control path taken to reach the current execution state. The special form `fluid-let` is introduced in MIT Scheme to make the addition of fluid bindings convenient.

In addition to these language extensions, MIT Scheme is largely distinguished by its particular choice of runtime environment. This is discussed in Section A.5.

## A.3 Procedures as Objects

While most modern dialects of Lisp, including CommonLisp and Scheme, allow procedures (also known as closures) to be treated as first-class objects, the practice is rare in most other languages and unfamiliar to most programmers. This report is certainly not the place to introduce the power and flexibility this allows (see, for example, references [2] and [54]), but an ability to at least understand programs written using this ability is essential to a number of the examples, especially in Chapter 4. The essential points to bear in mind can be summarized in the following "mantras" of the Scheme language:

- Every expression has a value.
- The value of a `lambda`-expression is a procedure.
- The *free* variables of a procedure receive their values from the environment in which the procedure was created (i.e. lexical scoping).
- The *bound* variables of a procedure receive their values from the argument list in the call to the procedure.

Because variables are lexically scoped and procedures can be created "on the fly" using `lambda`-expressions, procedures are often used to encapsulate expressions whose value is to be computed at a later time. Procedures used

in this manner are known, for historical reasons, as **thunks**. This term is used somewhat loosely in this report, and does not necessarily imply that the procedure has no formal parameters (the traditional usage).

```
(define (make-incrementer! amount)
  (lambda (increment)
    (set! amount (+ amount increment))
    amount))

(define make-incrementer!
  (lambda (amount)
    (lambda (increment)
      (set! amount (+ amount increment))
      amount)))

(define (make-incrementer! amount)
  (define (do-increment! increment)
    (set! amount (+ amount increment))
    amount)
  do-increment!)
```

Figure A.5: Three Ways to Define a Procedure-Generator

The sample program shown in Figure A.3 demonstrates a very common use of `lambda`-expressions to create a procedure for use as an input argument. Figure A.5 shows three equivalent ways of defining a very simple procedure-generating procedure, and Figure A.6 is a demonstration of its use. If the syntax or operation of this procedure is difficult to understand, a thorough treatment of these topics is provided in the chapters 1 and 3 of reference [2].

## A.4 Continuations as Objects

There is one type of object available in Scheme that does not exist in other programming languages: the **continuation**. Continuations were introduced as part of the formal description of programming languages through denotational semantics. They are used there to provide a mathematical handle for

```
⇒ (define incrementer-a
    (make-incrementer! 3))    ⇝   INCREMENTER-A
⇒ (define incrementer-b
    (make-incrementer! 3))    ⇝   INCREMENTER-B
⇒ (define incrementer-c
    incrementer-a)            ⇝   INCREMENTER-C
⇒ (incrementer-a 5)           ⇝   8
⇒ (incrementer-b 3)           ⇝   6
⇒ (incrementer-b 2)           ⇝   8
⇒ (incrementer-a 2)           ⇝   10
⇒ (incrementer-c 3)           ⇝   13
⇒ (incrementer-a 2)           ⇝   15
⇒ ((make-incrementer! 3) 2)   ⇝   5
```

Figure A.6: Use of `Make-Incrementer!`

expressing control flow in programs (see Stoy[54] for an excellent overview of the mathematics involved).

### A.4.1 Continuations: An Introduction

The main idea behind a continuation is that every expression or statement in any language is provided with a function it is expected to call when it has completed its own processing. The notion of "returning a value" is thus eliminated and replaced with the simpler (mathematically) notion of calling a function.

For example, the meaning (semantics) of the Scheme statement

(if *predicate consequent alternative*)

can be described as follows. First, each of the three sub-expressions is converted into its equivalent semantic function (i.e. the mathematical function that computes the same value the sub-expression itself computes). Recall that *every* semantic function, including these three, will receive a continuation as one of its arguments.

The semantic function of the `if` expression itself, then, receives a continuation, $\kappa$. Since the first thing an `if` statement must do is to evaluate the predicate expression, the semantic function for the `if` statement will call the semantic function for the `predicate` sub-expression. But this requires a continuation expressing the work to be performed after the value of the predicate has been computed. If we call the semantic functions for the consequent and the alternative $\texttt{consequent}_{sf}$ and $\texttt{alternative}_{sf}$, respectively, then this new continuation will receive the value of the predicate sub-expression and call either $\texttt{consequent}_{sf}$ or $\texttt{alternative}_{sf}$.

```
(define (if_sf-generator predicate_sf consequent_sf alternative_sf)
  (lambda (κ)
    (define (κ' value-of-predicate)
      (cond ((eq? value-of-predicate #f)
             (alternative_sf κ))
            (else (consequent_sf κ))))
    (predicate_sf κ')))
```

Figure A.7: The Semantic Function for `if`

To help readers who are not familiar with the notion of continuations, Figure A.7 shows, using Scheme notation, a procedure that generates the semantic function for an `if` expression given the semantic functions for the sub-expressions. The most important point to notice is that $\kappa$' uses $\kappa$, the continuation for the `if` expression itself, as the continuation for use by the consequent and alternative sub-expressions. This corresponds to the fact that the predicate sub-expression in Scheme is a "subproblem" of the original expression (there is more work to be performed after its value is calculated) whereas the consequent and alternative sub-expressions are "reductions" of the original — their value *is* the value of the overall expression. This idea will be examined again in Section A.4.3.

### A.4.2 Continuations in the Scheme Language

In 1978, Steele[52] introduced the idea of using continuations as a way of analyzing Scheme programs during the compilation process. He explored this

possibility and demonstrated the idea by building the first working compiler for the Scheme language. He also introduced a programming style, called "continuation passing," that makes the continuations explicitly visible within the program itself. Exploiting Scheme's ability to treat procedures as objects, Steele showed a simple procedure for converting any procedure written in Scheme into an equivalent procedure that does not return a value but rather invokes an explicitly supplied continuation.

In Steele's system, an expression's continuations was simply a procedure of one argument (the value computed by the expression). The fact that this simple conversion process can be applied to any program demonstrated that the Scheme language was sufficient to implement continuations without any extensions. The availability of continuations within the language itself adds a vast amount of expressive power since it encompasses all the forms of control structure that can be described using the mechanics of denotational semantics. The ramifications of the ability to program using continuations are still being explored by the Scheme community.

Unfortunately, the procedures resulting from the conversion process are often difficult to understand. The argument that continuations need not be added to the Scheme language is factually correct. It has as much validity as the statement that "the names of formal parameters can be chosen arbitrarily." And both of these arguments have the same basic flaw: the form in which a statement is written can have a major impact on how easily a *person* can understand the statement. While understanding that the language does not inherently need any extensions to support programming using continuations, the Scheme community nevertheless chose to add one operation to the language to ease the chore.

Instead of rewriting a program so that all continuations are made explicitly visible in the text, Scheme supports an operation with the somewhat daunting name `call-with-current-continuation`. Rather than name every continuation and pass them explicitly as procedures, this operation allows Scheme programs to acquire a name for just those continuations used in a way not supported directly by the syntax of the language. The continuation objects made available using this operation are, from the point of view of a programmer on a serial machine, just another kind of procedure object. They expect a single argument, ordinarily the value computed by the expression whose continuation they represent.

A simple use of `call-with-current-continuation` is demonstrated in

```
(define (list-of-square-roots list)
  (call-with-current-continuation
   (lambda (early-exit)                          ; (1)
     (map (lambda (object)                       ; (2)
            (cond ((not (number? object))
                   (early-exit 'NOT-A-NUMBER))   ; (3)
                  ((negative? object)
                   (early-exit 'NEGATIVE))       ; (4)
                  (else (sqrt object))))         ; (5)
          list))))
```

**Examples of Use:**

```
⇒ (list-of-square-roots '(4 9 16))  ⇝ (2 3 4)
⇒ (list-of-square-roots '(4 x 16))  ⇝ NOT-A-NUMBER
⇒ (list-of-square-roots '(4 9 -16)) ⇝ NEGATIVE
```

**Notes:**

1. This procedure (the value of the `lambda` expression) is called with `early-exit` bound to a continuation object.
2. Recall that `map` applies a procedure to each element of an input list and creates a list of the resulting values.
3. First premature exit. Notice that the continuation object, `early-exit`, is treated exactly like a procedure of one argument. The value of that argument is returned as the value of that call to `call-with-current-continuation` that created the continuation object.
4. Second premature exit. See note 3, above.
5. Ordinary exit. The square root is returned using the ordinary (implicit) continuation — in this case one created somewhere inside of the `map` procedure.

Figure A.8: Simple Use of `Call-With-Current-Continuation`

Figure A.8. Here the continuation is used to support a "non-standard exit" similar to the CommonLisp `catch` and `throw` operations or the `exit` procedure of UCSD Pascal.

Notice the somewhat unusual interface provided by the `call-with-current-continuation` primitive. It takes one argument, a procedure. This procedure is called and passed its own continuation as its argument. This continuation, of course, is the one corresponding to returning from the call to `call-with-current-continuation` itself. Since Scheme requires every procedure to return a value, the continuation object itself expects one argument: the value to be returned from `call-with-current-continuation`.

> Calling a continuation object has the effect of returning from the call to `call-with-current-continuation` that created the continuation.

It is important to bear in mind that the operation `call-with-current-continuation` is nothing more than a way of avoiding the inconvenience of syntactically rewriting Scheme procedures. In particular, the continuation object created in this way *is* a first-class object. It has indefinite extent and *can* be used multiple times, just like an ordinary procedure. This implies, of course, that a call to `call-with-current-continuation` may return more than once: an unusual but useful possibility.

### A.4.3 Continuations: One Implementor's View

There are a number of different implementations of the Scheme language, and each of them has its own techniques for handling first-class continuation objects. The description here is based on the MIT interpreter since it forms the basis of MultiScheme.

Because of Scheme's lexical scoping and consequent block structure, its implementation in many ways resembles that of the more familiar Algol family of languages (Pascal and Ada, for example). Common implementations of these languages rely (at least conceptually) on a stack to support recursive procedure calls. Scheme differs in substantial ways, however, and the simplest way to describe Scheme's handling of continuations is by contrast with a stack-based implementation of one of these languages. The three major differences are summarized in Figure A.9. A typical Pascal implementation

(such as the UCSD implementation) uses a single stack for three different kinds of activities.

1. Variable bindings. The values of formal parameters to a procedure are pushed on the stack along with the "static link" to the lexically enclosing stack frame. All references to variables are through this stack area, either to a local variable stored in the frame itself or through the static link to enclosing frames[1]. Return from a procedure includes popping the parameters off the stack.

2. Return addresses or "dynamic link." Since procedures are entered, executed, and then return to the caller, the stack is a natural data structure for storing this call chain. The address where control will return when the current procedure completes is therefore pushed on a stack when the call begins and is popped when the procedure ends.

3. Intermediate storage while evaluating sub-expressions. In calculating the value of a complicated arithmetic expression, for example, the values of sub-expressions are pushed onto the stack as they are computed and then popped back off when they are combined with the values of other sub-expressions.

The Scheme language is more flexible than most Algol family languages, and this flexibility comes at a price. Scheme's ability to create and return a procedural value at run time (as in Figure A.5) requires that some variable bindings have a lifetime extending beyond the execution time of the procedure that created them. Calls to these procedures cannot, therefore, use the procedure call stack to store variable bindings. Instead, in the general case, bindings must be stored in the heap (the same place where `cons` cells are allocated). This leads to the first statement in Figure A.9.

One of the unusual features of Scheme, mentioned in Section A.2, is that there are no iteration constructs. All implementations must correctly implement the notions of subproblem and reduction mentioned earlier. In order to understand the impact of this requirement it is important to understand the relationship between these concepts and the use of the stack. For the purposes of this report, we can use the following informal definitions:

[1]Sometimes a "display" is used to access the enclosing frames rather than a chain of static links. The frames, however, are still stored on the stack.

1. Values of variables are *not* (in general) stored on the Scheme stack.
2. Procedure call in Scheme does *not* push entries onto the stack. Subproblems push entries on the stack, but reductions do not.
3. The current continuation *is* the stack, and it (or a copy) can become directly visible to programs.

Figure A.9: Scheme's Stack vs Pascal's Stack

**Subproblem:**
A sub-expression is a subproblem of an original expression if the value of the full expression requires finding the value of the sub-expression *and then* doing additional work. Alternatively, a subproblem is a sub-expression evaluated using a continuation *other than* the one supplied for the overall expression.

**Reduction:**
A sub-expression is a reduction of an original expression if the value of the full expression *is* the value of the sub-expression (should that sub-expression ever by evaluated). Alternatively, a reduction is a sub-expression evaluated using *the same* continuation as that of the original expression.

Using these definitions and Figure A.7 we can see that the `predicate` sub-expression of an `if`-expression is a subproblem (it has $\kappa$', not $\kappa$, for its continuation), while the `consequent` and `alternative` are reductions (they have the original $\kappa$ as their continuation). A similar analysis, based on the formal semantics given in [48], will distinguish subproblems from reductions for all the sub-expressions in the Scheme language. The two most important reductions in the language, leading to the majority of the so-called tail-recursive nature of the language, are

1. The final sub-expression of a `begin` special form. That is, each of

the earlier sub-expressions is a subproblem (using a continuation that causes the evaluation to continue on to the next sub-expression). The final sub-expression, however, is a reduction since its value *is* the value of the entire expression. This reduction is particularly significant since there is an implicit `begin` special form around the body of all procedures (including the bodies of `let`, `letrec`, and `fluid-let` special forms) and the consequents of `cond` special forms.

2. The body of a procedure. That is, after evaluating all of the sub-expressions of a combination, the evaluation of the combination reduces to the body of the procedure.

With this understanding, we can reexamine the original question: what is the relationship between these two concepts and the use of a stack in the implementation. The answer lies in the fact that the continuations are for the most part left implicit in Scheme programs. As long as this is true, an analysis of the language will reveal that the continuations created for subproblems are used in a last-in first-out manner and can therefore be efficiently stored on a stack. This should come as no surprise, since the language has none of the control constructs that would lead to anything other than the ordinary stack-like discipline of procedure call in other languages.

In fact, the MIT Scheme implementation *does* use a stack for storing continuations. Thus, whenever a new subproblem is about to be evaluated a new entry is pushed on the continuation stack, and completion of an evaluation step (where the semantic function would call the original continuation of an expression) pops this continuation off of the stack. This usage of the stack corresponds to the return address use (number 2) described earlier. But there is an important difference. Notice that Scheme pushes an entry onto the stack only for subproblems of an expression, not reductions. Since the body of a procedure is a reduction it follows that, as shown in Figure A.9, procedure call in Scheme does *not* push entries onto the stack.

Instead, it is the position in which the call occurs that determines whether an entry is pushed on the stack. Calls occurring in subproblem position push entries onto the stack; calls occurring in reduction position leave the stack unchanged. It is this property that allows a (syntactically) recursive Scheme program to operate without requiring additional space (i.e. iterate). This is a generalization of the more common tail recursion optimization found in compilers for some languages.

With this terminology and implementation background, we can explain the implementation of `call-with-current-continuation`. Rather than produce a full procedure at every point where the semantics of the language requires a new continuation, the MIT implementation pushes an entry onto a stack. This is fine for all of the ordinary operations of the language, since they require knowing only the immediately following operation to be performed. `Call-with-current-continuation`, on the other hand, must make an applicable object with indefinite extent that can be used to reference this control state at any time in the future.

The first, and simplest, way this can be accomplished is to "package up" the *entire* stack by copying it into the heap. The stack can then be emptied and a single entry pushed onto it. This entry says, in essence, that the object just created must become the stack if control ever returns to this point — and thus the continuation will be copied back from the heap into the stack. This describes the original implementation of continuations in MIT Scheme. Measurements have shown that the stack rarely grows very deep (over 100 entries) even in lengthy (syntactically) recursive programs, although it is quite easy to write a program that allows the stack to grow arbitrarily deep. Thus, the cost of copying the entire stack is not large, and since `call-with-current-continuation` is rarely used, this implementation causes no major problems. Thus, the third major departure from the Algol family of languages is the ability to capture the stack as an object in the programming language. This is shown as the third major difference in Figure A.9.

There have been two changes to the implementation of `call-with-current-continuation`, however, as a result of the work on MultiScheme. As will be discussed in Chapter 4, this primitive is used to support task switching in MultiScheme and its performance is therefore much more important than in MIT Scheme. The first change is an alternative implementation of the interpreter (and compiler interface) allowing the stack to be allocated in the heap and thus eliminates the need to copy the stack at the time the continuation object is made. Instead, it is copied incrementally when control returns to the part of the stack that existed at the time of the call to `call-with-current-continuation`. The second change was the recognition (as explained in Chapter 4) that the continuation objects used for task switch are in fact never reused. Although they have indefinite extent, they are created and used exactly once. A far more efficient implementa-

tion can be made under these circumstances, and an additional primitive, `non-reentrant-call-with-current-continuation` was added precisely to support this operation.

## A.5 MIT Scheme as a System

One traditional method for building a programming system is to divide the task into three parts: language implementation, portable language-based library, and a system-specific library for interfacing to the underlying operating system and external (non-language specific) facilities. The goal of this structure is to facilitate the construction of portable programs (by promoting the use of the portable library) while at the same time allowing efficient access to non-portable features available within a specific implementation. Lisps exhibit a similar structure, although the delineation of the components is frequently far less clear and the portable library tends to contain a wider range of procedures than those of many other systems.

Since standard computer architectures do not provide support for the garbage collection essential to the Lisp language, most Lisp implementations are forced to supply their own memory management routines. But once the Lisp implementation becomes responsible for its own memory management it becomes considerably simpler to support the dynamic allocation of what are frequently supplied only as static objects (tasks, files, and so forth). Thus Lisp systems tend to build very large portable libraries consisting of what would ordinarily be considered "systems code." Furthermore, there is a bias within the Lisp community to provide users of the system with access to a very wide range of implementation decisions within this kind of code. Where possible the system builders provide "hooks" for users to specify their own extensions to a framework provided and maintained along with the rest of the Lisp system.

Lisp systems tend to be built in four parts. There is a core language implemented in some non-Lisp language or the result of cross-compiling carefully written Lisp code. An extensible set of primitive procedures provides the very low-level support needed to implement the rest of the system is generally written in assembly language for the target machine. On top of these a set of system procedures (written in Lisp) provides an interface between the external environment, the core language, and user programs. Finally,

a system supplied library of general utility procedures is written using the other three parts as a base. A large component of this report is devoted to the design and implementation of the interface procedures for a parallel processor system. Since these procedures are closely tied to the underlying language core, the broad outlines of this core must be understood before their implementation can be explained.

### A.5.1 Machine Model or Core Interpreter

The MIT Scheme system is built to run on a virtual machine whose origins are in the Scheme '79[32] and Scheme '81[5] projects. Since neither of these projects (nor their successor hardware projects) has become widely available, a portable interpreter has been implemented to simulate the instruction set of these machines. This interpreter is the basis for the actual implementation of MultiScheme[2].

These machines are based around the ability to manipulate typed objects. In the current implementation all objects consist of two parts: the data type field, and the value field[3]. The value field can contain either immediate data (for example, a short signed integer) or more commonly an address. The machine itself runs a tree-structured instruction set (known as SCode). SCode is little more than a parse tree for the program, using an opcode for each of the Scheme special forms (plus some additional ones for combinations, variables, and constants). These operations are encoded in the data type field. The machine maintains a stack of continuations as described in Section A.4, and has a small set of dedicated registers referenced implicitly by many of the instructions.

In addition to this simple execution engine, there is a linear memory space with a free pointer used to allocate the heap. When this pointer is incremented beyond some specified address, the processor indicates the condition by setting one of a number of interrupt request bits, indicating that a garbage collection cycle should begin as soon as convenient. These bits, along

[2] An MIT Scheme compiler is being currently under development. This compiler accepts as input programs written in the instruction set of these machines, performs a number of optimizations, and generates instructions for a standard computer architecture.

[3] Currently, the data type field is 7 bits wide, the value is 24 bits wide, and there is a 1 bit wide field called the "danger bit." This danger bit is being phased out of the implementation and is not discussed further in this report.

with an interrupt request mask (set under program control), are sampled periodically and cause specific Scheme procedures to be invoked. In addition to this garbage collection interrupt a number of other interrupt conditions are treated in the same way. A priority system is used to handle stacking of these interrupt conditions.

Finally, some instructions can detect error conditions. For example, the variable reference instruction can fail if a variable is unbound in the current environment. In these cases a trap into Scheme code is also invoked. Since these errors are synchronous with instruction execution there is no need for the interrupt bits or priority mechanism. Any instruction that can generate an error is required to "back out" and leave the system in the same state it was in prior to the attempt to execute the instruction. This allows the instruction to be retried later if the error condition is removed. Thus executing an erroneous instruction appears to the Scheme system as though the user had inserted a call to the appropriate error handler immediately prior to the erroneous instruction.

In order to allow the hardware to call Scheme procedures in case of errors or interrupts, the machine and the Scheme runtime system communicate through a region of memory known to both. This area contains the Scheme procedure object to invoke for each interrupt or error condition, as well as other information accessible to both.

### A.5.2 Primitive Procedures

The majority of the core interpreter actually consists of the implementation of a (unfortunately) large number of primitive procedures. These procedures can be roughly divided into three classes.

1. Unimplementable. These primitives support language features and cannot be coded directly in Scheme without resorting to directly reading and writing absolute memory locations. Sample primitives in this category are `garbage-collect`, `general-car-cdr`, `apply`, and `call-with-current-continuation`. The majority of these operations are primitive space allocation or data structure referencing operations.

2. Speed-up. Most of the primitives are supplied only to make the operations faster than they would be if implemented in Scheme. String and

list search operations, case conversion, integer restricted arithmetic, *etc.* are in this category. The theory is that when a "good enough compiler" is available, these will be supplied as part of the runtime system (i.e. written in Scheme) rather than as primitives.

3. External interface. About one fourth of the primitives in the MIT system are used to interface with objects outside of the Scheme world. This includes file operations, date and time handling, and so forth. Most of these primitives are not needed by the Scheme system directly, although users of the system might be disappointed if there were no way, for example, to read or write files.

From the point of view of the core interpreter, the number and composition of the primitives is completely immaterial. They have a standard interface allowing the interpreter to check that the correct number of arguments have been supplied and then turn control over to the primitive itself. The primitives themselves are responsible for type and range checking of any arguments, and they use a well defined interface to signal errors or interrupt conditions. Some primitives also alter the flow of control of the interpreter itself (for example, the `apply` primitive stores information on the continuation stack and then requests that the interpreter proceed on to a procedure application rather than by processing the returned value of the primitive).

### A.5.3 System Code

The third and fourth components of the MIT Scheme system constitute the "runtime system," written in Scheme itself. As might be expected, it really consists of two categories of code. There is a "low level" portion of the system dealing with the details of interfacing with the interpreter core, and an extensive set of utility procedures designed for users of the system.

The runtime system (low level portion) is itself organized as a number of interrelated packages (i.e. lexical environments) of procedures. The following list is incomplete, but describes a number of these packages with particular reference to parts that are impacted by the system changes described in this report.

- System initialization, binary file loading and dumping.

- The `reader` and `printer` handle input and output operations and interaction with the host operating system.
- The `parser` and `unparser` convert between the external (character string) and internal (typed pointer) representations of objects. User operations such as `read` depend on the `reader` to gather characters from the input source, then pass them to the `parser` for conversion to the internal object representation.
- The `syntaxer` and `unsyntaxer` convert between the program (SCode) and surface syntax (list) representations of programs.
- Type dispatching and SCode abstraction allow a convenient interface to the underlying representation of objects and programs.
- The `REP-Loop` package allows the construction and stacking of user interaction procedures. It is tightly coupled with the error and interrupt systems (below). The interaction of this structure with exception handling is discussed in Appendix D.
- The `interrupt` and `error` system respond to traps from the core interpreter. Changes and extensions to the interrupt system (in particular to support the initiation of garbage collection) are discussed in Section 3.1.1. The error system reacts to most errors by using the `REP-Loop` component of the system code. Exception handling is *not* a part of the error system, but is closely related to the `REP-Loop` component as well.
- The `scheduler` is a component of the system added in the conversion from MIT Scheme to MultiScheme. Chapter 4 discusses this component in detail.

## A.6 Summary

The Scheme language as used in this report has a very simple syntax (Section A.1, Figure A.1, and Figure A.4). In addition to the ordinarily encountered procedures of CommonLisp, some of the examples in this report use the procedures described in Figure A.2. The language makes heavy use of

the ability to create and return procedures on the fly (in older Lisp parlance it "solves the upward and downward funarg problems"), and examples of this ability are provided in Figures A.5 and A.6.

The Scheme dialect, but not CommonLisp, provides the ability to convert the otherwise implicit continuation for an expression into an object of the language. This allows control structures of arbitrary complexity to be built within the language. Figure A.8 shows one very simple use of this ability. This usage is the key to understanding the task switch of MultiScheme (discussed in Chapter 4). The implementation of continuations in the MIT Scheme system is based on a stack model and the key ideas of reductions and subproblems as presented in Section A.4.3.

Finally, the MIT Scheme implementation is layered in four constituent parts. There is a core interpreter (Section A.5.1), an extensible set of primitive procedures (Section A.5.2), system code (Section A.5.3) and a library of utility procedures. Errors and interrupts are reflected into the system code by calling Scheme procedures provided as part of the standard system. A set of hooks allows users to customize the Scheme runtime system by supplying their own code for use in a variety of well defined circumstances.

# Appendix B

# Implementation of the Pipeline

This appendix includes the code used to implement the pipeline example of Section 5.3. This code is presented complete and, with the exception of additional comments, unedited.

## B.1 Locks for Serializing Access

Recall that `(set-car-if-eq?! a b c)` is an atomic operation that either operates like `(set-car! a b)` (if `(car a)` is currently `c`), or it does nothing. It returns `a` if the modification takes place and `'()` if not.

```
(define (lock obj)                              ; Internal
  (if (null? (set-car-if-eq?! obj 'LOCKED 'UNLOCKED))
      (lock obj)))
(define (unlock obj) (set-car! obj 'UNLOCKED)) ; Internal

(define (make-lock (make-lock value) (cons 'UNLOCKED value)))
(define lock.value cdr)
(define set-lock.value! set-cdr!)
(define (atomically object procedure)
  (lock object)
  (let ((result (procedure)))
    (unlock object)
    result))
```

## B.2 General Utilities

```
(define (make-placeholder)
  ((access make-future scheduler)
   'MAKE-PLACEHOLDER 'MAKE-PLACEHOLDER "Waiting Forever"))

(define (make-list count object-generator)
  (let loop ((answer '())
             (count count))
    (if (= count 0)
        answer
        (loop (cons (object-generator) answer)
              (-1+ count)))))

(define (make-vector count obj-gen)
  (list->vector (make-list count obj-gen)))

(define (start-pipeline pipeline vector)
  (if (and (pair? pipeline)
           (eq? (car pipeline) 'INPUT-NODE))
      (vector 'MESSAGES (cdr pipeline))
      (error "START-PIPELINE: not an input node" pipeline)))
```

Refer to procedure `make-pipe-vector` in Section B.6 to understand how pipeline vectors handle the messages operation.

## B.3 Hash Tables

One and two-dimensional tables are provided, based on hash numbers generated using the interning service described in Section 2.2.2. As mentioned there, this implementation should really use weak cons cells, but I was lazy. It is important that the tables not retain the objects entered in the table, since there is no code to remove entries when they are no longer needed. That should have been implemented, too.

```
(define (make-table) (make-lock '()))

(define (hash-add table hash-code adder)
  (let ((first-try (assq hash-code (lock.value table))))
    (if first-try
        (cadr first-try)
        (atomically table
          (lambda ()
            (let ((second-try (assq hash-code
                                    (lock.value table))))
              (if second-try
                  (cadr second-try)
                  (let ((result (adder)))
                    (set-lock.value! table
                     (cons (list hash-code result)
                           (lock.value table)))
                    result))))))))

(define (1d-add table entry adder)
  (let ((hash-code (object-hash entry)))
    (hash-add table hash-code adder)))

(define (2d-add table first second adder)
  (let ((first-hash (object-hash first))
        (second-hash (object-hash second)))
    (let ((sub-table (hash-add table first-hash
                               (lambda () (make-table)))))
      (hash-add sub-table second-hash adder))))
```

## B.4 Simple Pipeline Constructors

The pipeline is simply a list of messages to be processed by the objects flowing through the pipe. During construction, a "dangling end" has a message in the `car`, and a `'()` in the `cdr`.

```
; Internal procedures:
; (1) Add a new operation to the end of an existing pipe
(define (next-op! pipe op)
  (let ((result (list op)))
    (set-cdr! pipe result)
    result))
; (2) Join two existing pipes so the first flows into the second
(define (weld! in-pipe out-pipe)
  (set-cdr! in-pipe out-pipe))

; External procedures, see Section 5.3.1:
(define (extend pipe operation) (next-op! pipe operation))

(define (make-input) (list 'INPUT-NODE))
```

## B.5 Complicated Pipeline Constructors

The main pipeline constructors operate by adding a procedural message to the growing pipeline. In the case of fork, the message causes an incoming object to make new objects with copies of its current state, each of which propagates down a different output branch:

```
(define (fork n input)    ; Yields list of n output pipes
  (let ((result (make-list (-1+ n) (lambda () (list 'FORK-START)))))
    (define (fork-operation obj)
      (let ((obj-num (obj 'OBJECT-NUMBER))
            (vector (obj 'VECTOR)))
        (map (lambda (pipe)
               (let ((new-object (obj 'NEW-OBJECT (obj 'COPY-STATE))))
                 (new-object vector obj-num (cdr pipe))))
             result)))
    (cons (next-op! input fork-operation) result)))
```

The remaining constructors all require synchronization of arriving objects. They use tables (constructed from the operators above) to locate corresponding objects from the same vector as they pass through a particular point in the pipeline.

At a join point, the first object corresponding to a vector and offset within that vector proceeds on to the output pipeline after performing the user-specified join operation. Objects arriving from other input pipeline branches for the same vector and offset are forced to halt after they make their state available to the first object:

```
(define (join combiner inputs)
  (let ((count (length inputs))
        (result (list 'IDLE))
        (my-table (make-table)))
    (define (make-operation input-number)
      (lambda (obj)
        (let ((vector (obj 'VECTOR))
              (object-number (obj 'OBJECT-NUMBER))
              (state (obj 'GET-STATE))
              (first-one-through? #F))
          (let ((mates (2d-add my-table vector object-number
                         (lambda ()
                           (set! first-one-through? #T)
                           (make-vector count make-placeholder)))))
            (determine! (vector-ref mates input-number) state)
            (if first-one-through?
                (obj combiner (vector->list mates))
                (obj 'HALT))))))
    (define (loop n inputs)
      (if (null? inputs)
          result
          (begin
            (weld! (next-op! (car inputs) (make-operation n))
                   result)
            (loop (1+ n) (cdr inputs)))))
    (loop 0 inputs)))
```

At output and interact points, each object looks to see if any other objects from the same vector have arrived. If so, it merely stores its state away in

the output structure already created, then halts. If not it creates a data structure for objects that arrive later:

```
(define (add-output pipe)        ; Yields list of answers
  (let ((my-table (make-table))
        (result (make-placeholder)))
    (define (output-operation obj)
      (let ((vector (obj 'VECTOR))
            (object-number (obj 'OBJECT-NUMBER))
            (my-state (obj 'GET-STATE)))
        (let ((result-vector
               (1d-add my-table vector
                 (lambda ()
                   (let ((next-out (make-placeholder))
                         (this-out (make-list (vector 'SIZE)
                                              make-placeholder)))
                     (determine! result (cons this-out next-out))
                     (set! result next-out)
                     (list->vector this-out))))))
          (determine! (vector-ref result-vector object-number)
                      my-state))))
    (next-op! (next-op! pipe output-operation) 'HALT)
    result))

(define (add-interactor pipe interactor)
  (let ((my-table (make-table))
        (result))
    (define (interactor-operation obj)
      (let ((obj-vector (obj 'VECTOR)))
        (let ((data (1d-add my-table obj-vector
                      (lambda ()
                        (let ((l (make-list (obj-vector 'SIZE)
                                            make-placeholder)))
                          (cons (list->vector l) l))))))
          (determine!
            (vector-ref (car data) (obj 'OBJECT-NUMBER))
            (obj 'GET-STATE))
          (obj interactor (cdr data)))))
    (set! result (extend pipe interactor-operation))
    result))
```

## B.6 Creating a Pipe Vector

Pipeline vectors are message receiving objects. They take either a messages message, used to associate the vector with a given pipeline input port (see `start-pipeline`, in the general utilities section), or a size message to report the number of objects in the vector. All of the objects in the vector are created using `make-object`, and they are activated by specifying the vector to which they belong, the offset within that vector, and the messages (pipeline) they are to process:

```
(define (make-pipe-vector elements)
  (let ((message-list (make-placeholder))
        (size (length elements)))
    (define (the-vector message . additional)
      (cond ((eq? message 'MESSAGES)
             (determine! message-list (car additional)))
            ((eq? message 'SIZE) size)
            (else (error "Bad message to vector" message))))
    (define (loop number elems)
      (if (null? elems)
          the-vector            ; Value returned: the vector
          (begin                ; Activate the objects
            ((car elems) the-vector number message-list)
            (loop (1+ number) (cdr elems)))))
    (loop 0 elements)))
```

## B.7 Task Creation and Removal

```
(define *all-objects* (make-lock (list 'OBJECTS)))

(define (make-cell before me after)
  (cons (cons before me) after))

(define (before cell)
  (if (or (future? cell)
          (future? (car cell)))
      (error "Before trouble" cell))
  (caar cell))
```

```
(define (after cell)
  (if (future? cell)
      (error "After trouble" cell))
  (cdr cell))

(define (set-before! cell value)
  (if (or (future? cell)
          (future? (car cell)))
      (error "Set-Before! trouble" cell))
  (set-car! (car cell) value)
  'new-before)

(define (set-after! cell value)
  (if (future? cell)
      (error "Set-After! trouble" cell))
  (set-cdr! cell value)
  'new-after)

(define (kill-task cell)
  (atomically *all-objects*
    (lambda ()
      (let ((before-me (before cell))
            (after-me (after cell)))
        (set-after! before-me after-me)
        (if after-me
            (set-before! after-me before-me)))))
  ((access next scheduler)))

(define (record-task! me)
  (atomically *all-objects*
   (lambda ()
     (let ((after-me (after (lock.value *all-objects*)))
           (before-me (lock.value *all-objects*)))
       (let ((my-cell (make-cell before-me me after-me)))
         (if after-me (set-before! after-me my-cell))
         (set-after! before-me my-cell)
         my-cell)))))
```

## B.8 Making an Object

A simplified version of this procedure was shown in Figure 5.11 on page 152. The complete code for handling all operations is:

```
(define (make-object state copy-state user-code)
  (let ((my-cell)
        (vector)
        (object-number)
        (the-messages (make-placeholder)))
    (define (loop message-list)
      (define (standard-handler m #!optional arg)
        (cond
         ((procedure? m) (m standard-handler))
         ((eq? m 'halt) (kill-task my-cell))
         ((eq? m 'idle) 'IDLED)
         ((eq? m 'new-object) (make-object arg copy-state user-code))
         ((eq? m 'object-number) object-number)
         ((eq? m 'vector) vector)
         ((eq? m 'get-state) state)
         ((eq? m 'set-state!) (set! state arg))
         ((eq? m 'copy-state) (copy-state state))
         (else (set! state
                 (user-code state m (if (unassigned? arg) '() arg))))))
      (standard-handler (car message-list))
      (loop (cdr message-list)))
    (set! my-cell (record-task! (future (loop the-messages))))
    (lambda (the-vector number messages)
      (set! vector the-vector)
      (set! object-number number)
      (determine! the-messages messages))))
```

# Appendix C

# Performance Measurements

The MultiScheme project was not primarily concerned with issues of implementation efficiency, but these are clearly important to any serious parallel programming system. This appendix provides measurements of some of the important operations internal to the MultiScheme interpreter, primarily to provide a baseline for comparison with future versions of the system.

Performance measurement is an area which by rights requires careful planning and a well chosen goal — typically aimed at discovering the importance of one aspect of the implementation to the overall performance of a large system. These measurements are *not* part of such an effort, and their utility is correspondingly limited.

## C.1 The Measured System

The measurements are taken from a simulator for MultiScheme that runs on the Hewlett-Packard Series 9000 Model 320 computer, based on a Motorola MC68020 processor running at 16.67 MHz using a 16-bit memory bus with a large (roughly 100K line) cache between the processor and memory. The simulator is written in C, sharing almost all of the code with the actual implementation of MultiScheme on the BBN Butterfly. The primary differences between the two implementations are:

- The Butterfly is a true multiprocessor, while the HP machine is a standard sequential machine. The simulator can be run with or without timer-driven scheduling interrupts. For these measurement no timer

was used since the examples were either sequential or were deliberately intended to measure the overhead of task switches at controlled times.

- The Butterfly hardware is based on an 8 MHz processor (normally a Motorola 68000 but a Motorola 68020 is available as an option) and a microcoded co-processor to provide a virtual memory system. The co-processor is responsible for all transactions across the Butterfly Switch. References to memory that is physically located with the processor making the reference are faster by a constant factor (which depends on the configuration of the particular Butterfly) than are references to any other memory. Switch transactions take place in 16-bit quantities, but the co-processor can be used to implement atomic 32-bit transfers at the cost of some set-up time. Since the MultiScheme interpreter uses a combination of both atomic and non-atomic transfers, memory reference times are not comparable on the two implementations. Typical memory access times are 2 microseconds for local memory and 6 microseconds for remote memory[7].

- The Butterfly memory management is based on the garbage collector described by Courtemanche[12]. It pre-allocates portions of the address space to each processor, and a garbage collection is initiated when any processor's area is filled. This alters the frequency of garbage collection on the Butterfly by comparison to the HP. None of the performance measurements include garbage collection activity on the HP.

- The queue of tasks awaiting processors is provided using the Butterfly's atomic "dual queue" operations[6] operations. These are simulated using primitive procedures (written in C) for queue manipulation on the HP. Similarly, the primitive procedures `global-interrupt` and `await-synchrony` are simulated on the HP.

- The operating system on the HP machine is HP-UX™, a version of Unix™. The Butterfly runs the Chrysalis™[6] operating system, developed for high-performance data communications applications on a multi-processor.

The effects of these differences are quite difficult to quantify. The author therefore suggests that these numbers *not* be considered to have any validity

with respect to an actual parallel processor implementation. In fact, I anxiously await the release of the BBN Butterfly Lisp system so that a realistic set of measurements can be undertaken.

One further caution: the system being measured is fully interpreted. The MultiScheme system code used during the measurements contains a scheduler that corresponds to the scheduler shown in Chapter 4, but with three significant differences:

1. The scheduler actually measured does not permit the independent creation of tasks and placeholders. In fact, the two data structures described in Chapter 4 are combined in one.

2. The race conditions inherent in the implementations of Chapter 4 are *not* present in the measured version. The scheduler measured runs unmodified on the Butterfly, and no races have been detected.

3. The measured version of the scheduler uses a primitive (written in C) to implement a faster version of `call-with-current-continuation` for use solely in task switching. This new primitive does not permit the continuation it creates to be used more than once, and is known (informally) as `task-catch`. Additionally, part of the code for `spawn-task` (whose implementation was *not* shown in Chapter 4) is written in Scheme, and part is written in C.

## C.2 Measurement Technique

For each item measured, a small section of code was written that exercised the desired operation. Because the clock resolution available was 1 tick every 0.01 seconds, the code was repeated 50 times in sequence and the total time for the 50 repetitions was measured. This group of 50 operations was then repeated 20 times (for a total of 1000 operations), separated by a garbage collection.

Using this technique, calls to the procedure `x` defined by

```
(define (x) 3)
```

were measured for purposes of normalization. A single call to `X`, measured in this way, took 0.360 milliseconds (ms). For comparison procedure calls of 1,

2, 6, and 12 arguments were measured. These took 0.396, 0.396[1], 0.576 and 0.781 milliseconds, respectively.

Because of concern over the accuracy of the clock, the remaining numbers were normalized with respect to the time required to perform a call to this procedure, `x`. This time unit, 0.36 milliseconds, is referred to as a "tick". The table in Figure C.1 summarizes the results.

## C.3 Analysis: The Cost of a `future`

In order to better understand the cost of touching a placeholder that does not yet have a value (131.6 ticks from Figure C.1), a number of additional measurements were made. The cost of the procedure that is used within the scheduler to handle this case was measured (this procedure is shown as `await-placeholder` in Figure 4.10 on page 111). The time required for this procedure, combined with a call to `task-catch`, was 48.65 ticks, so the cost of `await-placeholder` itself is approximately 39.12 ticks. In addition, a measurement of the time required to release the current processor and get it back again (roughly the cost of a call to `saving-state` and then using the underlying task queues to save and restore that state) was 39.8 ticks.

Using these measurements, we can compute the amount of time we would expect a `(touch (future 3))` to take, as shown in Figure C.2. The difference between the measured cost of 131.6 ticks and the computed value of 137.0 ticks (from Figure C.2) is well within the margin of error in these timing measurements, which have been observed to fluctuate by as much as 5% (presumably due to the cost of certain critical Unix background jobs).

[1]The Scheme interpreter is optimized for one and two argument function calls (but not zero arguments), lending credence to this first pair of numbers.

| Code | Time | | Notes |
|---|---|---|---|
| | *(ticks)* | *(ms.)* | |
| `(touch (future 3))` | 131.6 | 47.4 | |
| variable reference | 0.7 | 0.3 | For a local variable |
| `(lambda (x) x)` | 0.2 | 0.1 | Creating a closure |
| `(future 3)` | 19.0 | 6.8 | Queues new task and returns to spawning task |
| `determine!` | 13.0 | 4.7 | |
| `touch` | 1.2 | 0.4 | Placeholder that has a value |
| `touch` | 0.9 | 0.3 | Non-placeholder (e.g. after a garbage collection replaces a placeholder with its value) |
| task dist. | 1.8 | 0.7 | Store and retrieve a value on the task distribution queue |
| enqueue | 5.3 | 2.0 | Add task to those waiting for placeholder's value |
| dequeue | 5.1 | 1.8 | Remove task from those waiting for placeholder's value |
| `call/cc` | 11.0 | 4.0 | Make a continuation using `call-with-current-continuation` and then invoking it |
| `task-catch` | 9.5 | 3.4 | Optimized version of above, used for task switch |

**Note:**
As mentioned in the text, all timings are reported in ticks corresponding to the time required to call a procedure of no arguments. This time was measured as 0.36 milliseconds, and includes the costs of: one lexical variable look-up (for the variable `X`); allocating and initializing an environment frame; saving the current continuation; evaluating a constant; and restoring the previous continuation.

Figure C.1: Internal Timing Measurements

| Item | Time *(ticks)* | Description |
|---|---|---|
| a | 19.0 | Creating the task and placeholder |
| b | 39.1 | Touching the undetermined placeholder and calling `await-placeholder` |
| c | 39.8 | Task switch from parent to child task |
| d | 13.0 | `Determine!` value of placeholder |
| e | 5.1 | Dequeue parent task from queue waiting for placeholder's value |
| f | 39.8 | Task switch from child back to parent task |
| g | 1.2 | Touch of the placeholder after value is known |
| | 157.0 | Sub-total |
| | -20.0 | Item b includes a suspend of the parent task, as does item c. Assuming that the cost of suspension is roughly half the cost of a full task switch, this double count has a cost of roughly 20 |
| | 137.0 | Total |

**Note:**
See the text and Figure C.1 for an explanation of the time unit.

Figure C.2: Components of the Cost of a Future

# Appendix D

# Exception Handling in MultiScheme

Section 3.3 explained how fluid variables can be used to provide storage on a per-task basis. In passing, it was mentioned that the ability to use first-class continuations to exit from the body of a `fluid-let` can lead to complications, and the implementation mechanism described in Section 3.3.4 correctly handles these difficulties. This implementation was newly introduced in MultiScheme to replace a mechanism based on the notion of twin dynamic state and control state introduced in MIT Scheme following the work of Hanson and Lamping[26].

These notions are useful beyond their ability to implement fluid variables; they provide the base on which exception handling mechanisms can be built. This section describes these twin notions, how they interact, and how they could be used in MultiScheme to build the kind of exception handling facilities available in other languages. MultiScheme, at present, provides only the basic support for control and dynamic state spaces. None of the applications developed so far, nor the system itself, have had need of an exception handling mechanism. As a result, this section describes a basic mechanism and several ways to use that mechanism. It provides no final system designer's choice — only application of the mechanism to real problems will provide the insight needed to make such a choice.

## D.1 Twin Notions

The addition of first-class continuation objects to Scheme introduced an unprecedented amount of power into the language. Control constructs such as `setjmp` and `longjmp` of many C implementations[30], co-routines, and even multiprocessing systems can be conveniently expressed. The control state of the system can be captured at any point in time and reactivated at any other time, and even reactivated multiple times. An excellent and provocative use of this last ability is demonstrated by Rozas[50], where the ability to reactivate a control state is used to simulate the fundamental operations of quantum mechanics. The creation and manipulation of these control state objects, called continuations, is described in Section D.3.

But this kind of expressive power is rarely added to a language without introducing some corresponding difficulty. In the case of Scheme the difficulty comes from the kind of problems that were mentioned when fluid variables were introduced in Section 3.3.3. There are times when a program needs to set up some state that must be maintained for the duration of a particular body of code. In the example of Section 3.3.3 this state consisted of the storage location associated with the variable `radix`. Other common cases include opening and closing files, locking and unlocking data structures, and saving and restoring hardware registers. One of the fastest growing sets of functions in the MIT Scheme system are the "`with-...`" functions (`with-interrupt-mask`, `with-output-to-file`, `with-syntax-table`, etc.) used to perform this sort of "modify...execute...unmodify" operation.

In all of these cases the intention is to perform some action when control exits a block of code, independent of the mechanism used to exit the block. This is a fairly standard problem, often handled as a special case of the more general problem of handling exceptions or errors. An overview of two different solutions from other languages is supplied in Section D.2.

Because Scheme provides the ability to *re*-enter a region of execution, however, a number of new difficulties arise. If a region of code requires a certain state to be in existence while it is executing, then reentry to the region must reestablish that state. In order to support this requirement MIT Scheme provides a data structure, called a dynamic state space, that maintains the procedures required for transitions into and out of execution regions. This mechanism is described in Section D.4. But the dynamic and control states must be related to one another. Transitions within the

control space correspond to the exiting of one block of code and the entry into another. The dynamic state records the transition functions that must therefore be invoked. This interconnection is described in Section D.5.

## D.2 Exception Handling

Languages like Ada[43], Clu[37] and Mesa[40] provide syntactically limited ways of exiting a block of code. One way typically corresponds to an ordinary exit and is implicit when the body of code completes in the normal manner. Another syntactically distinct mechanism is used if the body of code completes in an unusual manner. These unusual exits are typically called signals or exceptions.

Users can provide exception handlers for certain blocks of code. When a program raises an exception (either by using the appropriate syntax or implicitly from errors detected by the runtime system) the block of code that has most recently been entered, has not yet been exited, and that contains an exception handler for the particular exception is located and the handler is invoked. A block of code that exits normally does *not* activate any of the exception handlers and furthermore these exception handlers are removed from consideration when future exceptions are raised (the block is now "exited"). If the code for an exception handler is required to run for all possible ways of exiting the block (as in the example of Section 3.3.2) the code for the block terminates by raising the appropriate exception condition itself instead of using the ordinary return mechanism.

Details of the mechanisms differ, but in general these systems provide for several important cases. Exception handlers may be supplied for a specific exception, a class of exceptions, and "all exceptions that aren't otherwise handled." Raising an exception can pass user selected data to the exception handlers. When an exception handler is invoked it may be permitted to re-invoke the exception after performing some code, thus becoming invisible except for providing clean-up code. Or it may raise a different exception, or force the normal exit from its own block rather than continuing the propagation of an exception. Or it may cause the procedure that raised the exception to appear to return a selected value.

In summary, these languages (a) annotate blocks of code with exception handlers; (b) distinguish normal return from unusual block exit; (c) provide

a path of communication from the point where an exception is detected to the point where it is handled; and (d) use this communication path to control the selection of relevant exception handlers. It is assumed that exceptions occur relatively infrequently, and thus the cost of invoking the mechanism for signalling and handling an exception can be high. The cost of both entering and normally exiting a body of code that contains exception handlers must be minimized.

The Lisp languages have evolved a different mechanism. They generally provide a means of *naming* a particular control state (using the `catch` mechanism in CommonLisp, for example) and a way of causing an arbitrary value to be returned after restoring that control state (`throw`). These constructs are utilized to provide many forms of control flow, generally not related to exceptional conditions. Typical uses include iterators and multiple-level procedure exit. There is also a way (`unwind-protect`) to associate with a block of code a set of actions that must be performed when it is exited in either manner (the normal return mechanism or the `throw` method). Unlike the other group of languages, Lisp dialects do not provide any direct method of disambiguating a normal return from an exceptional case, although it is easy enough to use the underlying structure to devise a mechanism for this purpose.

The way an exception condition is typically handled in Lisp is to provide a "well-known name" for a procedure intended to handle the exceptional condition. The system supplies a standard handler for exceptions, but users are free to rebind the name (in a dynamically scoped Lisp) or provide an alternative fluid value (in Scheme) to provide customized handling of the condition. These exception handlers are called in such a way that the previous value for the exception handler is always available under the standard name. This allows an exception to be propagated on to the previous handler. The control flow can be altered by a handler in the same way it would be by any procedure, using `throw` to reestablish an existing control state. Normal return from the handler may either attempt to continue the computation from the point where the exception occurred or invoke the previously existing handler, depending upon the particular Lisp system.

The two mechanisms appear very different from one another, and yet there is a deep similarity. There are a number of features that can be abstracted from both of these methods of handling exceptions to provide a basis for building other mechanisms. The essence of the mechanisms appears to

be the ability to:

1. Mark an execution state for possible use at a later time.
2. Select a destination execution state and cause a transfer of control to it.
3. Maintain the ordering information that corresponds to the normal return path between marked execution states.
4. Associate transition functions with marked execution states.
5. For each exception condition maintain a chain, correlated with the return path chain, of (potential) handlers for that exception.
6. Pass user specified information about the exception to the exception handlers.

MIT Scheme has divided these problems into two categories. The first involves creating, naming, and moving among different points in the execution of a program. Collectively, these operations (numbers 1 and 2) refer to what is known as the control state of the program. The second is related to inserting, removing, locating, and executing code that handles exception conditions. This group (numbers 4, 5, and 6) is known as the dynamic state of the program. The two are not independent, and it is in the interconnection of the two (number 3) that the mechanism for handling exception conditions lies.

## D.3 Control State

In order to support an exception handling mechanism there must be some way of marking, at run time, certain points in the execution of the program so that control can be restored to these points later. The notion, from denotational semantics, of a continuation embodies this in an elegant mathematical formulation. The continuation encapsulates not only a point in the code (like a label in assembly language or BCPL), but also the inherent stack of operations that remains to be performed at a particular point of time in the execution of a program. Section A.4 describes the use of

`call-with-current-continuation` in Scheme to capture a first-class object corresponding the the current continuation. The formal semantics of this primitive can be gleaned from the denotational description of Scheme contained in the implementation-independent language definition[48].

This mechanism allows any point in the execution of a program to be labeled (requirement 1 from Section D.2), by merely making the continuation in existence at that point into a standard Scheme object. The selection of an execution state to be resumed, then, is identical to choosing any other object in the system. The usual naming rules, parameter passing rules, and data structures can be used to devise arbitrary mechanisms. Transfer of control to the chosen state is accomplished by (at least syntactically) procedure call, since continuation objects are interchangeable with procedures (requirement 2).

## D.4 Dynamic State

MIT Scheme provides a mechanism (proposed by Hanson and Lamping[26]) intended to support requirements 4 and 5 of Section D.2. A dynamic state space (the word "dynamic" is often omitted) consists of a number of state points connected by arcs with a pair of transition functions on each arc, one for each direction along the arc. The MIT Scheme system is always operating at a specific point in each space that has been created. Programs can move within each space, causing the transition functions between the original position and the destination point to be executed. In order to guarantee a unique path between any two points within a space, the space is in fact constrained to be a tree.

State spaces are ordinary Scheme data structures. When they are created initially (by `make-state-space`) they contain a single point in which the system is considered to be operating. They can be extended by adding a point adjacent to the current one (`execute-at-new-state-point`), specifying the transition functions into and out of that point. The system executes the entering transition and is then located at the new point. Normal return from the primitive causes a transition out of the newly created point and into the point where execution of the primitive began. And a program can request transfer to another point in the space (`translate-to-state-point`). In this case a path is calculated to the destination and the system moves a step at

a time toward the destination, executing a transition function at each step.

The state space abstraction provides a convenient packaging of the notion of transition functions, providing support for requirement 4. Furthermore, the state space is built in a manner that causes it to reflect the normal return path for programs (unless the program performs a translate-to-state-point), so they can be used to satisfy requirement 5. By coupling the use of translate-to-state-point (motion in dynamic state space) with the use of continuations (motion in control space) these two can be kept in complete agreement. This is discussed in Section D.5.

The current dynamic state space model does have a deficiency, although it is easily repaired. There is currently no convenient way to pass information along the chain as transitions occur. By allowing the transition functions to receive arguments this problem would be solved, allowing requirement 6 to be easily met.

## D.5 Connecting the States

At the lowest level of the MIT Scheme system (in the "microcode") the control space and the dynamic state space are completely independent as described so far. This separation has been introduced largely to allow experimentation with mechanisms that support the exception handling at the system level. For ordinary use, however, they must be interconnected in a way that supports the common use of the system. MIT Scheme uses a variant on the familiar Lisp procedure `unwind-protect`. While `unwind-protect` allows a region of code to have an exit handler attached to it, this variant (introduced by Stallman[51] and known as `dynamic-wind`) allows a program to provide transition functions that will be invoked upon both (re-)entry and exit of a block of code:

```
(define (dynamic-wind re/entry code exit)
  (execute-at-new-state-point system-state-space
                              re/entry
                              code
                              exit))
```

Dynamic-wind operates by manipulating a specific dynamic state space, `system-state-space`. It creates a new point within that space, specifies the function `re/entry` for use on transition into that point and `exit` as the

transition function out of the point. It moves the system into the newly created point (causing `re/entry` to be evaluated) and then executes `code`. When this procedure ends the system will return to the previous state point (causing `exit` to be evaluated). But, by itself, this will not cause control transfers that occur through the use of continuations to evaluate the appropriate thunks.

In order to make dynamic-wind a standard facility the continuation object that users normally manipulate is *not* just the control state as might be expected. The procedures that create continuations (such as `call-with-current-continuation`) produce procedures (continuations) that capture *both* the control state and the current point within the system-state-space at the time the continuation is created. When this procedure is called (corresponding to CommonLisp's throw construct) both the saved control state and saved dynamic state point are restored. Since restoring the dynamic state point is done by moving from the current location in state space to the one being restored (using translate-to-state-point) all of the appropriate transition functions are invoked in the process.

The introduction of parallel processing raises a problem with this state space model. The system can no longer be considered to reside in a single state (control or dynamic) at each instant of time. The notion of *simultaneously* being in multiple states must be considered. The control state of the system is easily captured in a stack and allowing each processor to provide its own stack conveniently solves this problem. The dynamic state, however, is characterized by arbitrary user specified transition functions. Since transition functions always come in pairs (entry and exit), a single processor can move (in theory) between states when it performs a task switch and return to the same set of conditions it was in when it left the task. But the states may be mutually exclusive and there is no information available to determine which states are compatible and which are not. By separating the implementation of `fluid-let` from the dynamic space model the majority of problems with incompatible states are avoided. The remaining mutually incompatible states must provide transition functions that guard against any simultaneous access.

## D.6 Exception Handling via State Space

The dynamic and control state mechanisms, interconnected as described in Section D.5, adequately support the method traditionally used by Lisp for exception handling, even in light of the ability to reenter a control region. In fact, on a single processor, the MIT Scheme system uses the coupling of dynamic space and control space to implement the `fluid-let` mechanism of Section 3.3.3. The exception mechanism of Ada (and so forth) can be implemented on top of this base in several ways. The MIT Scheme system does not use this mechanism, so the choices outlined here have not been fully investigated, although they do not appear to pose any difficulties.

- Exceptions are named in these languages and these names are then used to indicate that the exception has occurred and invoke a handler. A direct analog of this approach in MIT Scheme is to use fluid variables to name the exception handlers themselves. Entering a block of code that contains a handler for a specifically named exception corresponds to surrounding that block of code with a `fluid-let` that changes the (fluid) binding of that name to the new handler. Signalling an exception is just procedure application, using the exception name as the name of the handler.

  In order to allow a handler to propagate an error back to the previous handler, it is critical that the fluid variables in effect when each handler is executed are those that existed when the handler was created. This can be achieved by writing the code for the handler in such a way that it (lexically) captures the fluid bindings when it is installed and restores them when it is invoked. A handler can return the flow of control back to the control stream at the block where the handler was installed by similarly capturing the control state when it is installed. Normal return from a handler would resume computation at the point where it was signalled, since it is just the ordinary procedure return of Scheme.

  This approach works well if the set of possible exceptions is statically known. Otherwise it may be difficult to accommodate such things as "default" exception handlers or handlers for groups of exceptions. Since the choice of names to bind must occur when the protected code block is entered it is not possible to anticipate the addition of a new exception after the block has been entered.

- The system-state-space is always correlated with the execution chain and can be used in place of the fluid variables to implement the selection of an exception handler and propagation of the exception. In order to install an exception handler, `dynamic-wind` is used to create a new state point. A pair of procedures generated in a standard way are installed as the transition functions. These procedures have two important properties: they are "easily recognized" as exception handlers, and when invoked they first test whether an exception is being signalled and then whether they were generated to handle that exception. Only if both of these conditions are met do they continue on to perform their work.

  When an exception is signalled, the system-state-space is walked back from the current point of execution to the root of the state space. Each transition function encountered along the way is tested to see if it is one of the "easily recognized" exception handlers. If so, it is invoked to determine whether or not it will handle the exception. If not the walk continues back as though the transition had not been marked.

  This approach deals well with the addition of exception conditions on a dynamic basis. Since the exception handlers are invoked for every exception, they can determine at the time when the exception occurs whether or not the exception is one to which they apply. Unfortunately, it requires *every* transition to be examined, even ones that do not apply to exception handling.

- By adding a new state space, say the exception-state-space, to the information that is saved and restored with a continuation it is possible to allow only exception handlers to be examined when an exception is signalled. The mechanism is the same as in the previous approach, but with transitions marked in this alternate space. This comes very close to the implementation normally found in languages like Mesa for the exception mechanism.

  This implementation does have an efficiency problem. All exception handlers are inspected, even ones that are not relevant to the particular exception being initiated. The argument generally given in defense of this is that exceptions occur infrequently and handlers for exceptions usually include a default handler, so the inefficiency is minimal.

- An intriguing alternative to this last approach is to provide a separate dynamic state space for each kind of *handler* that is installed. A separate data structure is maintained that can be used to determine whether a particular exception needs to execute any transition functions in that space. A single exception condition may require walking several of these "exception spaces," since the exception may satisfy more than one criterion. Thus these spaces must somehow be walked in parallel with each other.

  Installing an exception handler involves choosing the correct dynamic state space to use. Then the current point of the system in any of the other exception spaces that might be relevant to an exception handled by this space must be captured. The transition function installed must first cause the point in each of these other spaces to be restored to the captured position and then execute the actual handler. This intertwines the walks of the trees in a way that guarantees that the order in which exception handlers are examined corresponds to the (reverse of the) order in which they were installed.

  By calculating which state spaces are relevant when an exception is signalled, it is possible to examine only transitions relevant to that exception. Whether this elaborate mechanism is more efficient than the others will largely depend upon the extent to which handlers are created and removed and the overlap between groups of exceptions.

Selecting among these alternatives is a matter of taste, and has been left to users of the MultiScheme system. As mentioned earlier, no current applications (nor the system itself) have required an exception handling mechanism. In keeping with the tradition of Lisp languages, when need arises for such a mechanism the system implementors and the users of the system will doubtless negotiate a standard interface to the facilities described here.