The Design and Implementation of ALLOY, a Higher Level Parallel Programming Language

by

Thanasis Mitsolides

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

Department of Computer Science
New York University
June, 1991

Approved
Malcolm C. Harrison

To my parents

Acknowledgments

I would like to express my gratitute to my adviser Prof. M.C. Harrison for his patience and cooperation through the long years of the PhD program. Throughout the years of research, he contributed insight, fresh ideas, and alternative solutions. His efforts in improving the presentation of this thesis, are deeply appreciated. Furthermore, this thesis was prompted during discussions with him and the two other members of the concurrent languages group N. Markantonatos and G. Papadopoulos. I enjoyed working with them and I wish to thank them both. Of particular importance were the individual after hours discussions with N. Markantonatos.

Professors B. Goldberg and E. Schonberg are thanked for fitting the reading of the thesis into their busy schedules. The other members of the thesis committee, professors R. Dewar and A. Gottlieb, are also acknowledged for participating in the presentation.

My initiation to the world of programming languages and most importantly to the world of research is due to prof. C. Halatsis and the people of the Digital Equipments and Computers Lab of the University of Thessaloniki, Greece. I consider myself lucky to have worked with them.

Special thanks are also due to all my friends in New York for providing me with a pleasant environment away from home, where I could work productively. H. Khan helped improve the language of this thesis.

I wish to thank my parents and brothers whose emotional support even from so far away kept me going for all these years. This thesis would not be possible without them.

The Design and Implementation of ALLOY,
a Higher Level Parallel Programming Language

Thanasis Mitsolides
Advisor: Malcolm C. Harrison

Abstract

The goal of this thesis is to show that it is possible to define a parallel higher level programming language for programming in the large which will be able to easily express both complicated parallel problems and traditional serial ones. Such a language would provide many good features of serial and parallel programming languages and be appropriate for programming massively parallel computing systems. To demonstrate this a simple language, called ALLOY, was designed. The main features of this language, could be incorporated into other languages.

ALLOY, directly supports functional, object oriented and logic programming styles in a unified and controlled framework. Evaluating modes support serial or parallel execution, eager or lazy evaluation, non-determinism or multiple solutions. These modes can be combined freely. ALLOY is simple, utilizing only 29 primitives, half of which are for object oriented programming.

The power of ALLOY is demonstrated through the use of an unusually wide variety of examples. Some of the examples are: a) partition sort and FP library demonstrating clarity, efficiency, and simple parallelism, b) prime numbers and buffering demonstrating the ability to select between eager and lazy evaluation, c) systolic sort and merge sort demonstrating dynamic networks of communicating processes, d) N-queens and list permutations demonstrating serial and parallel searching. A library is given for programming in logic programming styles. Finally a number of parallel objects demonstrate ALLOY's ability to exploit massively parallel architectures effectively.

An interpreter of ALLOY together with a number of utilities and a programming environment has been written in Common Lisp. The system is available for anonymous ftp. It is shown that ALLOY can have reasonably efficient implementation on shared memory multiprocessor (MIMD) systems supporting highly parallel operations, on distributed architectures, and possibly on Data Flow architectures as well.

Contents

1 Introduction **1**

1.1 Scope of Thesis 2

1.2 Contents 3

1.3 Three programming paradigms 4

1.3.1 Parallel Functional Programming Languages 4

1.3.2 Parallel Object Oriented Programming Languages 5

1.3.3 Parallel Logic Programming Languages 6

1.4 ALLOY 6

1.4.1 Features 8

2 A Tour of ALLOY **9**

2.1 Basic Functional 9

2.1.1 Top Level 10

2.1.2 Factorial 10

2.1.3 Parallel Factorial 11

2.1.4 Partition Sort using lists 11

2.1.5 FP functions 12

2.2 Commitment 13

2.2.1 Tree equality 13

2.2.2 Process management 14

2.3 Lazy Evaluation 15

2.3.1 Integers 15

2.3.2 Fibonacci 16
2.3.3 Prime Numbers 16
2.3.4 Buffers 17
2.4 Networks 18
2.4.1 Asynchronous Systolic Sort 18
2.4.2 Merge Sort 19
2.4.3 Process Networks 20
2.5 Backtracking and Searching 20
2.5.1 Generators 21
2.5.2 Intersection 21
2.5.3 Trees scanning 22
2.5.4 Permutations 23
2.5.5 N queens 23
2.6 Parallel Objects 25
2.6.1 Counter 25
2.6.2 Putlist 25
2.6.3 Getlist 26
2.6.4 Queue 27
2.6.5 Blocking Counting Semaphore 28
2.6.6 Dining Philosophers 29

3 Reference Manual 31

3.1 Introduction 31
3.2 Essential expressions and primitives 32
3.2.1 Literal Expressions and Variables 32
3.2.2 Essential Special Expressions 33
3.2.3 Essential ALLOY primitives for functional programming 33
3.2.4 Essential ALLOY primitives for lazy evaluation 34
3.2.5 Essential ALLOY primitives for generator calls/use 35
3.2.6 Essential ALLOY primitives for object oriented programming 35
3.2.7 Other essential ALLOY primitives 37
3.3 Non-essential special forms and primitives 38

3.3.1 Special forms . . . 38
Obtained through pre-processing . . . 38
Macros . . . 38
Macro expressions for useful types . . . 40
Other messages accepted by objects . . . 40
3.3.2 Basic objects . . . 41
List management . . . 41
Square List management . . . 42
Arithmetic Functions . . . 42
Atoms . . . 43
Strings . . . 43
3.4 Recommended Libraries . . . 44
3.4.1 Higher order programming . . . 44
Functions taking functions as arguments . . . 44
Functions returning functions . . . 45
FP support . . . 45
3.4.2 Input Output . . . 46
Input . . . 46
Output . . . 47
3.4.3 Miscellaneous objects . . . 48
Single Assignment Variable . . . 48
Multiple Assignment Variable . . . 48
Single Assignment Vector . . . 49
Multiple Assignment Vector . . . 49
Single Assignment Dictionary . . . 49
Multiple Assignment Dictionary . . . 49
Top Level Caller . . . 50
Fetch and Add . . . 50
Other Objects . . . 50
3.4.4 Miscellaneous . . . 50
Variables . . . 50
Boolean functions . . . 50

Blocks 51
Loops 51
System Functions 52
Implementation Specific Functions 52

4 Functions, Closures, and their Evaluation Modes 55

4.1 Introduction 55
4.1.1 Why another Functional Kernel? 56
4.2 Closures 57
4.2.1 Functions 58
4.2.2 Environments and Values 58
4.3 Evaluation Modes 59
4.3.1 Message passing and Function Calls 59
4.3.2 Evaluation Modes 60
4.3.3 Generators 60
4.4 Macros for Local Environments 61
4.4.1 The `lets` expression 61
4.4.2 The `let` expression 62
4.5 Comparisons 62
4.5.1 ZLISP 62
4.5.2 MultiLISP 64
4.5.3 Queue Based Multiprocessing LISP 65
4.5.4 Concurrent Functional Language 66
4.6 Summary 67

5 Objects and Interprocess Communication 69

5.1 Introduction 69
5.1.1 Classes 70
5.2 Single item communications 71
5.2.1 One to one (shared) Communication 71
5.2.2 Many to one (shared) Communication 71
5.2.3 Communication to Many 72
5.2.4 Inter-message Communication 72

5.2.5 Parallel Architectures 74
5.3 Stream communication 75
5.3.1 One to one (shared) Communication 75
5.3.2 Many to One (shared) Communication 75
5.3.3 One to Many Communication 77
5.3.4 Many to Many Communication 77
5.4 Examples 78
5.4.1 Semaphore 78
5.4.2 Key 79
5.4.3 Lock 80
5.4.4 N Philosophers 81
5.5 Default Messages and Modules 82
5.5.1 Bubble-Up 82
5.6 Objects with Perpetual Processes 83
5.6.1 Co-expressions 83
5.7 Comparison with other Programming Languages 84
5.7.1 Actors 84
5.7.2 Vulcan 86
5.7.3 Linda 87
5.8 Summary 89

6 Generators and the Replicator Control Structure* 91
6.1 Introduction 91
6.1.1 Generate-and-test Problems 92
6.2 Definitions 94
6.2.1 First Look at Generators 94
6.2.2 Generators in a Replicator 95
6.2.3 Nested Generators 97
6.2.4 Values are Generators 98
6.3 Examples 99
6.3.1 Two Functions/Generators 99
6.3.2 More Examples 100

6.3.3 Computing and Processing the Values of a Generator in Parallel . . 101
6.3.4 Computing the Values of a Generator using Control Backtracking . 103
6.4 Implementation 104
6.4.1 Transformations and complications 104
6.4.2 Parallel Architectures 105
6.5 Discussion 106
6.5.1 Comparison with other Languages 106
6.5.2 Discussion 107
6.6 Summary 108

7 Logic Programming Styles **109**
7.1 Introduction 109
7.2 Logic Programming and Styles 110
7.2.1 Prolog 110
7.2.2 Or Parallel Prolog 111
7.2.3 Parlog 111
7.3 Comparisons 113
7.3.1 Prolog styles 113
7.3.2 OR-parallel Prolog 114
7.3.3 Programming in Parlog Style 115
7.3.4 And and Or parallel algorithms 116
7.4 Supporting Library 116
7.4.1 Prolog 117
7.4.2 Unification 117
7.4.3 Logical Variable 119
7.4.4 Automating the Translation 120
7.5 Summary 121

8 An Interpreter Simulating Parallelism **123**
8.1 Introduction 123
8.2 The interpreter in Lisp 124
8.2.1 Parts of the System 125
Top Level 125

Parser . . . 125
Pre-processor . . . 125
Functional Features . . . 126
Object Oriented Features . . . 126
Generators and Replicator . . . 127
Built in Functions and Expressions . . . 127
Built in Classes . . . 128
8.2.2 Data Structures of the Functional Part . . . 128
Synchronizing Variables . . . 128
Closure . . . 128
Processes . . . 129
Environment . . . 130
8.2.3 Data Structures for Classes . . . 131
Classes . . . 131
Objects . . . 131
8.2.4 Data Structures for Replication and Generators . . . 132
Fake generators . . . 132
Normal Generator . . . 133
Multiple Generators . . . 133
8.3 Practical Considerations . . . 134
8.3.1 Portability . . . 134
8.3.2 Benchmarks . . . 134
8.3.3 Efficiency and Compilation . . . 135
8.3.4 Ineffiecient Implementation of Features . . . 136
8.3.5 Using the Interpreter . . . 136
8.4 Implementations on Parallel Architectures . . . 136
8.4.1 Shared Memory MIMD . . . 137
Data Flow Architectures . . . 137
8.4.2 Distributed Architectures . . . 137
8.5 Summary . . . 138

9 Conclusion **139**

A The Interpreter of ALLOY in Common Lisp **141**

A.1 Top level . . . 141

A.1.1 alloy.l . . . 141

A.1.2 load.l . . . 143

A.1.3 inter.l . . . 145

A.2 Main Interpreter . . . 151

A.2.1 fcall.l . . . 151

A.2.2 ocall.l . . . 161

A.2.3 rcall.l . . . 168

A.2.4 gcall.l . . . 175

A.3 Pre-processor . . . 181

A.3.1 prep.l . . . 181

A.3.2 gprep.l . . . 187

A.3.3 oprep.l . . . 191

A.4 Utilities . . . 193

A.4.1 io.l . . . 193

A.4.2 values.l . . . 197

A.4.3 alutil.l . . . 200

A.4.4 util.l . . . 204

A.5 Built in functions and objects . . . 207

A.5.1 build.l . . . 207

A.5.2 build2.l . . . 214

A.5.3 obuild.l . . . 220

A.6 Library in ALLOY . . . 230

A.6.1 alloylib.a . . . 230

A.6.2 lists.a . . . 231

A.6.3 bool.a . . . 232

A.6.4 loop.a . . . 233

A.6.5 misc.a . . . 234

A.6.6 hifherf.a . . . 235

A.6.7 fp.a . . . 237

A.7 Creating ALLOY . . . 238

A.7.1 Makefile . . . 238
A.7.2 execfirst . . . 240

B Benchmarks **241**

B.1 Simple Functional . . . 242
B.1.1 Append and naive reverse . . . 242
B.1.2 Make and find length of list . . . 243
B.1.3 While and until . . . 243
B.2 Sorting . . . 243
B.2.1 Quick sort using lists . . . 243
B.2.2 Systolic sort . . . 244
B.3 Lazy evaluation . . . 245
B.3.1 Prime numbers . . . 245
B.4 Generators . . . 246
B.4.1 Intersection . . . 246
B.4.2 Permutations . . . 246
B.4.3 Queens . . . 247
B.5 Prolog . . . 248
B.5.1 Append, Or Parallel Append, and Or Parallel Intersection . . . 248
B.6 Miscellaneous . . . 251
B.6.1 Dining Philosophers . . . 251

Chapter 1

Introduction

The availability of low-cost powerful single processors has made the use of large-scale parallelism an attractive alternative to the most powerful computers such as super-computers. The latter approach to powerful computing (mainframes and supercomputers) has been successful in a number of areas, most of which are characterized by problems in which the flow graph of the computation is relatively static, and thus subject to analysis and optimization for these systems.

However, in the area of what might be called "unpredictable" computations in which the flow graph is essentially unknown, there has been much less success. These problems are exemplified by compilation, search problems, natural language analysis, simulation, reasoning, high-level pattern recognition, and most problems in artificial intelligence. These are problems in which the size and shape of data-structures is unknown, the sequencing of computations is unknown, there is little repetition of computation, and the execution time of components of computation is also unknown. Such complex problems can not be solved efficiently by supercomputers whose speed is largely due to pipelining and vector processing.

As more knowledge of effective algorithms for these problems is developed, there will be an increasing need for techniques for executing these algorithms efficiently. The higher expressiveness of Parallel Languages can be essential to the description of such complex algorithms.

Such computations do not lend themselves to SIMD or pipeline machines, with their requirement for computational regularity. The only reasonable alternatives are very fast

single processor machines and MIMD machines (shared memory or distributed), with the possibility of data-flow machines lurking around the corner. However, there is so far no common agreement on the best technique for exploiting such machines. Three general approaches have attracted considerable support: functional languages, logic languages, and object oriented languages. Unfortunately, these approaches are regarded as if they were mutually exclusive. Once having chosen one of these languages, capabilities provided by the others are not available. For toy problems this disadvantage does not appear; only as real-world problems are encountered is it recognized that different aspects of the problem require different approaches. Furthermore, adding parallelism carelessly can make some of these languages weaker is certain important aspects. A naive mixture of programming languages can help a little but leads to considerable confusion in the long run.

1.1 Scope of Thesis

This thesis defines a number of primitives in a unified form. Most of these primitives appear in other languages, though some of them do not. These primitives are presented as parts of a simple language called ALLOY.

ALLOY is, a high level parallel programming language appropriate for programming massively parallel computing systems. ALLOY directly supports functional, object oriented and logic programming styles in a simple, unified and controlled framework. Evaluation modes support serial or parallel execution, eager or lazy evaluation, non-determinism or multiple solutions. Most importantly, these modes can be combined freely.

ALLOY is simple, utilizing only 29 primitives and expressions, half of which are for object oriented programming. It is based on a combination of ideas from functional, object oriented and Logic Programming languages.

A full interpreter of ALLOY has been implemented in Common lisp. An easy to use interactive environment is also provided. Parallelism is emulated in this implementation. Under Austin-Kyoto Common Lisp the interpreter averages about 60 fc/mi (function calls / million (VAX equivalent) cpu instructions).

1.2 Contents

The rest of this chapter gives a concise description of some desirable and undesirable features (or lack thereof) found in languages following the three programming paradigms of functional, object oriented and logic programming. It concludes with an outline of ALLOY's properties.

Chapter 2 gives a tour of the ALLOY system. A particularly wide variety of examples are shown, tested and briefly explained. It is maybe the most important chapter in this thesis

Chapter 3 gives a detailed description of the ALLOY primitives, expressions, and other utilities. This chapter can serve as the manual. It should be consulted during the reading of other chapters whenever clarification is needed.

Chapter 4 describes the functional part of ALLOY. It is a set of evaluation modes and closures. It has only 12 primitives or expressions and yet provides serial evaluation, and-parallelism, or-parallelism, eager and lazy evaluation, commitment, non-determinism, and generators. Finally, the functional part is compared with languages such as Zlisp, Multilisp, and Queue Based Multiprocessing Lisp.

Chapter 5 describes the interprocess communication facilities of ALLOY. Simple (and frequent) communication is supported directly (by μ calculus) and advanced communication through the use of parallel objects (implemented in ALLOY, not built-in). These facilities are compared with similar features of languages such as Actors, Vulcan, and Linda.

Chapter 6 describes the searching facilities of ALLOY. These support depth first (control backtracking) and breadth first search. They are based on the replicator control structure in combination with the ability to control generators. These facilities are compared with similar features of languages such as ICON, PROLOG, Andorra Prolog, SETL, Scheme, and GHC.

Chapter 7 describes ALLOY's abilities to emulate logic programming styles. It is shown that classes of logic programs can be expressed more simply in a functional style (some-times with generators). ALLOY is compared against languages such as PROLOG, or-parallel Prolog, and PARLOG.

Chapter 8 describes the current implementation of ALLOY. It explains the various

parts of the interpreter and describes in detail the data structures used by it. This is followed by a set of benchmarks performed on a Sparcstation 1. It concludes with a commentary on parallel implementations on shared memory MIMD and distributed architectures.

Finally, Appendix A lists the current interpreter of ALLOY. It is written in Common Lisp. This appendix includes the list sources, ALLOY libraries and the makefile to construct the executables. The interpreter is followed by a set of benchmarks in Appendix B. These benchmarks are used in chapter 8.

1.3 Three programming paradigms

The three programming paradigms mentioned above are examined briefly here. A short description of these systems is followed by a reference to desirable and undesirable features encountered in languages of these forms. It is clear that even though most languages are really good in some important aspects, they are weak in some other important aspects.

Usually their designers concentrate on kernels, and leave the issues of higher level features and control structures to the programmer. In some cases, it may not be clear how some features can be provided. In other cases, the selection of one feature prevents selection of another feature. Also, features are omitted for efficiency reasons.

In later chapters (chapter 4, chapter 5, chapter 6) these features are examined in detail. These chapters show how ALLOY supports many of these important features.

1.3.1 Parallel Functional Programming Languages

The primary distinguishing features of functional languages [Hud89] are higher order functions, anonymous functions, eager or lazy evaluation and pattern matching. Lately more advanced features have appeared in languages of this form; these include non-deterministic evaluation, generators, cacheing, polymorphic type inference, and implicit parallelism because of referential transparency.

These languages have been praised for their clarity and ability to express algorithms neatly and efficiently. Their simplicity makes them an attractive base for a design.

On the other hand, many important features are missing from languages of this type. Most PFPLs do not provide any form of backtracking, generators or a meaning of multiple

solutions. Non-deterministic evaluation is usually absent. Often, parallelism looks artificial or weak. Either synchronization or mutual exclusion is left weak in such languages. Eager and lazy evaluation are rarely available both in the same language. Object oriented programming is complicated when available and then it does not support parallel ADTs.

1.3.2 Parallel Object Oriented Programming Languages

Object oriented programs are collections of independent components each providing some services [Wag90]. Evaluation involves interaction among these objects. This model of computation makes these languages suitable for programming distributed architectures.

Abstraction is the first goal achieved through object oriented programming. As a result, these languages are highly suitable for programming in the very large. Re-usability of components and easy management of complex software libraries is another major advantage.

Inheritance plays an important role in object oriented programming. It captures a higher form of abstraction, that complements data abstraction with object management. It provides a natural mechanism for subtyping. Alternatively, delegation simplifies the task of multiple inheritance by giving more control to the user.

However, object oriented languages often treat simple program components with the same techniques used for the management of very large programs. This complicates unnecessarily simple algorithms and makes their implementations far from elegant. Parallel object oriented languages are particularly weak in this area. Additionally, there seems to be little support for parallel abstract data types.

Unfortunately, multiple inheritance has as many different forms as there are object oriented languages. The problems include deciding what is to be inherited from an object, what is shared between the two, and how ambiguities are resolved. Abstraction and incremental modification of behavior is often sacrificed for the sake of incremental modification of code. Simplicity is often sacrificed for the sake of efficiency. For example, classes are allowed to inherit variables of other classes. This may improve efficiency of interaction among the two classes but damages abstraction and complicates maintenace of the super-class.

1.3.3 Parallel Logic Programming Languages

Logic Programming has demonstrated a high degree of expressiveness [Sha89]. This is largely based on the flexibility of the logical variable, which needs only one operation, unification, to read it or write it. Unification provides impressive pattern matching power.

The highly declarative nature of these languages and heavy use of non-determinism complete their features. Programming in these languages is sometimes very close to a detailed specification of the problem. One such example is the definite clause grammars in Prolog. Though many programming languages have claimed that programming in them involves describing what the problem is rather than how to solve it, logic programming languages are the most succesfull.

However, the fact that the logical variable is very powerful, and yet the only possible type of a variable, introduces a number of rather undesirable complications. The flow of control, even in simple cases, can be so vague that understanding a program can degenerate into trying to determine what is input and what is output. For the same reason compilers have trouble producing efficient code, particularly for distributed architectures.

Parallelizing logic programming languages is not a simple task. *Don't know* non-determinism (or-parallelism) turns out to be virtually incompatible with and-parallelism in the context of the logical variables. Thus, parallel versions of these languages lack either full or-parallelism or full and-parallelism.

All communication is done with streams formed from logical variables. This imposes unnecessary and inefficient serialization when there are multiple producers or consumers for a stream. This prevents implementation of very parallel algorithms. Finally, lazy evaluation requires different programming techniques than those employed for eager evaluation and that often complicates constructing and understanding programs which mix the two evaluation modes.

1.4 ALLOY

The design of ALLOY is ability-oriented. The objective is to identify most features desirable to a programmer, and show that they can be provided in a simple language. Efficiency is not the primary consideration, though care has been taken to avoid interaction among features which could result in unexpected inefficiencies.

During the numerous stages of its design, ALLOY's expressiveness was tested by implementing a variety of algorithms. These algorithms were selected because they are used in papers to demonstrate some neat property of a language. Having elegant solutions for these algorithms implied the existence of certain desirable features in ALLOY. ALLOY supports the following:

- Clear and efficient basic functional style.
- Serial and Parallel evaluation with commitment.
- Eager and Lazy evaluation.
- Generators and Replicators.
- Classes of Parallel Objects.
- Unrestricted combinations of the above.

It is shown that all of the above features[1] are compatible, in that they cannot interact in ways that prevent efficient implementations. The efficiency of each feature on MIMD shared memory multiprocessors can be in the same order of magnitude as in the implementation of the feature when isolated.

The kernel of ALLOY has only 29 primitives, half of which are for the support of Object Oriented programming. For all its simplicity, it is able to provide simple and clear solutions to problems such as the following:

- factorial, partition sort and FP style library, each highly parallel.
- Fibonacci sequence, and prime numbers (sieve), each eager or lazy.
- systolic sort, and hamming network, each clear in spite of complex communication patterns.
- list member, tree leaves, list permutation, and n-queens, each generating one or all solutions.
- queue, stack, counter, and semaphore, each a parallel object without any explicit synchronization.
- Dining philosophers, making use of abstraction and explicit synchronization.

[1] In particular the interaction of parallel generators and parallel objects with the rest of the language.

These and other examples are shown and compared with their best implementation in the most convenient programming language(s) [Mit89b]. These comparisons are intended to serve as an informal proof that ALLOY is a superset (in expressiveness) of some powerful and dissimilar languages and thus provides an unusually high degree of expressiveness.

1.4.1 Features

The number of elementary operations in ALLOY is either about the same as, or a little more that, the number of elementary operations in languages such as PARLOG, Prolog, Scheme, Multi-lisp, Actors, Vulcan, Flavors, ICON etc. each taken independently. Even though the elementary operations in ALLOY are not always found in that form in other languages, its design has been influenced by Pure-Lisp, ICON, the Relational Language, Smalltalk and Vulcan and less from Prolog and PARLOG.

Non-determinism is provided in ALLOY to permit the expression of indifference or uncertainty in an algorithm. Generators are included to ease the development of programs which make use of the generate-and-test technique. The replicator control structure for driving generators allows easy expression of algorithms performing depth-first (backtracking) and/or breadth-first searches.

ALLOY provides features for expressing parallel algorithms. These include: parallel evaluation of expressions (and-parallelism), non-deterministic selection of alternatives (or-parallelism), synchronization which can be implicit (using synchronizing variables) or explicit (by means of serial execution), and parallel abstract data types (dynamic state of objects).

ALLOY is object oriented. Closures are just objects. Classes and other objects with simple interfaces can be accessed with a functional syntax, in addition to the more traditional message passing syntax. Objects can be parallel and support delegation.

Overall, ALLOY has dynamic type checking, static scoping, closures, high order functions, parallel objects with delegation and abstract interfaces, single assignment synchronizing variables, serial and parallel execution, eager or lazy evaluation, non-deterministic functions, generators and replicators.

Chapter 2

A Tour of ALLOY

This section is intended to stimulate the reader's curiosity. It contains a wide variety of example programs written in ALLOY. Each, is followed by illustrations of its use. It is recommended that this chapter is only brushed through at first reading and that it be re-read after chapter 3.

All these examples and many others can be found in the `~mitsolid/alloy/progs` directory of system `spunky.cs.nyu.edu`. On `spunky` ALLOY can be entered with the command `~mitsolid/alloy/alloy`. The libraries and functions used in these examples can be loaded with the `require` declaration.

The above, as well as the sources of the interpreter, some benchmarks, the manual, a paper, and the thesis itself are all available for anonymous ftp on system `cs.nyu.edu` (address `128.122.140.22`) directory `pub/local/alloy`.

2.1 Basic Functional

The kernel of ALLOY is functional. Functional programming is clean, easy to understand, efficient and appropriate for parallel architectures. The important points here are the clarity in expression of simple parallel algorithms, and the power in the combination of high order functions.

2.1.1 Top Level

The top level behaves as if it has as local variables all possible names. Usually a program is created in an editor then loaded in ALLOY then tested and so on until it is debugged.

The next command is executed in parallel since it is enclosed in parenthesis. Synchronization is implicit, due to the *synchronizing* variables:

```
spunky % ~mitsolid/alloy/alloy

ALLOY version 2.0 5/21/91

ALLOY > (list (set y3 (sum y6 y7)) (set y1 (sum y2 y3)) (set y2 (sum y4 y5))
              (set (y4 y5 y6 y7) '(1 2 3 4)))
==> (7 10 3 (1 2 3 4))
```

The next command shows the effects of suspension at the top level:

```
ALLOY > (sum x 2)
ALLOY > (set x 3)
==> 3
==> 5
```

Using a higher order function:

```
ALLOY > (mapcar (mu (x) (lreturn (sum 1 x))) '(2 3 4 5))
==> (3 4 5 6)
```

Multiple assignment variables:

```
ALLOY > _x
==> NIL
ALLOY > (set _x 2)
==> 2
ALLOY > _x
==> 2
ALLOY > (set _x 3)
==> 3
```

2.1.2 Factorial

Even though the algorithm is serial in nature the following program makes no attempt to explicitly synchronize processes or evaluate serially. Implicit synchronization is based solely on the *synchronizing* variable:

```
(setfun fact(n)
   (return (if (gt n 0) (times n (fact (diff n 1))) 1)))
```

Test runs:

```
ALLOY > (fact 25)
==> 15511210043330985984000000
```

2.1.3 Parallel Factorial

This program makes use of the divide-and-conquer technique of distributing the task of making the multiplications to many processors. Assuming $O(n)$ processors are available, this executes in $O(\log n)$ time (for appropriate implementation):

```
(setfun pfact(n)
   (return (pfact2 1 n)))

(setfun pfact2(from to)
   (return (if (ge from to) from
               (let ((mid (quotient (sum from to) 2)))
                  (times (pfact2 from mid) (pfact2 (sum mid 1) to))))))
```

Test runs:

```
ALLOY > (pfact 25)
==> 15511210043330985984000000
```

2.1.4 Partition Sort using lists

This solution to Partition Sort is a good indication of how powerful implicit synchronization can be. All function calls are evaluated in a parallel way. Assuming $O(n)$ processors are available it can sort a list in worst case $O(n)$ time.

```
(provide "psort")
(require "putlist")

(setfun psort(l)
   (return (psortrest l nil)))

(setfun psortrest(l rest)
   (if (null-p l) (return rest)
     (return (let (((lsmall lgreat) (partition (car l) (cdr l))))
                (psortrest lsmall (cons (car l) (psortrest lgreat rest)))))))

(setfun partition(x l)
```

```
(if (null-p l) (return '(nil nil))
  (return (let (((lsmall lgreat) (partition x (cdr l))))
              (if (lt x (car l)) (list lsmall (cons (car l) lgreat))
                (list (cons (car l) lsmall) lgreat))))))
```

Test runs:

```
ALLOY > (require "psort")
Redefining function: PSORT
Redefining function: PSORTREST
Redefining function: PARTITION
==> T
ALLOY > (psort '(5 2 7 3 5 1 4 2 3))
==> (1 2 2 3 3 4 5 5 7)
```

Note how the psort function and other supporting functions have been loaded with the `require` declaration. Should this declaration be given again, these functions will not be reloaded.

2.1.5 FP functions

Writing programs in Backus' FP style is easy. The only disadvantage over FP is the requirement of longer names for functors and functions (FP uses many special symbols). Also, before a built-in ALLOY function is used, it must be made unary by applying function `fp-fun` to it. Normally though, it would be simpler in ALLOY to use higher order functions

Here are some functions written in fp style using the `fp` utilities:

```
(require "fp")

(setfun fp-list-length(e)
   (return ((h-comp (h-red sum 0) (h-map (h-const 1))) e)))

(setfun fp-int-prod(e)
   (return ((h-comp (h-map (fp-fun times)) fp-trans) e)))

(setfun fp-mat-prod(e)
   (return (let ((f (h-compn (h-map (fp-fun sum))
                             (h-map fp-int-prod)
                             fp-distl)))
             ((h-compn fp-trans (h-map f) fp-distr
                       (h-construct car (h-comp fp-trans cadr))) e))))
```

Test runs:

```
ALLOY > (fp-list-length '(a s d f))
==> 4
ALLOY > (fp-int-prod '((1 2 3) (4 5 6)))
==> NIL
ALLOY > (fp-int-prod '((1 2 3) (4 5 6)))
==> (4 10 18)
ALLOY > ((fp-fun sum) (fp-int-prod '((1 2 3) (4 5 6))))
==> 32
ALLOY > (fp-mat-prod '(((1 2) (3 4)) ((5 6) (7 8))))
==> ((19 22) (43 50))
```

2.2 Commitment

Commitment evaluation appears in two basic forms. The most common is or-parallel evaluation of alternatives. Less common, but not less important, is the ability of the commitment mechanism to control execution of arbitrary processes. This last form becomes easier to use when combined with object oriented programming.

2.2.1 Tree equality

The next function succeeds only if its two arguments are the same tree. It fails as soon as the inequality is identified. Thus it avoids redundant computation:

```
(provide "trees")

(setfun eqtrees(t1 t2)
   (if (and (cons-p t1) (cons-p t2))
     (if [block (block (unless (eqtrees (car t1) (car t2)) (fail))
                       (unless (eqtrees (cdr t1) (cdr t2)) (fail)))
                (return t1))
     (return (eql t1 t2))))
```

Test runs:

```
ALLOY > (eqtrees '((1 2)((3))) '((1 2)((3))))
==> ((1 2) ((3)))
ALLOY > (eqtrees '(1 ((1 2)((3)))) '(2 ((1 2)((3)))))
(isotrees '(1 ((1 2)((3)))) '(2 ((1 2)((3)))))
==> %f
```

If a number of processors linear in the size of the trees are available the above function determines that trees are equal in $O(\log n)$ time, where n is the depth of the trees. Note that (depending on scheduling) the second call may fail immediately.

2.2.2 Process management

Object `proca` provides simple process management operations. It provides operations for making a call, testing if it is finished, examine the result of the call even before it is finished, and terminate the call before it is finished. The object is defined in ALLOY as follows:

```
(provide "proca")

(class-start proca
             (static val _f fin ter closure args)
             (methods execute value terminate finished))

(setfun new(c . a)
   (set closure c)
   (set args a)
   (set _f %f))

(setfun execute()
   [block (set val (closure . args))
          (set _f 't) (set fin 't)]
   (when fin (return val))
   (when ter (return val)))

(setfun value()
   (return val))

(setfun finished()
   (return _f))

[setfun terminate()
   (set ter 't)
   (return (if _f %f %t))]

(class-end proca)
```

Test runs:

```
ALLOY > (require "proca")
==> T
ALLOY > (set p1 ('new proca sum x 2))
==> %(object of class PROCA)
ALLOY > ('execute p1)
ALLOY > ('finished p1)
==> %f
ALLOY > (set x 5)
==> 5
==> 7
ALLOY > ('finished p1)
```

```
==> T
ALLOY > ('terminate p1)
==> %f
```

An object process `p1` was created. Then it was called, tested if it finished, then allowed to finish, and then an attempt was made to terminate it. It is not possible to terminate it once it is finished.

Test runs:

```
ALLOY > (set p2 ('new proca (mu() (lreturn (cons 4 (sum y 2))))))
==> %(object of class PROCA)
ALLOY > ('execute p2)
ALLOY > ('finished p2)
==> %f
ALLOY > ('value p2)
==> (4 . %w)
ALLOY > ('terminate p2)
==> T
==> (4 . %w)
ALLOY > (set y 4)
==> 4
ALLOY > ('finished p2)
==> %f
ALLOY > ('value p2)
==> (4 . %w)
```

In this example the created process was terminated before it finished execution. The output of the process is available before and after it was terminated. Note that uninstantiated variables are printed as `%w`.

2.3 Lazy Evaluation

Lazy evaluation is a natural way for the consumer of a stream of values to control the producer. Placing a buffer between these two processes adds some advantages of asynchronous communication. It is shown how a buffer can be implemented based on the same principles.

2.3.1 Integers

The next function returns all integers from an integer on:

```
(setfun ints(n)
  (return (cons n (ints (sum n 1)))))
```

Test runs:

```
ALLOY > (set i (lazy (ints 5)))
ALLOY > (data (car i))
==> 5
ALLOY > (data (cadr i))
==> 6
ALLOY > (data (nth 7 i))
==> 11
ALLOY > (set i2 (cdrn 10 i))
==> %w
ALLOY > (data (car i2))
==> 15
ALLOY > (data (car i))
==> 5
```

2.3.2 Fibonacci

The following functions `Fibonacci` and `Fibonacci` return the Fibonacci sequence, the first eagerly the second lazily:

```
(setfun fibonaccil()
   (return (lazy (fib 0 1))))

(setfun fibonacci()
   (return (fib 0 1)))

(setfun fib(f1 f2)
   (return (let ((f3 (sum f1 f2)))
               (cons f3 (fib f2 f3)))))
```

Test runs:

```
ALLOY > (set f (fibonaccil))
ALLOY > (mapcar data (sublist 6 9 f))
==> (13 21 34 55)
ALLOY > (mapcar data (sublist 50 55 f))
==> (20365011074 32951280099 53316291173 86267571272 139583862445
     225851433717)
```

2.3.3 Prime Numbers

Function `lprimes` returns lazily the list of all prime numbers in order:

```
(setfun lprimes()
  (return (lazy (primes))))

(setfun primes()
  (return (cons 1 (sieve (ints 2)))))

(setfun sieve(l)
  (return (cons (car l) (sieve (filter (car l) (cdr l))))))

(setfun filter(x l)
  (if (equal (remainder (car l) x) 0) (return (filter x (cdr l)))
    (return (cons (car l) (filter x (cdr l))))))

(setfun ints(n)
  (return (cons n (ints (sum n 1)))))
```

Test runs:

```
ALLOY > (set p (lprimes))
ALLOY > (car p)
==> 1
ALLOY > (mapcar data (sublist 1 5 p))
==> (1 2 3 5 7)
ALLOY > (mapcar data (sublist 20 25 p))
==> (67 71 73 79 83 89)
ALLOY > p
==> (1 2 3 5 7 %w %w %w %w %w %w %w %w %w %w %w %w %w %w %w 67 71 73 79 83
     89 . %w)
ALLOY > (data (nth 15 p))
==> 43
ALLOY > p
==> (1 2 3 5 7 %w %w %w %w %w %w %w %w %w 43 %w %w %w %w 67 71 73 79 83
     89 . %w)
```

Note that elements of the list of all prime numbers which are not needed are not computed. In general, lazy evaluation is an easy way to save unnecessary computation.

2.3.4 Buffers

It is often desirable to have a buffer between the consumer and the lazy producer of a stream of values. This example implements a buffer of variable length.

```
(provide "buffer")

(setfun buffer(length l)
  (return (let ((rest (bb-1 length l)))
            (lazy (bb-2 l rest)))))
```

```
(setfun bb-1(n l)
  (if (lt n 1) (return l)
              [block (data (car l))
                     (return (bb-1 (diff n 1) (cdr l)))]))

[setfun bb-2(l rest)
  (data (car rest))
  (return (cons (car l) (bb-2 (cdr l) (cdr rest))))]
```

Test runs:

```
ALLOY > (set f (fibonaccil))
ALLOY > f
==> %w
ALLOY > (set fb (buffer 5 f))
ALLOY > f
==> (1 2 3 5 8 . %w)
ALLOY > fb
==> %w
ALLOY > (data (nth 9 fb))
==> 55
ALLOY > fb
==> (%w %w %w %w %w %w %w %w 55 . %w)
ALLOY > f
==> (1 2 3 5 8 13 21 34 55 89 144 233 377 610 . %w)
```

Variable `f` is the original stream. Normally, only variable `fb` would be used once the buffer was created. Here we read `f` to see if the buffer works fine. Note that after we asked for the 9^{th} element of `fb` the original stream `f` got evaluated up to its 14^{th} element.

2.4 Networks

Some algorithms are too complicated to be expressed in functional terms. These cases are more naturally solved as networks of communicating processes. The use of appropriate objects is convenient in cleaning up such communication.

2.4.1 Asynchronous Systolic Sort

This algorithm makes use of an object which creates a limited form of stream allowing only back communication. When the way the object works is clear and the properties of the returning stream understood, understanding this sorting algorithm is easy. The object `putlist` is described later in this chapter.

Each call of function `ssort` creates a filter (function `ssortx`) which finds and returns the smallest element in the input stream (list) and places the rest of its elements in another stream (object `rest`), instance of class `putlist`). Function `ssort` is called recursively on that stream. Assuming $O(n)$ processors are available it can sort a list in worst case $O(n)$ time.

```
(require "putlist")

(setfun ssort(l)
   (return (if (null-p l) nil
               (let ((rest ('new putlist)))
                  (cons (ssortx (car l) (cdr l) rest)
                        (ssort ('get-orig-head rest)))))))

(setfun ssortx(x l rest)
   [when (null-p l) ('end rest) (return x)]
   (when (cons-p l)
      [when (gt (car l) x) ('put rest (car l))
                           (return (ssortx x (cdr l) rest))]
      [when (le (car l) x) ('put rest x)
                           (return (ssortx (car l) (cdr l) rest))]))
```

Test runs:

```
ALLOY > (ssort '(5 2 7 3 5 1 4 2 3))
==> (1 2 2 3 3 4 5 5 7)
```

2.4.2 Merge Sort

Another basic form of streams is realized by objects of class `getlist` which is described later in this chapter. Parallel merge sort is more efficient in terms of computational requirements than systolic sort is as it needs no more than $O(n \log n)$ computation.

```
(require "merge")
(require "getlist")

(setfun msort(l)
   (return (msortx ('new getlist l) (length l))))

(setfun msortx(gl n)
   (if (gt n 1)
      (return (amerge (msortx gl (quotient n 2))
                      (msortx gl (diff n (quotient n 2)))))
      (return (list ('get gl)))))
```

The definition of `amerge` follows:

```
(provide "merge")

(setfun amerge(l1 l2)
  (when (null-p l1) (return l2))
  (when (null-p l2) (return l1))
  (when (le (car l1) (car l2))
        (return (cons (car l1) (amerge (cdr l1) l2))))
  (when (ge (car l1) (car l2))
        (return (cons (car l2) (amerge (cdr l2) l1)))))
```

Test runs:

```
ALLOY > (msort '(5 2 7 3 5 1 4 2 3))
==> (1 2 2 3 3 4 5 5 7)
```

2.4.3 Process Networks

The synchronizing value is powerful enough to express simple channels (streams) without the need for objects. An example of a static network of processes communicating through a single channel is illuminating. In the next example the stream produced contains all numbers of the form $2^x * 3^y * f^z \ \forall x, y, z \geq 0$:

```
(setfun hamming()
  (return (let (ham)
            (let ((mults (mu(c) (lreturn
                                 (mapcar (mu(x) (lreturn (times c x)))
                                         ham)))))
              (set ham (cons 1 (amerge (mults 2)
                                       (amerge (mults 3) (mults 5)))))))))
```

Of course, in general the network of processes does not need to be static.

2.5 Backtracking and Searching

Searching a solution over a domain of values is often necessary. In such cases, having to create the domain and later iterate on its elements is not only inconvenient or inefficient but can also be impossible when the domain is infinite in size. The use of generators overcomes these problems. It is shown that generators can be used for depth first (backtracking) or breadth first search.

2.5.1 Generators

The next example gives some uses of simple generators. The next function `intsxy` returns one or each number in a range (depending on how it has been called).

```
(setfun intsxy(x y)
   (when (le x y)
     (lreturn x)
     #!(lreturn *(intsxy (sum x 1) y))))
```

Test runs:

```
ALLOY > (intsxy 1 10)
==> 1
ALLOY > #*(intsxy 1 10)
==> (1 2 3 4 5 6 7 8 9 10)
ALLOY > #(list *(intsxy 1 10))
==> ((1) (2) (3) (4) (5) (6) (7) (8) (9) (10))
ALLOY > #(list *(intsxy 1 2) *(intsxy 5 7))
==> ((1 5) (1 6) (1 7) (2 5) (2 6) (2 7))
ALLOY > #*(intsxy 1 *(intsxy 1 5))
==> (1 1 2 1 2 3 1 2 3 4 5 1 2 3 4)
ALLOY > #(list *(intsxy 1 *(intsxy 1 3))  *(intsxy 6 *(intsxy 6 8)))
==> ((1 6) (1 6) (1 7) (1 6) (1 7) (1 8) (1 6) (1 6) (1 7) (1 6) (1 7) (1 8)
     (2 6) (2 6) (2 7) (2 6) (2 7) (2 8) (1 6) (1 6) (1 7) (1 6) (1 7) (1 8)
     (2 6) (2 6) (2 7) (2 6) (2 7) (2 8) (3 6) (3 6) (3 7) (3 6) (3 7) (3 8)
    )
```

Note the difference in behavior between nested generators and generators in the same level. In the one case we have a nested triggering, while in the other a cartesian product. Function `intsxy` does an amount of computation linear in the number of numbers returned.

2.5.2 Intersection

This example demonstrates the use of generators. Function `getmember` can generate all elements of a list using a number of computational steps linear in the size of the list. Note the cartesian product created in function `intersection`. Also, all functions can be used either normally or as generators.

```
(setfun getmember(l)
  (when (cons-p l) (return (car l)) #!(return *(getmember (cdr l)))))

(setfun intersection(s1 s2)
  #(lets ((x *(getmember s1)) (y *(getmember s2)))
      (when (eql x y) (return x))))
```

```
(setfun ismember(x l)
  #(if (eql x *(getmember l)) (return x)))
```

Test runs:

```
ALLOY > (ismember 2 '(4 2 6))
==> 2
ALLOY > (ismember 3 '(4 2 6))
==> %f
ALLOY > (getmember '(2 4 6 4 1))
==> 2
ALLOY > #*(getmember '(2 4 6 4 1))
==> (2 4 6 4 1)
ALLOY > (intersection '(2 5 1 3) '(8 2 6 5))
==> 5
ALLOY > #*(intersection '(2 5 1 3) '(8 2 6 5))
==> (2 5)
```

Note that by using the null replicator in function `getmember` we allow the function to return the elements of a list with an amount of computation linear in the length of the list.

2.5.3 Trees scanning

Function `get-leaf` will return a leaf when called simply and all the leafs when called as a generator:

```
(provide "trees")

(setfun get-leaf(t)
   (if (cons-p t) (block #!(return *(get-leaf (car t)))
                         #!(return *(get-leaf (cdr t))))
                  (return t)))
```

Test runs:

```
ALLOY > (get-leaf '((1 2)((3))))
==> 1
ALLOY > (get-leaf '((1 2)((3))))
==> 2
ALLOY > (get-leaf '((1 2)((3))))
==> 1
ALLOY > (list #(list 'leaf *(get-leaf '((1 2)((3))))))
==> ((LEAF 1) (LEAF 2) (LEAF NIL) (LEAF 3) (LEAF NIL) (LEAF NIL)
     (LEAF NIL))
ALLOY > (list #(list 'leaf *(get-leaf '((1 . 2) 3 . 4))))
==> ((LEAF 2) (LEAF 3) (LEAF 4) (LEAF 1))
```

```
ALLOY > (list #(list *(get-leaf '((1 . 2) 3 . 4))
                     *(get-leaf '((1 . 2) 3 . 4))))
==> ((1 1) (1 2) (1 3) (1 4) (2 1) (2 2) (2 3) (2 4) (3 1) (3 2) (3 3) (3 4)
     (4 1) (4 2) (4 3) (4 4))
```

2.5.4 Permutations

This function should be called as a generator and it then produces all permutation of the list passed to it as an argument.

```
(provide "perms")

(setfun permutations(l)
  (if (cons-p l)
      (list #(lets (((x rest) *(delete-one l)))
                #(return (cons x *(permutations rest)))))
      (return nil)))

(setfun delete-one(l)
  (when (cons-p l)
    (return (list (car l) (cdr l)))
   #(return (let (((x rest) *(delete-one (cdr l))))
              (list x (cons (car l) rest))))))
```

Test runs:

```
ALLOY > #*(permutations '(1 2 3))
==> ((1 2 3) (1 3 2) (2 1 3) (2 3 1) (3 1 2) (3 2 1))
```

2.5.5 N queens

This program solves the N Queens problem. N queens must be placed on a chess board in a way that they can not attack each other. The number N is given as a parameter. The same program can find only one solution when called normally. As expected, finding one solution can be much faster than finding all.

```
(require "perms")

(setfun queens(n)
  (lets ((lst (make-list n)))
    #(lets ((poslist (board lst *(permutations lst))))
       (when (not (unsafeall poslist)) (printnl poslist)
```

```
                                        (return poslist)))))

(setfun unsafeall(board)
  (when (cons-p board)
    (when (unsafe (car board) (cdr board)) (return))
    (when (unsafeall (cdr board))          (return))))

(setfun unsafe(q board)
  (when (cons-p board)
      (when (unsafe1 q (car board)) (return))
      (when (unsafe  q (cdr board)) (return))))

(setfun abs(x)
  (return (if (lt x 0) (diff x) x)))

(setfun unsafe1(q p)
  (when (eq (abs (diff (car q) (car p)))
            (abs (diff (cdr q) (cdr p)))) (return)))

(setfun board(lst places)
  (return (if (null-p lst) ()
            (cons (cons (car lst) (car places))
                  (board (cdr lst) (cdr places))))))

(setfun make-list(n)
  (if (ge n 1)
     (return (cons n (make-list (diff n 1))))
     (return ())))
```

Test runs:

```
ALLOY > (queens 4)
((4 . 3) (3 . 1) (2 . 4) (1 . 2))
==> ((4 . 3) (3 . 1) (2 . 4) (1 . 2))
ALLOY > #*(queens 4)
((4 . 3) (3 . 1) (2 . 4) (1 . 2))
((4 . 2) (3 . 4) (2 . 1) (1 . 3))
==> (((4 . 3) (3 . 1) (2 . 4) (1 . 2)) ((4 . 2) (3 . 4) (2 . 1) (1 . 3)))
```

It is important to note here that the possible board configurations are created and tested in parallel. Also, the space of configurations is not actually created as an entity and in fact if only one solution is needed some configurations (or parts thereof) may not be created at all.

A more efficient program would represent all permutations as a lazily evaluated tree. In this way, many permutations would never be generated. Also, a lot of repeated checks would be avoided. Of cource, using the permutations in that form would be more complicated.

2.6 Parallel Objects

Since objects are often accessed by many processes at the same time it is important that they do not create bottlenecks. Synchronizing variables allow objects to process many messages in parallel and to keep their state consistent without the requirement for low level synchronization.

2.6.1 Counter

An simple class, demonstrating implicit synchronization through dynamic synchronizing variables, is a counter. This example makes use of the default message mechanism.

```
(class-start counter
  (dynamic v)
  (import number-p sum)
  (methods eval))

(setfun new(x)
  (if (number-p x) (set next-v x) (set next-v 0)))

(setfun eval(i)
  (return (block2 (set next-v (sum v i)) v)))

(class-end counter)
```

Test runs:

```
ALLOY > (set i (counter 10))
==> %(object of class COUNTER)
ALLOY > (list (i) ('eval i) (i) (i))
==> (10 11 12 13)
ALLOY > (i)
==> 14
```

This object although consistent it is not parallel even if the architecture can support such parallel operations. On a parallel architecture, we should implement it in terms of the `faa` (fetch and add) object. That object is guaranteed to be completely parallel if the architecture can support it.

Note that both the object `i` and class `counter` are treated as if they were functions.

2.6.2 Putlist

An object list whose end can be incrementally instantiated my methods. The head is available as a normal list/channel. This stream (many-to-one communication) and the

following one (`getlist`, which provides one-to-many communication) are often very useful. The program demonstrates the use of dynamic variables in an object.

```
(provide "putlist")

(class-start putlist
   (static  _orig-head)
   (dynamic head)
   (methods put end get-orig-head)
   (interface puter   put end))

(setfun new()
   (set _orig-head next-head))

(setfun put(x)
   (return (let() (set head (cons x next-head)) x)))

(setfun end()
   (return (set head nil)))

(setfun get-orig-head()
   (return  [let(h) (set h _orig-head) (set head next-head)
                    (set _orig-head nil) h]))

(class-end putlist)
```

Test runs:

```
ALLOY > (list (set pl ('new putlist)) (set h ('get-orig-head pl))
              ('put pl 2) ('put pl 3) ('put pl 4) h)
==> (%(object of class PUTLIST) (2 3 4 . %w) 2 3 4 (2 3 4 . %w))
ALLOY > h
==> (2 3 4 . %w)
ALLOY > ('put pl 6)
==> 6
ALLOY > h
==> (2 3 4 6 . %w)
```

2.6.3 Getlist

An object list whose head can be accessed using messages. The tail is incrementally instantiated elsewhere. This stream provides one-to-many communication.

```
(provide "getlist")

(class-start getlist
   (dynamic head)
```

```
   (methods get))

(setfun new(tail)
   (set next-head tail))

(setfun get()
   (return (let ()
              (set next-head (cdr head)) (car head))))

(class-end getlist)
```

Test runs:

```
ALLOY > (set gl ('new getlist (cons 1 gx)))
==> %(object of class GETLIST)
ALLOY > ('get gl)
==> 1
ALLOY > ('get gl)
ALLOY > (set gx '(4))
==> (4)
==> 4
ALLOY > ('get gl)
==> %f
```

2.6.4 Queue

A parallel queue. Elements can be enqueued or dequeued at any time. The only serialization is that a dequeue must wait for the previous message (and only the previous one) to complete.

```
(provide "queue")
(require "putlist")
(require "getlist")

(class-start queue
   (static  puts gets)
   (import  putlist getlist)
   (methods enqueue dequeue)
   (interface enqueuer enqueue)
   (interface dequeuer dequeue))

(setfun new()
   (set puts ('new putlist))
   (set gets ('new getlist ('get-orig-head puts))))

(setfun enqueue(x)
```

```
   (return ('put puts x)))

(setfun dequeue()
   (return ('get gets)))

(class-end queue)
```

Test runs:

```
ALLOY > (list (set q ('new queue))
              ('dequeue q) ('enqueue q 1) ('enqueue q 2) ('dequeue q))
==> (%(object of class QUEUE) 1 1 2 2)
ALLOY > ('dequeue q)
ALLOY > ('enqueue q 4)
==> 4
==> 4
```

2.6.5 Blocking Counting Semaphore

This is a counting semaphore. The only serialization is that a `p` message cannot be completed until the previous message has be processed.

```
(provide "semaq")
(require "queue")

(class-start semaphore
  (import queue data repeat)
  (static q)
  (methods p v))

(setfun new(n)
  (set q ('new queue))
  (vs (if n n 0)))

[setfun p(n)
   (ps n) (return %t)]

[setfun v(n)
   (vs n) (return %t)]

(setfun ps(n)
   (repeat (if n n 1) ^(data ('dequeue q))))

(setfun vs(n)
   (repeat (if n n 1) ^('enqueue q 't)))

(class-end semaphore)
```

Test runs:

```
ALLOY > (set ss ('new semaphore))
==> %(object of class SEMAPHORE)
ALLOY > ('p ss)
ALLOY > ('v ss)
==> T
==> T
ALLOY > ('v ss)
==> T
ALLOY > ('p ss)
==> T
ALLOY > ('p ss)
```

2.6.6 Dining Philosophers

The following program provides a solution to the *5 philosophers* problem. In to this problem, a number of philosophers are having dinner. There are a number of forks equal to the number of philosophers, placed between adjacent philosophers. A philosopher spends some time thinking and then attempts to eat. In order to eat a philosopher needs both forks. After he finishes eating he starts thinking until he is hungry again.

The following solution is starvation and deadlock free:

```
(require "semaq")

(class-start philosopher
             (static id fork1 fork2)
             (import gt diff printnl)
             (common semaphore)
             (methods live))

(setfun new(f1 f2 i)
   (set id i) (set fork1 f1) (set fork2 f2)
   (if (gt id 1) ('v c-semaphore)))

[setfun live()
   ('p c-semaphore) ('p fork1) ('p fork2)
   ('v fork1) ('v fork2) ('v c-semaphore)
   (live)]

(class-end philosopher)

; Dinner for 5.

(setfun dine()
```

```
(let ((f1 ('new semaphore 1))(f2 ('new semaphore 1))
      (f3 ('new semaphore 1))(f4 ('new semaphore 1))
      (f5 ('new semaphore 1)))
   ('live ('new philosopher f1 f2 1) n)
   ('live ('new philosopher f2 f3 2) n)
   ('live ('new philosopher f3 f4 3) n)
   ('live ('new philosopher f4 f5 4) n)
   ('live ('new philosopher f5 f1 5) n)))
```

Test runs:

```
ALLOY > (dine)
```

A general solution to this problem and one which does not make use of any particular trick (i.e. keeping one of the philosophers out of the room) to avoid deadlocks can be found in the `progs` directory.

Chapter 3

Reference Manual

This chapter gives a precise definition of the small ALLOY kernel and some useful libraries.

3.1 Introduction

ALLOY is a dynamically typed, statically scoped parallel programming language based on the programming paradigms of functional and object oriented programming, in a way that subsumes most of the logic programming paradigm.

Evaluation modes support serial or parallel execution, eager or lazy evaluation, and non-deterministic evaluation or multiple solutions. Multiple solutions are used *and* controlled by the replicator control structure.

Variables are *synchronizing*. Any attempt to read the value of a variable returns its current value or its future value if it has no value at the time. When a process needs to examine the value of such a variable it suspends until a value is available from it. No objects can be destroyed, but their space can be reused for other objects when they are no longer accessible.

Functions can fail but if they succeed they can also return a result. They are objects, instances of the built-in class `closure`. Closures evaluate when sent message `'eval`. A function call, sends the default message `'eval` to an object. Any object (not only closures) which accepts this message can be called as a function. Parameters are always passed by value. *Local* and *Global* commitment of the evaluation of a closure are possible. When a function commits, useless processes it created are killed. This mechanism provides or

parallelism, non-deterministic selection, generators, and exceptions.

Functions can be called as generators. In that case, instead of committing at the first opportunity, functions can return many values. These generators can be driven with the *replicator* control structure or by other generators. The replicator and generators are useful for searching in a serial depth first or parallel breadth first way, can provide backtracking, and combine multiple results.

Object oriented programming is fundamental in ALLOY. Every value in ALLOY, including classes, messages, functions, and other objects are first class objects. Any can be used as a function. Everything, except classes, is an instance of a class.

If the name of a message to a class is omitted it is assumed to be `'new`. Thus, classes can be requested to produce an instance of themselves using a syntax like that of a function call. For example, the basic operation for list management, `cons`, is a class and its instances are pairs.

Multiple inheritance is available through delegation. Special variables `self` and `here` have the usual meaning. Any object can have many abstract interfaces. While normal *static* variables can be used to describe the state of objects, *dynamic* variables are useful in the definition of parallel objects which can process many messages at the same time.

3.2 Essential expressions and primitives

This section lists and explains all the primitive expressions and operations in ALLOY. This can be thought of as the ALLOY kernel. Most non-essential expressions and operations are easy to to define in term of the essential ones.

3.2.1 Literal Expressions and Variables

- `(quote` *e*`)`

  Its value is the argument itself unevaluated.

- *name*

  A synchronizing variable. Any attempt to read the variable returns a value-to-become object if a value is not yet available. When a process needs to examine the value of such a variable it suspends until one is available. A variable is distinguished as *uninstantiated* if it has not been assigned to anything, and *set* or *instantiated* if

it has.

3.2.2 Essential Special Expressions

- $(e_1 e_2 \cdots e_n)$

  A message is sent to object e_2 when e_1 evaluates to a symbol (message name). The message is then processed by the method with that name in that object. If no method with that name is defined (or inherited) the call fails. All expressions evaluate in parallel and the method handling that message is called as soon as it is available.

  The message's name is optional and is assumed to be omitted when e_1 is not a symbol. In that case, if e_1 is a class, it is assumed to be preceded by symbol `'new`. If e_1 is a non-class object it is assumed to be preceded by symbol `'eval`. This makes functional-style calls possible. In other words, message name `'new` is the default for classes and message `'eval` is the default for objects.

- $(e_1 e_2 \cdots \ . \ e_n)$

  Similar to the above. In general the arguments can be incomplete (or even infinite) expressions which are matched against the formal arguments.

- `[...]`

  Similar to (...) but arguments are evaluated serially, and the call is made after all arguments have been evaluated.

3.2.3 Essential ALLOY primitives for functional programming

- `(mu` *args-pattern* $e_1 e_2 \cdots e_n$`)`

  Creates a new function/closure. The closure is an object, an instance of class `closure`. However, this class is not allowed to create closures directly with the `'new` message.

  The function has access to the environment that is accessible at the point of the closure's creation. The argument's pattern is an expression of variables. It can be an incomplete list, a single variable, can contain other expressions etc. When message `'eval` is sent to a closure (implicitly or explicitly,) the function is invoked

and the expressions forming its body evaluate serially or concurrently depending on the type of bracket the body is enclosed in, in the definition.

Unless a function is explicitly requested to succeed and return a result (see special form `return`), it fails. A function that fails returns a special value which is being refered to as the *failure value*. The failure value does not behave like a normal generator and can be used in conditional expressions.

- `(return` e`)`

  The global return. It affects the outermost statically visible function. Useful when it is needed to make that function return a value from a closure nested in another closure. It can also be used for exceptions. If the function has not been called as a generator it commits and returns the value which is the parameter of the `lreturn` expression. As a result of the commitment, all processes created by that function, except the ones created by the `return` expression, are killed.

  If that function has been called as a generator then each `return` expression does not commit execution of the function but just provides one more result for the function. In the latter case, the `return` expression is not considered complete until a new value has been requested from the generator. This is useful when we need to control execution of a generator depending on the way the generator is used (e.g. backtracking).

- `(fail)`

  The global failure. It affects the outermost statically visible function. If that function has not been called as a generator then the effect of this expression is equivalent to `(return ((mu() )))` (i.e. returning a failure). If that function has been called as a generator then its execution terminates and no more values are generated by it.

3.2.4 Essential ALLOY primitives for lazy evaluation

- `(lazy` e`)`

  The argument and every expression evaluated as a result of that is evaluated lazily. That is, it is not evaluated unless its value is needed. There is one exception: Expressions which a) are called in functions called as a result of the evaluation of this expression and b) are not evaluated by (l)return statements evaluate eagerly as

soon as the result(s) of that function is needed.

- `(eager` *e*`)`

  This results in having the argument evaluate eagerly. Used to guarantee eager execution even when the call was lazy.

3.2.5 Essential ALLOY primitives for generator calls/use

- `#` *expr*

  Replicator operator. The expression is replicated once for each combination of the values returned by the enclosing (non-nested) generators. An attempt to form a new replica starts as soon as a new value is needed (this can be delayed during serial execution or lazy evaluation). New values are requested from the generators as needed. Replicas are executed as soon as they get created.

- `#!` *expr*

  Null Replicator. It replicates and evaluates its arguments like the normal replicator does, but apart from side effects (i.e. a return statement) it looks as if no replication has taken place. Efficient.

- `*` *expr*

  If argument is a function call the function behaves as a generator. It also returns values on demand. See `return` expression. If the generator contains other generators it is called once for each combination of the results of the enclosed (non-nested) generators.

  If argument is not a function call then is a value and generates itself. However the failure value does not generate anything. This allows the expression of conditional expressions.

3.2.6 Essential ALLOY primitives for object oriented programming

- `(class-start` *class – name class-declaration-list*`)`

  Starts definition of a class. The current implementation allows such a definition only at the top level. After the definition of the class, variable *class-name* will contain the object representing this class. Thus classes, as well as messages and any other

objects, are first class objects.

A class can contain methods and local functions while its state can be described using static or dynamic variables.

class-definition-list is a list of any of the following declarations:

- `(static` *name* ...`)`

  Declares the static variables describing the state of an instance. Static variables are *normal* (single-assignment, synchronizing) variables. These variables are initially uninstantiated. They can be instantiated with the `set` expression. Each message while it is processed has access to these variables. An attempt to access an uninstantiated state variable returns the *future* value of that variable. An attempt to examine a future value is suspended until the *future* value is known.

- `(dynamic` *name* ...`)`

  Declares the dynamic variables describing the state of an instance. Each message while it is processed has access to its own set of dynamic variables, and also to those of the next message. These variables are initially uninstantiated. They can be instantiated with the `set` expression.

- `(import` *name* ...`)`

  Declares variables imported by the class from the enclosing scope. These variables are initially uninstantiated.

- `(common` *name* ...`)`

  For each class *name* an instance of it is created and called `c-`*name*. That instance is known to all instances of the defining class.

- `(methods` *function-name* ...`)`

  Defines the methods of the class. All other functions defined in the class are not methods.

- `(interface` *interface-name method-name* ...`)`

  Defines an auxiliary interface to the class. There can be any number of interface declarations. *interface-name* becomes a method of the class returning a restricted form of the recipient-object accepting only the interface methods. Only interfaces which are subsets of the current interface can be requested from

an object.

- `(class-end` *class-name*`)`

  Ends definition of a class. The top level object as well as classes during their definition implicitly provide definitions for all possible variables. If a variable is used during the definition of a class, then it is treated like a `static` variable. After the definition of the class, no new variables can be created.

- `('new` *class-object argument ...*`)`

  Creates and returns a new instance of the class described by the class object. Method `new` can be customized.

- `next_`*name*

  Is the instance of the dynamic variable *name* which is/will-be current for the next message to this object.

- `(set` *vars-pattern e*`)`

  Gives values to uninstantiated variables. Returns the pattern of variables with their values, but failure values wherever a variable already had a value.

- `('delegate` *object*`)`

  Delegates current message to `object`. Returns the result from its processing.

- `('resend` *object*`)`

  Re sends current message to `object`. Returns the result from its processing.

- `self`

  Is the first object which received the current message directly, and not as a result of delegation.

- `here`

  Is the current object.

3.2.7 Other essential ALLOY primitives

- `(eql` e_1 e_2 $\cdots$ e_n`)`

  Returns the last of the expressions if they are equal. Otherwise returns the failure value. Two objects are equal if they are the same object (i.e. pointers), equal numbers, equal atoms, or equal strings.

3.3 Non-essential special forms and primitives

The following features can be implemented either directly in the language itself or with some simple pre-processing. These are available as built in for convenience.

3.3.1 Special forms

A special form, in contrast to a function call, does not follow the normal evaluation rules. It can be thought of as a macro that replaces itself by some regular function call.

Obtained through pre-processing

These special forms require some non-local pre-processing. In practice it will be easier to have them implemented directly.

- `(lreturn` e`)`
  The local return. Affects the local-most function.
- `(lfail)`
  The local failure. Affects the local-most function.
- `(var-p` *variable-name*`)`
  Succeeds if variable is not set. It can be implemented in terms of an enhanced `sa-var` class.
- *_other-name*
  A multiple assignment variable (experimental). This is a *conventional* variable. All multiple assignment variables are initialized to nil. It can be implemented in terms of `ma-var` class.

Macros

The following expressions are simple macros:

- `(setfun` *name vars-pattern* $e_1 e_2 \cdots e_n$`)`
  The same with `(set` *name* `(mu` *vars-pattern* $e_1 e_2 \cdots e_n$`))` except that a function can be redefined if it exists at the top level. This redefinition can, in theory, be achieved by keeping closures inside multiple assignment variables.

- `(setfun (`*name vars-pattern*`)` $e_1 e_2 \cdots e_n$`)`

  The same as above.

- `(let ((`*vars-pattern expression*`)` $\cdots$`)` be_1 be_2 $\cdots$ be_n`)`

  First evaluates expressions and binds the patterns of variables. This is done in parallel or serially depending on whether these bindings are given in parentheses or brackets. After that, if the bindings are evaluated serially, or in parallel, if they are evaluated concurrently, the expressions be_i in the body of let are evaluated. The result of be_n is the result of the `let` expression. Evaluation of the expressions in the body is done concurrently or serially, depending on whether the let expression is enclosed in parentheses or brackets.

  If an *expression* is not available the corresponding variable is left uninstantiated. Uninstantiated variables can be given a value with the `set` expression as with the state variables of objects. This macro uses objects to create uninstantiated variables. When be_n is a generator, and `let` is called as a generator, `let` can return all the results of be_n.

- `(lets ((`*vars-pattern expression*`)` $\cdots$`)` be_1 be_2 $\cdots$ be_n`)`

  Like the `let` expression but returns a failure value by default. An `lreturn` expression can be used to return something else.

- `(letrec ((`*vars-pattern expression*`)` $\cdots$`)` be_1 be_2 $\cdots$ be_n`)`

  Like the `let` expression but all *expression* can access the variables in *vars-pattern*. Useful for the definition of recursive local functions.

- `(letrecs ((`*vars-pattern expression*`)` $\cdots$`)` be_1 be_2 $\cdots$ be_n`)`

  Like the `letrec` expression but does not return anything by default. An `lreturn` expression can be used to return something.

- `(when` e_1 e_2 $\cdots$ e_n`)`

  Conditional expression. It first evaluates the first argument and only when it become clear that it will not fail, does it start evaluating the rest of the arguments. Evaluation of the body expressions follow the usual rules of serial/parallel evaluation. Similarly, e_1 does not have to finish execution before the other arguments start executing, unless evaluation is serial. It returns the value of e_n when e_1 succeeds, the failure value otherwise.

- `(if` e_1 e_2 e_3`)`

  Conditional expression. Evaluates and returns the second argument if the first one succeeds but the third one otherwise. The third argument is optional and is assumed to be the failure value if it is missing. Again, the rules of serial/parallel evaluation are followed.

- `'` e

  Equivalent to expression: `(quote` e`)`.

- `^` e

  Equivalent to expression: `(mu () #(lreturn` e`))`.

- `^!` e

  Equivalent to expression: `[mu () #(lreturn` e`)]`.

- `(delay` e`)`

  Equivalent to expression: `(lazy (eager` e`))`.

Macro expressions for useful types

- *number*

  Returns an object representing that number.

- "*string*"

  Returns an object representing that string.

Other messages accepted by objects

The pre-processor adds these messages to a class at the moment of its definition.

- `('class-p` *class*`)`

  If argument is a class, its name is returned.

- `('object-p` *object*`)`

  If argument is a not a class, the name of its class is returned.

- `('class` *object*`)`

  Returns the class in which the object belongs.

The following function is used for conventional type checking:

- (*name*-p *object*)

  Succeeds only when the *object* is an instance of the class with name *name*.

3.3.2 Basic objects

There are special syntactic conveniences for the creation of each one of them. For example a number can be created just by writing it normally without having to sent a `'new` message to its class etc.

List management

Since lists (objects of class `cons`) are very useful objects, the following functions are predefined to ease their use:

- `(car` e`)`

  The car (first element) of a pair (an object).

- `(cdr` e`)`

  The cdr (second element) of a pair.

- `(cons` e_1 e_2 $\cdots$ e_n`)`

  Groups the expressions in a list of pairs.

- `(list` e_1 e_2 $\cdots$ e_n`)`

  Like cons but the last element of the last pair is the special atom `nil`.

- `(null-p` e`)`

  Same as `(eql` e `nil)`.

- `(nth` n e`)`

  Will return the n^{th} element of list e.

- `(carn` n e`)`

  Will return the n^{th} car of list e.

- `(cdrn` n e`)`

  Will return the n^{th} cdr of list e.

- `(sublist` i j e`)`

  Will return the part of list e from position i to position j.

- `(append` l_1 l_2 $\cdots$ l_n`)`

  Returns the concatenation of the lists given to it as arguments.

- `(length` l`)`

  Will return the length of list l.

- `(mapcar` *function_message* l`)`

  Will return the list of the results of the application of *function_message* to each of the elements of list l.

Square List management

Square lists (printed in square brackets) are used in serial evaluation. These are objects of class `scons`. The following functions are predefined for ease of use:

- `(scar` e`)`

  The car (first element) of a square pair (an object).

- `(scdr` e`)`

  The cdr (second element) of a square pair.

- `(scons` e_1 e_2 $\cdots$ e_n`)`

  Groups the expressions in a list of pairs.

Arithmetic Functions

Here are some messages accepted by numbers:

- `('string` *number*`)`

  Returns a string containing the characters in the decimal representation of the input number.

The following is a list of predefined functions operating on numbers:

- `(sum` *number* ...`)`

  Will return the sum of its arguments. Like the functions described bellow, the operation treats its arguments in a left associative way.

- `(diff` *number* ...`)`

  Will return the difference of its arguments.

- `(times` *number* ...`)`

  Will return the product of its arguments.

- `(div` *number* ...`)`

  Will return the the division of its arguments.

- `(quotient` *number* ...`)`

  Will return the quotient of the division of its arguments.

- `(remainder` *number* ...`)`

  Will return the remainder of the division of its arguments.

- `(eq` *number* ...`)`

  Returns first argument if all arguments are equal numbers, otherwise fails.

- `(lt` *number* ...`)`

  Returns first argument if all arguments are numbers and of each pair of successive ones the first is less than the second, otherwise fails.

- `(gt` *number* ...`)`

  Returns first argument if all arguments are numbers and of each pair of successive ones the first is greater than the second, otherwise fails.

- `(le` *number* ...`)`

  Returns first argument if all arguments are numbers and of each pair of successive ones the first is less than or equal to the second, otherwise fails.

- `(ge` *number* ...`)`

  Returns first argument if all arguments are numbers and of each pair of successive ones the first is greater than or equal to the second, otherwise fails.

Atoms

Here are some important messages accepted by atoms:

- `('string` *atom*`)`

  Returns a string containing the characters in the name of the input atom.

Strings

Here are some important messages accepted by strings:

- (`'atom` *string*)

  Returns an atom whose name consists of the characters in the input string.

- (`'number` *string*)

  Returns a number represented by the input string.

- (`'explode` *string*)

  Causes the *string* to return the characters it contains as a list of strings, one for each character.

- (`'append` s_1 s_2 ... s_n)

  Returns a string containing the characters of the argument strings in order.

3.4 Recommended Libraries

These are some utilities which are easily implemented in ALLOY. It may be necessary to execute the `require` function before their first use, in order to have their definition loaded.

3.4.1 Higher order programming

A number of predicates facilitating higher order programming. They are defined in library file `higherf`. Can be loaded using the `require` expression.

Functions taking functions as arguments

- (`reduce` *function value list*)

  Equivalent to (*function* l_1 ... (*function* l_n *value*)).

- (`filter` *predicate list*)

  Equivalent to (*function* l_1 ... (*function* l_n *value*)).

- (`mapcar2` *function list1* `v`*list2*)

  Applies `function` to pairs of elements from the two lists with the same position. Returns the list of results.

- (`mapcarn` *function* l_1 l_2 ... l_n)

  Like `mapcar2` but for any number of lists.

Functions returning functions

The operation of the returned functions is given with an brief description and an example.

- `(h-bu` *function value*`)`
  Returns a unary function which when applied to an argument *a* returns (*function value a*).
- `(h-rev` *function*`)`
  Returned function expects the arguments of *function* in the reverse order. Special case is function `h-rev2` for binary functions.
- `(h-red` *function value*`)`
  Returned function does reduction using *function* and an initial value of *value*.
- `(h-dup` *function*`)`
  Returns a unary function which when applied to an argument *a* returns (*function a a*).
- `(h-comp` f_1 f_2`)`
  Returned function is a composition of argument functions.
- `(h-compn` f_1 f_2 ... f_n`)`
  Returned function is a composition of argument functions.
- `(h-Const` *value*`)`
  Returned function always returns *value*.
- `(h-construct` f_1 f_2 ... f_n`)`
  Returned function takes n arguments and passes one to each unary function f_i.
- `(h-mapn` f_1 f_2 ... f_n`)`
  Returned function maps function f_i to the elements of its i^{th} argument successively and returns the list of the results of each mapping. Efficient special cases are functions `h-map` and `h-map2`.

FP support

A number of predicates facilitating programming in FP style is provided in library file `fp`. It can be loaded using the `require` expression. All of these functions (except `fp-fun`)

expect one physical argument which is a list of the actual arguments.

- `(fp-fun` *function*`)`
  Returned function expects all the arguments of the original function in one list.
- `(fp-rev` *function*`)`
  Returned function expects the argument list in the reverse order.
- `(fp-trans (`$list_1$ $list_2$ ... $list_n$`))`
  The lists in the input list must have the same length. The returned list has length equal to the length of $list_i$ and the elements are lists of length n. The j^{th} element of the i^{th} sublist returned is the i^{th} element of $list_j$.
- `(fp-distl (`*value list*`))`
  Returns a list whose elements are lists of two elements the first of which is *value* and the second the corresponding element from *list*.
- `(fp-distr (`*value list*`))`
  Distributes *value* to the right of the elements in *list*.

3.4.2 Input Output

Objects of these classes are responsible for I/O to the screen or files. These classes can not be defined in ALLOY itself.

Input

Class `input` accepts message `'new` and optionally an argument which must be a string and must be the name of a file. If the argument is string `"console"` then the objects gets input from the console. Object `input-console` is predefined and gets input from the console. Objects created by this class accept the following messages:

- `('new input` *file-name*`)`
  Will return an object for input from the file. The file name is optional. If it is not given, the default is `"console"`.
- `('read` *input*`)`
  Will read an expression from object *input*.

- (`'read-line` *input*)

  Will read the next line as a string from object *input*.

- (`'read-string1` *input*)

  Will read a string of length 1 from object *input*.

- (`'eof` *input*)

  Succeeds only if end of file has been reached.

- (`'empty` *input*)

  Succeeds only if a character is available. This is especially useful in a parallel environment as it can be used to peek at the input stream and read only when some input is indeed available, thus avoiding unnecessary blocking.

- (`'active` *input*)

  Succeeds only when the stream has not been closed.

- (`'close` *input*)

  Closes the stream.

Output

Class `output` accepts message `'new` and optionally an argument which must be a string and will be the name of a file. If the argument is string `"console"` then the objects sends output to the console. Object `output-console` is predefined and sends output to the console. Object `error-console` is predefined and sends output to the error console. Objects created by this class accept the following messages:

- (`'new` *output file-name*)

  Will return an object for output to the file. The file name is optional. If it is not given, the default is `"console"`.

- (`'print` *output e*)

  Will print expression *e* to object *output*.

- (`'printnl` *output e*)

  Will print expression *e* to object *output* and sent a new line character too.

- (`'nl` *output*)

  Will sent a new line character to object *output*.

- (`'active` *output*)

  Succeeds only if the stream has not been closed.

- (`'close` *output*)

  Closes the stream.

A failure is printed as `%f`, an uninstantiated variable as `%w`, a class as `%(class` *NAME*`)`, and other objects as `%(object of class` *NAME*`)`.

3.4.3 Miscellaneous objects

These features are provided in the form of predefined classes. They can be used directly in the `global` object but have to be imported in other objects before they can be used. Some of these objects must have certain behavior when implemented on serial or parallel architectures.

Single Assignment Variable

Messages associated with class `sa-var` and its instances:

- (`'new` *sa-var value*)

  Will return a new single assignment variable with initial value *value*. If no initial value is given to the variable it is not initialized.

- (`'put` *sav value*)

  If *sa-var* has no value it is set to *value* and that value is returned, otherwise it fails.

- (`'get` *sav*)

  The value (current or future) in *sa-var* is returned.

- (`'reader` *sav*)

  Returns a version of the *sav* accepting only message `'get`.

- (`'writer` *sav*)

  Returns a version of the *sav* accepting only message `'put`.

Multiple Assignment Variable

Messages associated with class `ma-var` and its instances are similar to those for the single assignment variable except that new values can be given to it and it is initialized to nil.

Single Assignment Vector

A collection of single assignment variables identified by an integer number (the *position*). The collection has a defined *size*. Implementations of this object must provide constant time access to any element without respect on the size of it. Messages associated with class `vector` and its instances are:

- `('new` *vector size*`)`
  Will return a single assignment vector of size *size*. The first element has index 0.

- `('put` *sav position value*`)`
  Will put *value* in *position* of single assignment vector *sav*.

- `('get` *sav* `position)`
  Will return the value in *position* of single assignment vector *sav*.

Multiple Assignment Vector

Messages associated with class `ma-vector` and its instances are similar to those for the single assignment vector except that new values can be assigned to the same position and positions are initialized to nil. Implementations of this object must provide constant time access to any element without respect on the size of it.

Single Assignment Dictionary

Messages associated with class `dictionary` and its instances are similar to those for the single assignment vector except that the position is now a name instead of an integer number. Also, no size declaration is needed.

Multiple Assignment Dictionary

Messages associated with class `ma-dictionary` and its instances are similar to those for the single assignment dictionary except that new values can be assigned to the same position and positions are initialized to nil.

Top Level Caller

Class `top-level-caller` creates objects which accept the message `'top-level-funcall`. This message must be accompanied with arguments a function and the arguments of that functions. That function call is executed at the top level. Useful in the writing of monitors. This object must be completely parallel when implemented in parallel systems.

Fetch and Add

Class `faa` creates objects which accept the message `'faa`. The object initially represents the number `0`. Message `'faa` must be accompanied by a number y as an argument. When a message is sent to this object the number x represented by that object is returned and subsequently that object represents number $x+y$. This object must be completely parallel when implemented in parallel systems capable to support that.

Other Objects

Classes `semaphore`, `queue`, `putlist`, `getlist`, and proca are defined in ALLOY later in this manual. They are often useful. Other useful objects and functions can be found in the `progs` directory as explained earlier.

3.4.4 Miscellaneous

Variables

- `%f`

  Set to a failure value (e.g. produced by call `((mu() ))`.

- `%t`

  Set to a true (i.e. `'t`).

Boolean functions

- `(and` e_1 e_2 $\cdots$ e_n`)`

  Returns the conjunction of the arguments values as soon as possible.

- `(or` e_1 e_2 $\cdots$ e_n`)`

  Returns the disjunction of the arguments values as soon as possible.

- `(not` e`)`

  Returns the negation of the argument's value.

- `(and-call` c_1 c_2 $\cdots$ c_n`)`

  Returns the conjunction of the results returned by calling the arguments, as soon as possible. Kills incomplete calls as soon as possible.

- `(or-call` c_1 c_2 $\cdots$ c_n`)`

  Returns the disjunction of the results returned by calling the arguments, as soon as possible. Kills incomplete calls as soon as possible.

- `(equal` e_1 e_2 $\cdots$ e_n`)`

  Like `eql` but works for list as well.

Blocks

- `(block1` e_1 e_2 $\cdots$ e_n`)`

  Returns the value of the first parameter.

- `(blockn` e_1 e_2 $\cdots$ e_n`)`

  Returns the value of the last parameter.

- `(block` e_1 e_2 $\cdots$ e_n`)`

  Returns `t`.

Loops

- `(while` c_c c_e`)`

  Calls c_e if c_c when called succeeds. Repeats as soon as evaluation of c_e has completed.

- `(until` c_c c_e`)`

  Calls c_e unless c_c when called succeeds. Repeats as soon as evaluation of c_e has completed.

- `(repeat` n c_e`)`

  Calls c_e n times in parallel.

- `(srepeat` n c_e`)`

  Calls c_e n times serially.

System Functions

- `(include` *fname*`)`

  Includes the contents of the file as if it were typed from the terminal and read at the top level.

- `(load` *fname*`)`

  Like `include` but will inhibit printing.

- `(require` *property*`)`

  If *property* has been `provide`-ed nothing happens. Otherwise file *property*`.a` is loaded been found the file in the appropriate path.

- `(provide` *property*`)`

  Declares that *property* has been provided.

- `(add-require-path` *directory*`)`

  Adds *directory* to the path of files `provide`-ing properties.

- `(read)`

  Will read an expression from the console using object `input-console`.

- `(print` *e*`)`

  Will print the value of expression *e* to the console using object `output-console`.

- `(nl)`

  Will print a new-line to the console using object `output-console`.

- `(warning-exec)`

  Toggles on and off various warnings during execution.

- `(warning-load)`

  Toggles on and off various warnings during loading.

- `(exit)`

  Will exit ALLOY.

Implementation Specific Functions

- `(scheduler-switch-steps` *number*`)`

  Sets the number of context switches performed by the scheduler, before the scheduler

schedules a process in random. To enhance parallelism use a small number.

- `(warning-exec)`

  Will switch execution warnings on/off.

- `(trace)`

  Will switch tracing on/off.

- `(time-prompt)`

  Will switch on/off printing of execution times.

- `(stats-prompt)`

  Will switch on/off printing of statistics.

Chapter 4

Functions, Closures, and their Evaluation Modes

This chapter describes the functional part of ALLOY. It consists of two parts: closures and their evaluation modes.

ALLOY's commitment to simplicity and unification of features appears at this basic level too. Functions are closures, closures are objects, and objects can be used as functions. Methods are closures.

ALLOY closures provide commitment, and can behave as either functions or generators. Evaluation can be parallel or serial, eager or lazy, single or multiple solution. These modes can be combined freely.

The result is believed to be a more natural parallel functional environment than the impure and/or complex models that have been used in other languages of this type. Finally, the functional part of ALLOY is compared with some parallel functional programming languages.

4.1 Introduction

The design of ALLOY closures has been greatly affected by the assumption that a practical language using it would support object oriented programming. This consideration allowed closures to be simple, since features such as state, powerful streams, and other abstract data types are handled naturally with object oriented facilities.

In ALLOY, closures are just instances of the special predefined class `closure`. Special syntax is provided to simplify functions calls. Any object which accepts message `'eval` (like all closures do,) or an object which accepts message `'new` (like all classes do,) can be called as a function.

A closure encapsulates a number of function calls with the environment at the time of its creation. A commitment mechanism is available in closures which makes or-parallel execution possible.

A call (or message in general) can be made serially or in parallel. It can evaluate in an eager or lazy way. Finally, it can be requested to return one or many solutions. Chapter 3 should be consulted for the precise definitions.

Function calls requested to return all solutions are called *generators*. Generators are managed by *replicators*. The replication process is controlled by the environment which can make use of serial or parallel, eager or lazy evaluation modes. These abilities are described extensively in chapter 6.

Finally, ALLOY closures and evaluation modes are compared with analogous features in other programming languages.

4.1.1 Why another Functional Kernel?

The functional kernel of ALLOY is not simply an extension of Lambda Calculus but is different and significantly more powerful. Pure Lambda Calculus has some serious disadvantages:

- No generators, backtracking, or non-determinism. No facilities for searching.
- No ability to select between serial/parallel evaluation mode.
- No ability to select between eager/lazy evaluation mode.
- Cannot handle large data structures efficiently. Environment of closures cannot be updated.
- Can not express process inter-dependencies among functions (except through simulation).

Lambda Calculus has been extended during the design of many functional programming languages. One significant such extension was the *conditional expression* which removes the requirement for non-strict evaluation. Still, it is an extra feature for a specific

case of non-strict evaluation.

Languages which use strict evaluation can also provide delayed evaluation (e.g. using closures) but still do not provide lazy evaluation (possibly because of undesirable interactions with other extra features). Scheme is one such language.

Another important extension is the *assignment* expression. This ability to give to a variable its value after the variable has been defined allows one to create recursive functions in a simple way as well as to create simple abstract data types[1] with modifiable state. Unfortunately in this case variables are not single assignment any more.

When both serial and parallel evaluation techniques are supported, parallelism is usually provided in a way that can be weak (e.g. synchronization, mutual exclusion, or searching may be hard,) impure (e.g. parallelism is not just an evaluation mode,) or both. MultiLISP [Hal86] chose the impure way and allows arbitrary processes to determine the value of *futures*. Queue Based Multiprocessing LISP [GM84] provides an impure mechanism to kill, suspend, or resume an arbitrary process.

CFL [LS87] provides only parallel evaluation while serial evaluation is provided as the ability to compile a function using a different compiler.

While non-deterministic execution and backtracking did appear in some serial FPLs (i.e. LISPkit by Henderson [Hen80]), in parallel FPLs only non-deterministic evaluation has been incorporated (i.e. CFL by Shapiro [LS87]). Generators are not available in PFPLs to our knowledge. ICON, though not exactly functional, supports generators.

ALLOY closures and their associated evaluation modes attempt to provide a solution to the above problems. Together with classes (of more general objects), they form the programming language ALLOY. Special care has been taken so that ALLOY closures are kept small, concise and efficient.

4.2 Closures

ALLOY closures are objects, instances of class `closure`. This is a class which must be provided by ALLOY although it does not have to be part of the kernel. Closures provide normal, anonymous, and higher order functions. These functions support commitment

[1] Environments accessible only by some functions can then be modified by that function using `set`

and can return one or many solutions.

4.2.1 Functions

Naturally, since functions are objects, they are first class values. Although objects are formed by sending message `'new` to their class, closures are formed with a special expression. This is mostly for convenience but also because it makes creation of closures more static. If the ability to create closures with dynamically created bodies is found, in the future, to be desirable, then class `closure` will be allowed to create closures directly.

Closures can be formed by using an expression like `(mu` $formals\ call_1\ call_2\ \ldots\ call_n$`)`. Expression $formals$ is matched against the *actual* arguments of the call. Any names appearing in $formals$ are treated as variable patterns. Constructor `mu` is used since μ is the letter after λ in the Greek alphabet.

Functions can fail but if they succeed they can also return a result. During evaluation of a function's body the function can *commit* evaluation of its body to *return* a result. A function commits[2] when a `return` statement is reached at which point all processes created by that function, except those created by the `return` statement, are killed. Commitment can be *local* (`lreturn` statement) or *global* (`return` statement). Local commitment commits evaluation of the innermost function call whereas global commitment commits evaluation of the innermost global-function[3] call.

Functions can fail explicitly with a `(fail)` special expression. There is a local form of this statement as well. A function called normally can also signal failure by `return`ing a failure value. Functions called as generators however, can only signal failure with the `(fail)` special expression.

4.2.2 Environments and Values

Variables can be declared as arguments of functions. They are single assignment and *synchronizing*. Such a variable is always given a value during parameter passing. A function call returns the *future* of its result. At some later point in time, this future may

[2] This feature makes possible the definition of objects such as `proca` (chapter 2) which allows for the creation and termination of an arbitrary processes. A process can only be suspended and resumed when appropriately written.

[3] Global functions are functions whose definition is not local to another function. Such functions are functions defined at the top level of a class.

become an actual value. When a process needs to examine a future value it suspends until the actual value is available[4].

The scope of variables is static. Thus, a function created in the scope of a variable has access to it. Type checking is dynamic. Structures can be created dynamically but cannot be destroyed or modified[5] but their space is *garbage-collected* when they cannot be accessed any more.

The top level of ALLOY is also an object. It is assumed to have definitions for all possible variables. These variables form the state of the top-level object and like other state variables they can only be initialized using the `set` expression (see chapter 5 subsection 5.1.1).

4.3 Evaluation Modes

In ALLOY, a program is a set of class declarations and message passing operations. This section describes how a message can be sent or a function call be made, and shows how to select the style of evaluation.

4.3.1 Message passing and Function Calls

To send a message with some arguments to an object the syntax is a list with first element the name of the message (must be a symbol,) second the object itself, followed by the arguments of the message.

When the message name is omitted, in which case the first element of the message is not a symbol, it is assumed that the message name is `'new` or `'eval` depending on whether it is sent to a class or not. Thus, closures and other objects can be called in a more *functional* style. Since function calls are just special cases of messages, the terms *message* and *function call* will be used without distinction, unless otherwise specified.

The arguments of function calls can be evaluated serially or in parallel. A function can commit during evaluation of its body into returning a result or not. Functions can be called as generators. Functions can evaluate in an eager or lazy way.

[4] The actual value may be formed in terms of future values too.
[5] Unless they contain objects whose state can change.

4.3.2 Evaluation Modes

In a message (i.e. list in a position of evaluation) whose elements are surrounded by parentheses (which we sometimes refer to as round brackets) the elements are evaluated in parallel. As soon as the body of the function which must evaluate the message is known, the body is evaluated. In a message whose elements are surrounded by square brackets the elements are evaluated serially. As soon as evaluation of all elements is complete, the respective function is called. Thus, calls made in round brackets result in a more natural unrestricted evaluation.

Normally expressions evaluate in an eager way. However expressions evaluated in an expression as in `(lazy` *e*`)` evaluate lazily. The body of a function always evaluates in an eager way until commitment. If the call was lazy, the expression whose value the function is committed to return, is evaluated lazily. This complication is necessary because of the non-deterministic behavior of a function before commitment. Expressions evaluating lazily evaluate only as much as is needed to produce the part of the result that needs to be examined at the time. Then they suspend until a new part of the result is needed.

4.3.3 Generators

Functions can be called as generators by prefixing the call by the operator `*`. In that case, a function does not commit. Thus, a function can return many results. Where a function would fail when called normally, if called as a generator it would not return any result. Generators exist in a *replicable* environment. That is a call prefixed by the operator `#`. Any value is a generator of itself but a failure generates nothing.

An expression prefixed with the replicator operator `#` is replicated once for each set of values of the (first level) generators in the expression. The value of a generator is *needed* if it is necessary for the replication to take place. Unless the value of a generator is needed, the `return` statement which provided it is not considered to be finished. This makes possible for the replicating process to control the generator (thus making backtracking possible).

ALLOY does not need any conditional expressions. The replicator is powerful enough to express conditions. The idea is that we use the value of the condition as a generator and since a *failure* generates nothing we can replicate and execute an expression only when

a condition is true. More details on generators are given in chapter 6.

4.4 Macros for Local Environments

Two expressions, `lets` and `let` are available in ALLOY. As in LISP, they simplify automatic creation-and-call of closures. In other words, they simplify creation of local environments.

In addition, these two macros are made to create uninstantiated variables when no value is given to them. In such cases the macro expansion is more complicated. For each let expression an extra class must be created whose instances (objects) are given all the accessible variables. These objects contain the actual body of the let expression. The uninstantiated variables are state variable of the object. These variables can then be initialized with the `set` expression as in `(set` *formals expression*`)` using pattern matching as with parameter passing. The `set` expression is the only way to initialize state variables of objects.

However, when the macros are defined at a lower level as they are in the current implementation, the implementation of these expressions is simpler. The expansion makes use of a special value. During the pattern matching of the function call with the closure representing the `let` expression, the pattern matcher recognizes the special value and leaves the respective variable uninstantiated. This special value is only known to the macro expander and the pattern matcher.

4.4.1 The `lets` expression

Expression `lets` is very similar to the `let` macro in LISP, except that `lets` requires explicit `lreturn` statements to return a result. In ALLOY, the macro expansion is somewhat more complicated since `lets` must handle combinations of serial and parallel execution.

The following `lets` expression:

$$({}^{b}\text{lets}\ \ ({}^{l}(p_1\ e_1)\ \cdots\ (p_i\ e_i)\ \cdots\ (p_n\ e_n))^{l}\\ \quad b_1\ \cdots\ b_i\ \cdots\ b_n)^{b}$$

is expanded to:

$$({}^{l}({}^{b}\text{mu}(p_1\ \cdots\ p_i\ \cdots\ p_n)\quad b_1\ \cdots\ b_i\ \cdots\ b_n)^{b}\\ \ e_1\ \cdots\ e_i\ \cdots\ e_n)^{l}$$

In this transformation brackets are marked with indexes. All brackets with the same index are of the same kind (the two kinds being round and square).

As it is clear from the expansion, the new variables defined by the `lets` expression are not available to the expressions which give them values. To define recursive functions at that point one should define uninstantiated variables and use a `set` expression in the body.

4.4.2 The `let` expression

The second form, `let`, has semantics closer to the ones of `let` in LISP, but its expansion is even more complex than the expansion of `lets` because of the need to behave as a generator[6] when its body is a generator.

$$({}^{b}\text{let}\ \ ({}^{l}(p_1\ e_1)\ \cdots\ (p_i\ e_i)\ \cdots\ (p_n\ e_n))^{l}\\ \quad b_1\ \cdots\ b_i\ \cdots\ b_n)^{b}$$

is expanded to:

$$({}^{l}({}^{b}\text{mu}(p_1\ \cdots\ p_i\ \cdots\ p_n)\\ \quad \#(\text{lreturn}\ *({}^{b}({}^{b}\text{mu dummy}\ \#(\text{lreturn}\ b_n))^{b}\\ \quad\quad b_1\ \cdots\ b_i\ \cdots\ b_{n-1})^{b}))\\ \quad e_1\ \cdots\ e_i\ \cdots\ e_n)^{l}$$

4.5 Comparisons

This section compares the functional part of ALLOY, closures and evaluation modes, with some parallel functional programming languages.

4.5.1 ZLISP

ZLISP [Dim88] provides very low level features such as a number of engines, usually equal to the number of available processors, explicit process creation to feed the engines, and engine interruption. These are very powerful features, but also very low level.

[6] Since a generator can be controlled, serial and parallel evaluation mode must be preserved.

Consider writing a parallel-or function which would evaluate its arguments in parallel and would terminate all arguments as soon as one of them terminates successfully. The following program makes use of a more powerful scheduler [Mit87] than the one provided in ZLISP by default:

```
(defstruct par:or:syncrons
    (par:or:start     (make-ultraint 0))
    (par:or:solution  (make-ultraint 0))
    (par:or:end       (make-sema))
    par:or:nofailed
    par:or:procs
    par:or:result)


(defmacro par-or nmacro(all)
    (declare (zLISPopts noredefine norttest))
    (cond ((endp all) nil)
          ((endp (cdr all)) (car all))
          (t (quote! car ! (par:sys:or:n all)))))

(defmacro par-or-n nmacro(all)
    (declare (zLISPopts noredefine norttest))
    (cond ((endp all) '(nil . 0))
          ((endp (cdr all)) (quote! cons ! (car all) 1))
          (t (par:sys:or:n all))))

(defun par:sys:or:n (all)
    (declare (zLISPopts static norttest))
    (list 'prog
          '( (par:sync (make-par:or:syncrons)) )
          (list 'set-par:or:nofailed 'par:sync
                (list 'make-ultraint (- (length all) 1)))
          (list 'set-par:or:procs 'par:sync
                (cons 'list (par:sys:create-call 'par:or:call all)))
          '(free-v (par:or:start  par:sync))
          '(wait   (par:or:end    par:sync))
          '(return (par:or:result par:sync))))

(defun par:or:call lambda(n sync this-result)
    (declare (zLISPopts noredefine norttest))
    (cond (this-result
              (when (tir (par:or:solution sync) 1 1)
                    (free-b (par:or:start sync))
                    (par:sys:kill-rest n (par:or:procs sync))
                    (set-par:or:result sync (cons this-result n))
                    (signal (par:or:end sync))))
          (t (unless (tdr (par:or:nofailed sync) 1)
                     (set-par:or:result sync (cons nil 0))
                     (signal (par:or:end sync))))))
```

Its is clearly a complex and involved program.

Here is the implementation in ALLOY:

```
(setfun (or-call . l)
   (when (cons-p l) (lets (rv xv)
                          [when (set xv ((car l)))           (return xv)]
                          [when (set rv (or-call . [cdr l])) (return rv)])))
```

Its simplicity is based on the powerful built in commitment mechanism. The behavior of the two programs is precisely the same.

4.5.2 MultiLISP

MultiLISP started very modestly and has efficient implementations [KHM89,Hal84]. Real lazy evaluation is not possible. It was possible to create processes or delay evaluation but that was all. Those processes could not be managed in any other way (e.g. terminated). Now, MultiLISP has been fortified with low level features for process management [Hal86]. It is now possible to *determine* the value of a future and kill the process which was supposed to resolve it. Thus, MultiLISP seems to share some of ZLISP's limitations. Similarly MultiLISP does not attempt to provide generators (much less parallel ones).

Consider the above parallel-or function but for two arguments only.

```
(defmacro par-or(Expr1 Expr2)
  `(let ((x nil) e1 e2)
     (setq x (future (or (future (if (setq e1 ,Expr1) (determine x e1)))
                         (future (if (setq e2 ,Expr2) (determine x e2))))))
     (touch x)))
```

Even though the solution is simple we claim that the `determine` operation is dangerous because of its arbitrary power over any future.

Here is ALLOY's solution for two arguments (no macros):

```
(setfun (or-call e1 e2)
   (lets [(r1 (e1))] (when r1 (return r1)))
   (lets [(r2 (e2))] (when r2 (return r2))))
```

Since ALLOY does not have macros, the arguments must be prefixed with operator ^ so that they become closures (see chapter 3).

The above solutions have the property to wait until one process has not only succeeded but also completed execution completely. One might claim that if we wanted to terminate the other processes as soon as one starts to succeed we could do that more easily in

MultiLISP because of the power of `determine`. However, in ALLOY, the above can be achieved by using object `proca` (see chapter 2) which can be used to create a process and later kill it by sending a message to it. The only difference would be that for a process to be controlled by object `proca` it must be created by it, which in our opinion is a significant advantage.

This follows ALLOY's philosophy that simple things should be done simply while it should not be hard to create the appropriate tools to make harder (and probably rarer) tasks easy too.

4.5.3 Queue Based Multiprocessing LISP

This language extends LISP with qlambda closures in addition to the lambda ones. Qlambda closures are assigned processes which execute their calls thus guaranteeing mutual exclusion of processes attempting to execute it. This mechanism to protect data seems crude and too serial when compared to the synchronizing variables of ALLOY parallel objects.

QBML also provides very low level facilities for suspending, resuming or terminating arbitrary processes. Termination in ALLOY is safer and easier to use in most cases. Suspension and resumption however must be done by giving to the process an object and having it suspend or resume by sending messages to it. Though in the general case this can be hard, in practice it is not a problem. It is often transparent.

Here is the parallel factorial in QBML:

```
(setfun pfact(n)
   (return (pfact2 1 n)))

(setfun pfact2(from to)
   (if (ge from to) from
      (let ((mid (quotient (sum from to) 2)))
        (qlet 'eager ((f1 (pfact2 from mid)) (f2 (pfact2 (sum mid 1) to)))
           (times f1 f2)))))
```

It is even simpler in ALLOY:

```
(setfun pfact(n)
   (return (pfact2 1 n)))

(setfun pfact2(from to)
   (return (if (ge from to) from
               (let ((mid (quotient (sum from to) 2)))
```

```
(times (pfact2 from mid) (pfact2 (sum mid 1) to))))))
```

4.5.4 Concurrent Functional Language

This language [LS87] is translated into Guarded Horn Clauses [Ued86] first and into Flat Concurrent Prolog (FCP) [Sha83b] eventually. Some of its advantages over FCP are given: a) the programmer does not have to deal with input output annotations which are needed in FCP. b) The language has both "downward" and "upward" closures. c) Using different pre-processors, different evaluation techniques are immediately available (serial/parallel eager/lazy).

For a simple functional program such as solving the towers of Hanoi problem, CFL is adequate [LS87]:

```
Hanoi(N, A, B) <-
          if N = 0 then cons(A, B)
          also if N > 0 then
               { let C = free(A, B) and N1=N-1 in
                   cons(hanoi(N1, A, C)), cons(cons(A, B),
                                               hanoi(N1, B, C))) }.

delete(E, L) <-
          if L = [] then []
          also if L = [E | T] then T
          else cons(car(L), delete(E, cdr(L)).

free(A, B) <-
          car(delete(A, delete(B, '[a, b, c]))).
```

The syntax of logic programming languages in CFP is evident. Of course the algorithm is not very efficient since functions `free` and `delete` are unnecessary.

The solution in ALLOY is just as simple:

```
(setfun hanoi(n a b)
    (when (eq n 0) (return (cons a b)))
    (when (gt n 0) (return (let ((c (free a b)) (n1 (diff n 1)))
                             (cons (hanoi n1 a c)
                                   (cons (cons a b)
                                         (hanoi n1 b c))))))

(setfun delete(e l)
    (when (null-p l) (return nil))
    (when (cons-p l) (return (if (eql e (car l)) (cdr l)
```

```
                                        (cons (car l) (delete e (cdr l)))))))
```

```
(setfun free(a b)
    (return (car (delete a (delete b '(a b c))))))
```

More efficiently, and to return the movements in a flat list:

```
(setfun hanoi(n a b c rest)
    (when (eq n 1) (return (cons (cons a b) rest)))
    (when (gt n 1) (return (let ((n1 (diff n 1)))
                             (hanoi n1 a c b
                                    (cons (cons a b)
                                          (hanoi n1 c b a rest)))))))
```

In more complicated problems, however, CFL is less effective. Serial/parallel or eager/lazy evaluation cannot be combined freely in CFL. The commitment mechanism is weak (i.e. exceptions are not possible). Generators or multiple solutions are not available. Object oriented programming or parallel classes are missing. For example, it is not possible to solve problems like those solved by classes such as `coex` for co-expressions or `proca` for process management. Solving the n-queens problem in a simple way is not possible either.

4.6 Summary

The functional part of ALLOY has been presented. It has two parts: closures and evaluation modes.

Closures are simple (restricted) objects accepting message `'eval` augmented with static scope. For convenience and to make closure declaration static, closures can only be created by a special expression (similar to that in LISP). Closures, as well as objects accepting the `'eval` message, can be invoked in a functional way.

Evaluation modes support serial or parallel execution, eager or lazy evaluation, and single or multiple solutions. Evaluation modes can be combined freely. It is shown that commitment in combination with these powerful evaluation modes is superior in simplicity and expressiveness (except in expressing suspension and resumption of arbitrary processes) to similar features of other languages.

Chapter 5

Objects and Interprocess Communication

When processes are light-weight, ease of expressing efficient inter-process communication is of utmost importance. At the same time, complex interprocess communication should be strong. For efficiency and clarity, ALLOY treats these two problems separately.

In ALLOY, variables are *synchronizing*. These variables, while efficient compared to logical variables, are powerful enough for basic synchronization. Their single assignment nature, static scope, and synchronizing abilities make them clean and easy to use.

Powerful inter-process communication is provided by user definable parallel objects. Parallel objects are possible because of the way they manage synchronizing variables. The mechanism is said to create *dynamic* synchronizing variables. The dynamic synchronizing variable is shown to be sufficiently powerful for the creation of parallel objects.

5.1 Introduction

This chapter describes the mechanisms which the language ALLOY provides for basic and complex interprocess communication. Basic communication is achieved with the synchronizing variable. Complex synchronization is possible with the support of object oriented programming in this parallel programming environment. Messages in ALLOY can be processed by an object in parallel. An object can have many states active at the same time while these states communicate through synchronizing variables.

Mechanisms are provided making possible the expression of fine-grain and efficient synchronization during the concurrent service of messages. The basic mechanism for message passing is synchronous but asynchronous communication comes for free since the language is inherently parallel.

Dynamic synchronizing variables makes it possible for an object to have many states active at one time, which can execute without any conflicts. At the same time, fine-grain synchronization between successive instances of the object may be enforced by these variables.

Finally, we provide a comparison of these features with analogous features in other programming languages. Chapter 3 should be consulted for the precise definitions.

5.1.1 Classes

Every value in ALLOY is an object. This means that even classes and messages are first class values. During the definition of a class one can define static[1] variables which are conventional variables, dynamic variables explained later, objects common to all instances of a class, variables imported from the top-level object, the methods, and subsets of methods as abstract interfaces to instances of the class.

Both static and dynamic variables (i.e. state variables) provided to an object are not initialized[2]. They can only be initialized by the object, using the `set` expression. An attempt to access an uninstantiated state variable returns the *future* value of that variable. An attempt to examine a future value is suspended until the *future* value is known.

Objects support delegation with the usual meaning for variables `self` and `here`. That is, variable `here` always contains the current object while variable `self` contains the last object which did not receive the current message through delegation. Objects can `resend` messages instead of delegating them and then the value of variable `self` is not preserved.

Objects, instances of the same class, can share *common* objects. Finally, objects can have many interfaces. More details are given in chapter 3.

[1] Static variables can also be defined indirectly by using them during the definition of the class.
[2] This is the only way to create non-initialized variables in ALLOY.

5.2 Single item communications

Single item interprocess communications are by far the most frequent in a fine grain parallel programming language. For that reason, ALLOY provides synchronizing variables.

5.2.1 One to one (shared) Communication

Synchronizing variables are very powerful and allow ALLOY to support fine grain parallelism without explicit synchronization. These variables were deemed to be preferable to logical variables [CG83,Sha86a] since the latter are too powerful and provide both normal and back communication in an unpredictable way. Channels [CG81] were not selected since they also provide back communication (even though limited). Futures [Hal86] were not satisfactory either, as they are properties of values rather than of variables and therefore are not sufficient in an environment using uninstantiated variables.

Synchronizing variables were selected to be powerful enough to satisfy the most common synchronization needs in a parallel object oriented environment. But they are not too powerful to have simple and clear semantics in a functional environment. The following example combines (though not in a very useful way) the two environments:

```
ALLOY > (let (x y) (set y (sum x 4)) (set x (sum 1 2)) (sum x y))
==> 10
```

5.2.2 Many to one (shared) Communication

Objects provide the programmer with more powerful interprocess communication for demanding applications. In this way programs become both efficient and easy to understand as powerful and ambiguous features are avoided.

Function `gvalue` returns one of these useful library objects. In a way, *logical_variable* = *synchronizing_variable*+*gvalue*. Function `gvalue` is defined as follows:

```
(setfun gvalue()
   (return (let ((v ('new sa-var)))
             (list ('writer v) ('get v)))))
(end-class gvalue)
```

It returns a list of two elements: a single assignment variable that can only be written upon and its value. Thus, the `gvalue` object enforces strict back-communication.

The following example is an implementation of the `bubble-up` function which surfaces the n^{th} element of a list at its head. Auxiliary function `delete-nth` deletes the n^{th} element of the list, returns the resulting list and back-communicates the deleted value through the `gvalue` object:

```
(setfun bubble-up(n l)
   (return (let (((gval value) (gvalue)))
             (cons value (delete-nthx n l gval)))))

(setfun delete-nthx(n l v)
  (when (cons-p l)
     (return (if (le n 1)
                 (block1 (cdr l) ('put v (car l)))
                 (cons (car l) (delete-nthx (diff n 1) (cdr l) v))))))
```

In the above program the `gvalue` object is assigned to variable `gval` and its future value to variable `ditem`. Note that the value represented by the object in `gval` can not be read using that object. That object can only be used to write the value once.

5.2.3 Communication to Many

In the above situations communication to one process is called *shared*. It is called shared because synchronizing values can be read by many processes. Thus, the above techniques can also be used for communication to many processes. Whether communication is to one or many processes depends on whether the synchronizing value has been given to one or many processes respectively.

5.2.4 Inter-message Communication

Dynamic synchronizing variables are used to provide a transparent mechanism for communication among messages to the same parallel object.

The state of an object is described with variables which can be either *static* or *dynamic*. Static variables are variables which have one instance in every object. These are the kind of variables normally used to describe the state of an object in programming languages.

Dynamic variables are special in that they have a new instance with every message sent to the object of whose state they are part. Every message has access to all static variables of that object and also to its own instances of dynamic variables together with the instances of the dynamic variables of the next message. That latter instance is ac-

cessed as `next-`*var_name*. Variables, whether static or dynamic, can be single or multiple assignment.

One of the simplest examples that makes use of the dynamic synchronization variables is the following. It creates a *counter* object which can be read and incremented by many processes at the same time. Even though keeping a global counter is an exclusion problem and not one of synchronization, it is used here to show the versatility of the dynamic synchronizing variables.

```
(class-start counter
  (dynamic v)
  (import number-p sum)
  (methods inc))

(setfun new(x)
  (if (number-p x) (set next-v x) (set next-v 0)))

(setfun inc(i)
  (return (let () (set next-v (sum v i)) v)))

(class-end counter)
```

A more complicated example is that of a *mostly parallel* stack. The stack defined next can be accessed by many processes in parallel. Any number of `push` messages can be processed *entirely* concurrently. A `pop` message has to wait for the last[3] message to finish its action, so `pop` operations are not very parallel.

```
(provide "estack")

(class-start eager-stack
   (dynamic head)
   (methods push pop))

(setfun new()
   (set next-head %f))

(setfun push(x)
   (return (block1 x          (set next-head (cons x head)))))

(setfun pop()
   (return (block1 (car head) (set next-head (cdr head)))))
(class-end eager-stack)
```

An attempt to `pop` from an empty stack results in a failure.

[3] Messages before the last one do not have to finish execution.

5.2.5 Parallel Architectures

Dynamic synchronization variables are important not only because they allow the easy description of parallel objects but also because they allow the utilization of underlying highly parallel primitives supported by the system. In particular, shared memory MIMD systems supporting the highly parallel `fetch-and-add` primitive [GLR83][4] can create any number of dynamic variables concurrently.

This can be achieved as follows. Let us assume that the object is meant to accept at most n messages at the same time. This number need not be large since it restricts only the number of concurrent acceptance (duplication of dynamic variables) of the message and not the processing of it. A small array of size n is needed by that object. Each slot of the array can keep a set of dynamic variables and two flags `empty` and `full`.

A parallel index is used to access the next available slot in the array. A message arriving first increments that index using `fetch-and-add`. That index modulo the size of the array points at the current slot. The message waits on the `empty` flag of that slot. Then it creates and places in that slot a set of dynamic variables. After that, setting the `full` flag it declares that the slot is filled. Finally, it waits on the `full` flag of the previous slot until that is also filled.

As soon as the previous slot is filled (by the previous message), the current message takes those dynamic variables and also the `next` dynamic variables it created itself and placed in the current slot, and starts execution. The previous slot is then freed by setting its `empty` flag and re-setting its `full` flag.

If a process can prevent context switches during the creation of the dynamic variables [ELS88], busy waiting would be virtually non-existent. Otherwise, messages can measure the length of busy waiting and when large, lock the object, replace the flags by blocking binary semaphores, and release the object. That would remove any trace of busy waiting (provided the blocking queues are free of busy waiting).

[4] `fetch-and-add` is an atomic operation which allows a processe to fetch the value of a variable and add to it another value. The execution time of the operation is not affected by the number of processors executing the operation on the same variable.

5.3 Stream communication

Streams are often the fundamental connecting media of networks of communicating processes. Synchronizing values can be used to provide the basic form of streams.

5.3.1 One to one (shared) Communication

The basic stream is an incomplete list (a list whose end is a synchronizing variable). This list can be incrementally given more elements by the process which has control of the list's tail. It can be read by any processes having access to its head. This is clarified in the following example:

```
(setfun hamming()
  (return (let (ham)
            (let ((mults (mu(c) (lreturn
                                  (mapcar (mu(x) (lreturn (times c x)))
                                          ham)))))
              (set ham (cons 1 (amerge (mults 2)
                                       (amerge (mults 3) (mults 5))))))))))

(setfun amerge(l1 l2)
   (when (lt (car l1) (car l2))
         (return (cons (car l1) (amerge (cdr l1) l2))))
   (when (gt (car l1) (car l2))
         (return (cons (car l2) (amerge l1 (cdr l2)))))
   (when (eq (car l1) (car l2))
         (return (cons (car l1) (amerge (cdr l1) (cdr l2))))))
```

Function hamming returns, sorted, all numbers of the form $2^i 3^j 5^k \forall i,j,k > 0$. For that, it creates three processes each of which multiplies all elements in the input stream by 2, 3 and 5 respectively and returns the result in a stream. Resulting streams are merged arithmetically and the final stream is fed as input to the three processes.

5.3.2 Many to One (shared) Communication

It is often necessary to have the stream be incremented by many processes. For that reason the following object `putlist` returns a stream whose tail is known to the object only. Thus, elements can be added to the list only by those processes which have access to the `putlist` object.

```
(provide "putlist")
```

```
(class-start putlist
   (static  _orig-head)
   (dynamic head)
   (methods put end get-orig-head)
   (interface puter   put end))

(setfun new()
   (set _orig-head next-head))

(setfun put(x)
   (return (let() (set head (cons x next-head)) x)))

(setfun end()
   (return (set head nil)))

(setfun get-orig-head()
   (return  [let(h)  (set head next-head) (set h _orig-head)
                     (set _orig-head nil) h]))

(class-end putlist)
```

An example of this object is the following implementation of the asynchronous systolic-sort algorithm. It sorts n elements in $O(n)$ time if $O(n)$ processors are available.

```
(require "putlist")

(setfun ssort(l)
   (return (if (null-p l) nil
               (let ((rest ('new putlist)))
                  (cons (ssortx (car l) (cdr l) rest)
                        (ssort ('get-orig-head rest)))))))

(setfun ssortx(x l rest)
   [when (null-p l) ('end rest) (return x)]
   (when (cons-p l)
      [when (gt (car l) x) ('put rest (car l))
                           (return (ssortx x       (cdr l) rest))]
      [when (le (car l) x) ('put rest x)
                           (return (ssortx (car l) (cdr l) rest))]))
```

Function `ssort` takes as argument a list of numbers and returns it sorted. For that, it calls auxiliary function `ssortx` which returns the smallest number and places the rest numbers in object `rest`. The stream represented by that object is then sorted recursively by `ssort`.

Function `ssortx` takes three arguments: a number, a list of numbers and a putlist object. It returns the smallest of the numbers and places the rest in the `putlist` object `rest`. The stream represented by that object is sorted recursively by the `ssort` function.

5.3.3 One to Many Communication

Although the streams created in previous paragraphs can be accessible by many processes, it is often desirable to *consume* the stream rather than just reading it. In the latter case, each value in the stream is taken by one process only.

This kind of an stream can be obtained by giving the head of a regular stream to an object. That object can then serve the various consumers by giving them each successive elements from the original stream. The `getlist` object does that:

```
(provide "getlist")

(class-start getlist
   (dynamic head)
   (methods get))

(setfun new(tail)
   (set next-head tail))

(setfun get()
   (return (let ()
             (set next-head (cdr head)) (car head))))

(class-end getlist)
```

5.3.4 Many to Many Communication

The simple parallel queue is the basis of communication between multiple producers and multiple consumers. This object can be implemented by combining the output of a `putlist` objects with the input of a `getlist` object:

```
(provide "queue")
(require "putlist")
(require "getlist")

(class-start queue
   (static  puts gets)
   (import  putlist getlist)
   (methods enqueue dequeue)
   (interface enqueuer enqueue)
   (interface dequeuer dequeue))

(setfun new()
   (set puts ('new putlist))
   (set gets ('new getlist ('get-orig-head puts))))
```

```
(setfun enqueue(x)
   (return ('put puts x)))

(setfun dequeue()
   (return ('get gets)))

(class-end queue)
```

Note that any number of elements can be placed in this queue entirely in parallel. However an element cannot be removed from the queue until the previous element has been removed. More parallel queues can be implemented with vectors and highly parallel indexes such as the `faa` object [GLR83]. If the architecture can support a parallel fetch-and-add then it is required that class `faa` should be implemented at a low level and its instances be completely parallel.

5.4 Examples

A *lock* requires many keys to be open. Otherwise it is closed. There can be many locks with different subsets of the same set of keys. Opening many locks at the same time is guaranteed to be a fair and deadlock free operation.

First an implementation of a blocking semaphore is given, then that of a *key* which is a semaphore with an id. Then the lock itself, and finally a simple solution to the 5 dining philosophers, using the locks. The solution to the problem of the dining philosophers is starvation and deadlock free even though it does not make use of any special techniques.

5.4.1 Semaphore

The following program implements a counting semaphore. Messages sent to it can be processed in parallel and without serialization except that `'p` messages cannot be processed until the previous message has completed execution.

```
(provide "semaq")
(require "queue")

(class-start semaphore
  (import queue data repeat)
  (static q)
  (methods p v))
```

```
(setfun new(n)
  (set q ('new queue))
  (vs (if n n 0)))

[setfun p(n)   (ps n) (return %t)]

[setfun v(n)   (vs n) (return %t)]

(setfun ps(n)  (repeat (if n n 1) ^(data ('dequeue q))))

(setfun vs(n)  (repeat (if n n 1) ^('enqueue q 't)))

(class-end semaphore)
```

5.4.2 Key

A key is a semaphore with an id (a unique number). One can ask for a key and suspend until the key is available. One may return a key. Also, one can ask for the id of the key, which is a unique number among all keys. The same messages `'p` and `'v` are accepted by a key:

```
(provide "key")
(require "faa")
(require "semaq")

(class-start key
  (import semaphore)
  (common faa)
  (static s id)
  (methods p v id))

(setfun new(n)
  (set s ('resend semaphore))
  (set id ('faa c-faa 1)))

(setfun id() (return id))

(setfun p()  [return ('resend s)])

(setfun v()  [return ('resend s)])

(class-end key)
```

Note the use of an object (`faa`) common to all instances of the class. It ensures that each object will have a unique id. Also, messages with their parameters are resent to the actual

semaphore with the `'resend` message. In this case, since variable `self` is not used by the semaphore, the message could also be `'delegate`-ed.

5.4.3 Lock

A lock opens as soon as all its keys are available. A lock is defined to require a set of keys during its creation. The operations for unlocking and locking a lock are again `'p` and `'v`.

As soon as the lock is requested to open, it starts collecting the needed keys in a fair and deadlock free manner. When it is locked, all keys are freed.

The keys are

```
(provide "lock")
(require "key")
(require "qsortcar")

(class-start lock
    (static ssorted)
    (import key qsortcar and mapcar)
    (methods p v))

(setfun new l
   (let ((skey (if (all-key l)
                   (qsortcar (idsemalist l)) ())))
      (set ssorted (mapcar cdr skey))))

(setfun idsemalist(l)
   (return (if (cons-p l) (cons (cons ('id (car l)) (car l))
                                (idsemalist (cdr l))) nil)))

(setfun all-key(l)
   (if (cons-p l)
      (if (and (eql ('object-p (car l)) 'key) (all-key (cdr l)))
         (return ())
         (return %f))
      (return ())))

[setfun p() (msg-all ssorted 'p) (return)]

[setfun v() (msg-all ssorted 'v) (return)]

(setfun msg-all(ls msg)
   (when (cons-p ls) (msg (car ls)) (msg-all (cdr ls) msg)))

(class-end lock)
```

In order for a lock to gather the necessary keys without creating a deadlock, it collects

them in increasing order of their ids. Thus circles of waiting lines are avoided. Starvation is avoided because once a key is taken it is not returned until the lock has opened and then closed again

5.4.4 N Philosophers

The following program provides a solution to the problem of the *N philosophers*. The following general solution is starvation and deadlock free:

```
(require "key")
(require "lock")

(class-start philosopher
             (static id forks)
             (import gt diff printnl lock)
             (methods live))

(setfun new(f1 f2 i)
   (set id i) (set forks ('new lock f1 f2)))

(setfun live(n)
   (loop n))

[setfun loop(n)
   ('p forks)
   [printnl (list "Philosopher" id "           took both forks")]
   [printnl (list "Philosopher" id "started eating")]
   (eat 2)
   [printnl (list "Philosopher" id "stopped eating")]
   [printnl (list "Philosopher" id "           put down both forks")]
   ('v forks)
   (if (gt n 0) (loop (diff n 1)))]

(setfun eat(n)
   (if (gt n 0) (eat (diff n 1))))

(class-end philosopher)
```

And the rest of the algorithm:

```
[setfun dine(p n)
   [printnl "The big feasting starts..."]
   (dinen2 (if p p 5) (if n n 3))]

(setfun dinen2(p n)
   (let ((fork1 ('new key 1)) create)
      (set create (mu (previous i)
```

```
                    (if (gt i 1)
                       (let ((f ('new key 1)))
                          ('live ('new philosopher previous f i) n)
                          (create f (diff i 1)))
                       ('live ('new philosopher previous fork1 i) n))))
      (create fork1 p)))
```

In this algorithm a fork is represented by a key. Each pair of forks potentially needed by a philosopher are placed in a lock accessible only by that philosopher. For a philosopher to eat it is sufficient to ask for its lock to open.

5.5 Default Messages and Modules

Modules provide only control abstraction. Some times they are convenient because their use requires only one piece of information (i.e. the function) whereas using an object requires more information (i.e. message and object).

The default message mechanism can be used to achieve the same simplicity of use in objects which accept one (important or only) message. The `counter` example in chapter 2 demonstrates this.

The convenience of stateless modules which export many important functions can be achieved by defining all the functions into a `lets` environment. The names of local functions should be given local uninstantiated definitions. Then, the exported functions become known outside of this environment but the local functions are hidden.

5.5.1 Bubble-Up

An example of module-like ease of use in the general case, when many functions are exported, is the abstracted `bubble-up` function:

```
(lets (delete-nthx)
   (setfun bubble-up(n l)
      (lreturn (let (((gval value) (gvalue)))
                  (cons value (delete-nthx n l gval)))))

   (setfun delete-nthx(n l v)
      (when (cons-p l)
         (lreturn (if (le n 1)
                     (block1 (cdr l) ('put v (car l)))
                     (cons (car l) (delete-nthx (diff n 1) (cdr l) v)))))))
```

Function `delete-nth` is now hidden. More than one function can be exported by not having a local name for it.

5.6 Objects with Perpetual Processes

An object is not available for use unless the `'new` message has completed execution. This is in order to avoid synchronization problems in cases where the object makes use of static multiple assignment objects. However this makes it harder for this message to create processes which are active during the lifetime of the object. To facilitate this, class `top-level-funcall` is defined in ALLOY and is available as a library.

5.6.1 Co-expressions

To demonstrate use of the `top-level-funcall` class we give the definition of an object transforming generators into co-expressions. Message `'new` creates a process to evaluate the values of the generator:

```
(provide "coex")

(class-start coex
   (common top-level-caller)
   (dynamic latest)
   (methods next))

(setfun new(closure)
   ('top-level-funcall c-top-level-caller
                       ^(set next-latest (lazy (list #*(closure))))))

(setfun next()
   (return (block1 (car latest) (set next-latest (cdr latest)))))

(class-end coex)
```

This is a generator used to test it:

```
(setfun ints2p(x y)
   [print 0]
   [list (lreturn x) (print 1)]
   [list (lreturn y) (print 2)])
```

Here is an example of its use:

```
ALLOY > (list #*(ints2p 1 2))
012==> (1 2)
```

```
ALLOY > (set c ('new coex ^!*(ints2p 1 2)))
==> %(object of class COEX)
ALLOY > ('next c)
01==> 1
ALLOY > ('next c)
2==> 2
ALLOY > ('next c)
==> %f
```

It is interesting to note the use of lazy evaluation in combination with generators, as well as the control of a generator through the use of the `return` statements.

5.7 Comparison with other Programming Languages

This section compares ALLOY's inter-process communication facilities with those of other parallel object oriented programming languages.

5.7.1 Actors

Actors [Agh86b,AH87] is a model of computation based on objects and message passing. It has been suggested that Actors could be viewed as an assembly language for Distributed Architectures. A disadvantage is that there seems to be no way for Actors to exploit the full power of shared memory multiprocessors.

The low level of Actors can be seen in the following implementation of factorial:

```
(define (Factorial())
  (Is-Communication (a doit (with customer = m) (with number = n)) do
    (become Factorial)
    (if (= n 0)
        (then (send m 1))
        (else (let (x = (new FactCust (with customer m) (with number n)))
                  (send Factorial
                        (a doit (with customer x) (with number n-1))))))))

(define (FactCust (with customer = m) (with number = n))
  (Is-Communication (a number k) do
        (send m n*k)))
```

The same program is much simpler in ALLOY:

```
(setfun fact(n)
   (return (if (gt n 0) (times n (fact (diff n 1))) 1)))
```

The complexity of such a simple program in Actors suggests that Actors be used as the implementation language of a higher level language. One such attempt has been the Act3

language. Act3 facilitates writing in a functional style, though it does nothing about and-parallelism, lazy evaluation, or generators.

Actors has been promoted for its ability to create abstract data types in a parallel system. Here is the implementation of a stack:

```
(define (new stack-node (with context = c) (with next-node = next))
  (Is-Communication (a pop (with customer = m)) do
    (if (NOT (= c empty-stack))
        (then (become forwarder (next))
              (send-to (m) (a popped-top (with value c))))))
  (Is-Communication (a push (with new-content = v)) do
    (let (x (new stack-node (with content c) (with next-node next)))
      (become (stack-node (with content v) (with next-node x))))))
```

Notice that if the actor does not specify a replacement actor (using become) then by default an identically behaving actor is used to get the next message from the mailbox. An Actor that is no longer accessible may be removed by garbage collection. The implementation of a similar eager-stack in ALLOY was given earlier in this chapter. Here it is shown how to define a more powerful blocking stack:

```
(provide "stack")
(require "semaq")
(require "estack")

(class-start stack
   (static  non-empty es)
   (import  semaphore eager-stack)
   (methods push pop))

(setfun new()
   (set non-empty ('new semaphore))
   (set es ('new eager-stack)))

(setfun push(x)
   (return [block1 ('push es x)    ('v non-empty)]))

(setfun pop()
   (return [blockn ('p non-empty)  ('pop es)]))

(class-end stack)
```

ALLOY has the extra advantage of being able to make any number of `push` operations in parallel (provided enough processors are available). Implementing a blocking stack like this one in Actors would not be very easy since fine grain synchronization (here serial evaluation) is rather complicated.

5.7.2 Vulcan

Vulcan [KTMB86,KTMB86] is a significant step in the evolution of parallel logic programming languages. By adding object oriented programming to Concurrent Prolog it brings powerful abstraction mechanisms to it. However, since it is based on logic programming, it shares some of their problems: too many side effects (because of the logical variable), no or-parallelism in a parallel environment, and unnecessary serialization of messages.

Vulcan, being just a pre-processor, provides objects with the same restrictions as those of the base language. Messages can not be implemented efficiently and streams of messages are too apparent all the time. When many processes need to send messages to the same object, they must be given separate streams to put their messages on, and the merging has to be explicit. Having to send all messages through streams, often imposes undesirable synchronization. Worse, no messages can be processed unless the previous one has been identified.

Consider how a queue object would be written in Vulcan [KTMB86]:

```
class(queue, [Head, Tail]).

queue :: create(Queue) :- queue : make([Head, Head], Queue).
queue :: enqueue(Item) :- Tail = [Item | (new Tail)].
queue :: dequeue(Item) :- Head = [Item | (new Head)].
```

The interesting property of this queue is that is can be used to dequeue elements in advance (before they were enqueued). An instance of a queue can be created and used as in this goal:

```
?- queue:create(Q), send(Q, enqueue(1), Q2), send(Q2, dequeue(X), Q3).
```

Notice that while CP is inherently parallel it does not make any difference if the dequeuing takes place before or after the enqueueing. X will finally be bound to 1. The way this queue is defined is somewhat unusual due to the properties of the logical variable. Every time an message is sent to the object, its *current* variables are duplicated into the *new* set of variables. A variable that is *new* for one message is *current* for the immediately succeeding message to that object. It is the responsibility of the methods to form a consistent state for the object, which is visible from the next messages.

ALLOY's dynamic synchronizing variables have been inspired by VULCAN. In ALLOY, however, parallelism during execution of messages has been increased considerably,

and the use of synchronization values (with their limited scope), instead of the logical values, reduces side-effects and is more efficient to implement (especially in a distributed system), because of their directionality. When logical values are needed in ALLOY they can be implemented as described in chapter 7. With the exception of object oriented programming, ALLOY has the same advantages over Vulcan as over other logic programming languages.

5.7.3 Linda

Linda [CG88] is not a programming language, nor a complete programming model. It is a model for process creation and coordination that is orthogonal to the base language in which it is embedded. Linda provides a mechanism called Tuple Space (TS) for creating and coordinating multiple execution threads in any programming language.

Tuple Space (TS) is a global pool of tuples. Tuples are sequences of typed fields, such as `("new stuff", 0, 16.01)`. Tuples can be matched to *templates*, where templates may contain variables (atoms starting with '?'), in which case the variables are set to the corresponding values of the tuple.

Processes can be created to place a tuple in TS, and processes can be suspended waiting for a tuple in TS. One difference between this and logical variables is that logical variables are local, not global[5]. This extra freedom can create security problems and also make implementation difficult since the search is complicated. An advantage is that both multiple producers and multiple consumers can be handled quite easily. In order to introduce some abstraction, there is ongoing research trying to split the Tuple Space into sub-spaces. Each of these sub-spaces will be accessible only to some of the processes.

Here is an implementation of the n philosophers problem in Linda. A special algorithm is used to guarantee deadlock-free operation [CG89]:

```
void phil(int i)
{ while(1) { think();
             in("room ticket"); in("chopstick", i);
             in("chopstick", (i+1)%Num);
             eat();
             out("chopstick", i); out("chopstick", (i+1)%Num);
             out("room ticket");}
```

[5] This can be simulated in Linda by placing a unique object (address) in the tuple, and require that anybody that needs to access the tuple, should specify that object.

```
}

initialize(int i)
Num = i;
{ for(i=0; i<Num; i++) { out("chopstick", i); eval(phil(i));
                         if(i>0(Num-1)) out("room ticket");}
}
```

The special algorithm used to guarantee there are no deadlocks is to ensure that one philosopher is always kept out of the room. This solution does not guarantee there will be no starvation (i.e. simple semaphores implemented in Linda are not fair, probably for efficiency reasons).

It has been shown previously in this chapter that with appropriate use of abstract data types, one can give a clean solution to the above problem without using any special algorithm to avoid deadlocks. However, tuple spaces are an interesting combination of dictionaries and queues. In, ALLOY one can define a class of tuple spaces as follows:

```
(provide "tuples")
(require "queue")

(class-start tuples
   (import dictionary var-p queue)
   (common top-level-caller)
   (static dict)
   (methods out eval in rd))

(setfun new()
   (set dict ('new dictionary)))

(setfun out(key info)
   (return (let [(x ('get dict key))]
              (when (var-p x) ('put dict key ('new queue)))
              ('enqueue x info))))

(setfun eval(key iclosure)
   (return (let ((info ('top-level-funcall c-top-level-caller iclosure)))
              (out key info))))

(setfun rd(key)
   (return ('top ('get dict key))))

(setfun in(key)
   (return ('dequeue ('get dict key))))

(class-end tuples)
```

In the case of the eval message, information must be passed as a closure. This solution

has the extra ability to be able to create many tuple spaces, thus improving abstraction. Also, tuples are taken in a first in first out order.

As an example of the use of class `tuples` following is a Linda-style solution of the n philosophers problem:

```
[setfun phil(i num ts)
    (think)
    ('in ts 'room-ticket) ('in ts (cons 'chopstick i))
    ('in ts (cons 'chopstick (quotient (sum i 1) num)))
    (eat)
    ('out ts (cons 'chopstick i))
    ('out ts (cons 'chopstick (quotient (sum i 1))))
    ('out ts 'room-ticket)
    (phil i num ts)]

(setfun initialize(i)
    (let ((ts ('new tuples)) (num i))
        (createph (diff i 1) num ts)))

(setfun createph(i num ts)
    ('out ts (cons 'chopstick i)) ('eval ts (phil i num ts))
    (when (gt i 0) ('out ts 'room-ticket) (createph (diff i 1) num ts)))
```

This solution uses a local tuple space and is therefore safe from other processes also using, for example, the name `'room-ticket`. For the same reason this solution allows many groups of philosophers to have dinner in separate rooms at the same time. It is also starvation free.

On the other hand, this solution finds a tuple through pattern matching of fixed keys. Linda performs a more powerful one way unification. If full unification is needed, the above class of `tuples` can be changed to reflect this.

5.8 Summary

In ALLOY, every value is an object. Thus, classes, messages, functions, numbers etc. are all objects. It has been shown that ALLOY can create highly parallel abstract data types using object oriented programming. In ALLOY, objects can process messages in parallel. Dynamic synchronizing variables make synchronization among messages transparent.

ALLOY supports basic interprocess communication through the use of synchronizing values. More advanced communication is supported through the use of specialized parallel objects. A comparison with a number of parallel object oriented languages indicates

that ALLOY is clean and efficient not only for simple but also for complex interprocess communication.

Chapter 6

Generators and the Replicator Control Structure*

The need for searching a space of solutions appears often. Many problems, such as iteration over a dynamically created domain, can be expressed most naturally using a generate-and-process style. Serial programming languages typically support solutions of these problems by providing some form of generators or backtracking.

A parallel programming language is more demanding since it needs to be able to express parallel generation and processing of elements. Failure driven computation is no longer sufficient and neither is multiple-assignment to generated values.

We describe the *replicator* control operator used in the high level parallel programming language ALLOY. The replicator control operator provides a new view of generators which deals with these problems.

6.1 Introduction

This chapter [MH90] describes the mechanisms which the language ALLOY provides for dealing with non-determinism and generators in a parallel programming language. Finally, it compares these with analogous features in other programming languages.

Non-determinism is provided in ALLOY to permit the expression of indifference or uncertainty in an algorithm [Dij75,CG83,Sha86a,Ued86]. Generators are included to

*This chapter is essentially a reprint of [MH90].

ease the development of programs which make use of the generate-and-test technique [GG83,SS86,Hen80].

In a parallel environment the natural view of generators requires them to be able to produce their results in parallel. This implies that a failure driven model is not adequate, since failure is inherently serial. Also, since ALLOY uses single-assignment variables to synchronize parallel computations, it is not clear how the (potentially) many values returned by a generator should be handled. Furthermore, since many generators may be operating in parallel, more powerful techniques are required for controlling the generating procedure and combining the results of generators. ALLOY introduces the *replicator* control structure to provide a new view of generators which deals with these problems.

6.1.1 Generate-and-test Problems

Iterators, generators, co-expressions, backtracking and or-parallelism, all provide ways to express algorithms that need to search a space of possible solutions.

CLU [LSAS77] introduced the notion of iterators. Iterators provide the ability to perform the same action over a collection of items which is represented by a function (*iterator*). The iterator produces the items in the collection, one at a time as they are requested for examination from a `for` control loop. Iterators are functions which can maintain their data and control state after they return a value.

ICON [GG83] provides limited control backtracking, generators, co-expressions, and coroutines. Generators are like iterators but can be invoked anywhere in a program. Generators return all their values at the point (syntactically) of their invocation. They can be driven explicitly by demand (a new value is requested) or implicitly by backtracking on failure (processing of the last generated item failed). Generators are more powerful than closures in that a generator maintains data/control state (e.g. values can be returned from the middle of a loop) whereas closures only maintain an internal data state. However, closures are first-class objects since they can be passed around in a program and get activated at any point while their internal data state is preserved. In contrast, pure generators generate all values at the point of their invocation as needed and then lose both data and control state.

Co-expressions [GG83] are essentially coroutines which can return values. They combine the capabilities of generators and closures. A co-expression is a generator transformed

into a first-class object so that it can generate individual items in different places in a program. The life-time of a co-expression can not in general be determined at compile time and so requires more complicated runtime support (than that needed for a generator). Co-expressions and coroutines can be emulated using (Scheme-like) continuations. Where co-expressions are emulated using just closures, it is necessary for the programmer to explicitly store and restore the control state of the closure.

Backtracking operates at another level and allows the expression of algorithms of the form "assume this is the correct choice and proceed" and any time later "if the choice is not correct forget all computation since last choice and continue from that point on". Control backtracking undoes only the decision made at the last backtracking point, while full backtracking also undoes any changes to data structures. While full backtracking is very powerful, it is often inefficient [SDDS86], as are full continuations. In Prolog, a language where side effects are severely restricted[1], backtracking is efficient. MU-Prolog [Nai86] adds coroutining.

Scheme [RG86] provides a limited form of continuations (call-cc), which directly provide control (but not full) backtracking, and can emulate generators and coroutines [HFW84]. For efficiency reasons, continuations in Scheme are limited to providing only continuation of control.

In a parallel programming language it is natural to replace backtracking, the serial generation of possible solutions, by mechanisms which refer to the set of possible solutions, leaving unspecified the order in which these solutions are generated. In logic languages this corresponds to full or-parallel execution [War87]. Recent work on Aurora [LBO+88], an or-parallel Prolog language, suggests that complete or-parallelism can be implemented efficiently.

Supporting full or-parallelism in an and-parallel parallel environment seems to be not only inefficient, but also exceedingly hard to implement [Mit88b,CS87]. As a result, parallel programming languages provide only partial solutions [Gre87,Sha86a,Mit89b]. In some cases full solutions are given to weaker subsets of a language [Ued87a,Ued87b,CG85]. These subsets allow little or no and-parallelism. Other experimental languages attempt to accommodate restricted and-parallelism [HB88] while others replace and-parallelism with

[1] Side effects are only possible by dynamically changing a program.

coroutining [Nai88].

Ultra Prolog [Mit88a], PARLOG [CG83], and GHC [Ued86], are examples of and-parallel logic programming languages, with limited or-parallelism (no multiple environments) [Sha89]. Concurrent Prolog [Sha86a] does provide full or-parallelism, but was abandoned (in favor of FCP [SHHS87]) as unacceptably inefficient; no implementation of CP on a parallel computer exists. As with Ultra Prolog [Mit88a] there is a lot of research on the identification of language subsets which can have full or-parallelism.

Multilisp [Hal84] and Queue-Based Multiprocessing Lisp [GM84], two parallel functional programming languages, make no attempt to support backtracking or even generators. It is in general hard to implement generate and test problems in these systems (harder in Multilisp). Controlling the generation process is even harder.

ALLOY proposes a particularly flexible form of generators to deal with these issues. First, we briefly describe those features of the language which are necessary to understand the mechanism, followed by a description of the *replicator* operator.

6.2 Definitions

6.2.1 First Look at Generators

As described above, functions in ALLOY can return one of many results, possibly non-deterministically. These functions become generators when they are called prefixed by the operator * (star). In effect, a function becomes a generator by removing the ability of its return statements to commit. An expression that is prefixed by the replicator operator # (hash) is a *replicable* expression.

Broadly speaking, the idea is that a replicable expression is replicated once for each combination of its generators. A generator is called once for each combination of its generators.

The simple case of a single generator inside a replicator:

```
#(p ... *e ...)
```

is evaluated as though textually replaced by

```
(p ... e1 ...) (p ... e2 ...) ... (p ... en ...)
```

where the expressions e1, e2, ... en represent the set of values which would have been generated by *e.

In a later section a function `(ints start stop)` will be defined which, when called as a generator, can produce all integers in the range `start ... stop` inclusively. Thus, the following two calls are equivalent:

```
(let() #(print w *(ints 1 5)))

(let() (print w 1) (print w 2) (print w 3)
       (print w 4) (print w 5))
```

Both of these will print a permutation of the numbers `1 2 3 4 5`.

Nested generators provide a form of backtracking. In a context of serial calls this is equivalent to depth first search, while in a context of parallel calls it is equivalent to (parallel) breadth first search. Thus, the next two forms are equivalent:

```
(let() (+ #*(ints 1 *(ints 1 3))))

(let() (+ #*(ints 1 1) #*(ints 1 2) #*(ints 1 3)))
```

and both will return the number `10`.

Other examples of uses of replicators will follow the formal definitions of generators and the replicator.

6.2.2 Generators in a Replicator

The definitions in this section will be given in terms of the `factory` object[2]. This is a library object (implemented in ALLOY) which in effect creates co-expressions. When an instance of a `factory` object is made, it is given a generator in a closure. By sending appropriate messages to the object we receive the values of the generator.

Some of the operations on the above object will be explained next. A generator can be transformed into a factory as in:

```
(set f (factory ^*(ints-inc 1 5)))
```

The `factory` class can be defined in terms of classes `coex` (chapter 5) and `vector` (chapter 3).

Just as `'e` is equivalent to `(quote e)` in lisp, `^e` is equivalent to `(mu #(lreturn e))`. The above expression[3] evaluates new members, or simply provides (previously evaluated) specific values with a call like:

[2] In practice however, this is not immediately possible since the `factory` object uses replicators and generators itself. This object should be implemented at a lower level.

[3] Note that `mu` is the function creation function like `lambda` is for Lisp.

```
('nth f 3)
```

The above expression will return the 3^{rd} value generated by that factory. If that value has been evaluated it is simply returned otherwise it is evaluated at that time. If there is no such value, the call fails.

A generator must be defined in the scope (static) of a replicator or of another generator. A generator a is in the scope of a replicator b if a is a parameter (direct or indirect) of the replicator, is evaluated in the same scope with it, and is not in the scope of another generator or replicator which is in the scope of b. A *replicable* expression is replicated in such a way that its enclosed generators are replaced by their values. The program fragment:

$(^{c}\cdots$ `#(`$^{r}\cdots *e_1 \cdots *e_i \cdots *e_n \cdots)^{r} \cdots)^{c}$

is transformed to:

$(^{c}$`let ( (`f_1 `('new factory ^*`e_1`))`
`(`f_i `('new factory ^*`e_i`))`
$\vdots \qquad\qquad \vdots$
`(`f_n `('new factory ^*`e_n`))`
`(rep (mu (`$p_1 \cdots p_i \cdots p_n$`)`
`(lreturn (`$^{r}\cdots p_1 \cdots p_i \cdots p_n \cdots)^{r}$`))))`

$(^{c}\cdots$ `(`^{r}`rep ('nth` f_1 `1)` $\cdots$ `('nth` f_i `1)` $\cdots$ `('nth` f_n `1) )`^{r}
`(`^{r}`rep ('nth` f_1 `1)` $\cdots$ `('nth` f_i `1)` $\cdots$ `('nth` f_n `2) )`^{r}
`(`^{r}`rep ('nth` f_1 `1)` $\cdots$ `('nth` f_i `1)` $\cdots$ `('nth` f_n m_n`))`^{r}
$\vdots \qquad\qquad \vdots \qquad\qquad \vdots$
`(`^{r}`rep ('nth` f_1 `1)` $\cdots$ `('nth` f_i `2)` $\cdots$ `('nth` f_n `1) )`^{r}
`(`^{r}`rep ('nth` f_1 `1)` $\cdots$ `('nth` f_i `2)` $\cdots$ `('nth` f_n `2) )`^{r}
`(`^{r}`rep ('nth` f_1 `1)` $\cdots$ `('nth` f_i `2)` $\cdots$ `('nth` f_n m_n`))`^{r}
$\vdots \qquad\qquad \vdots \qquad\qquad \vdots$
`(`^{r}`rep ('nth` f_1 `1)` $\cdots$ `('nth` f_i m_i`)` $\cdots$ `('nth` f_n m_n`))`^{r}
$\vdots \qquad\qquad \vdots \qquad\qquad \vdots$
`(`^{r}`rep ('nth` f_1 m_1`)` $\cdots$ `('nth` f_i m_i`)` $\cdots$ `('nth` f_n m_n`))`^{r}
$\cdots)^{c}$

where generators $e_1 \cdots e_n$ are those in the scope of the replicator and must all execute either concurrently or serially as specified by parentheses '^{r}'. In cases where the calls to

these generators have to be made by following a more complex synchronization pattern, in the calls to `rep` the generators must be grouped[4] accordingly. Of course, function `rep` must first take from these grouped values the values themselves.

The above transformation first creates a factory for each generator and then requests and accesses the values of that generator by sending messages to the corresponding factory object. The replications are done dynamically depending on the number of solutions m_i for each generator f_i. For example, if at least one generator f_i fails to return any result at all, the replicable expression affecting it disappears[5]. Note that in a parallel call, the above order of combinations may not be retained.

Some times a replicated expression is only important for its side effects (i.e. return statement, changing state of objects). In cases like these, the result of the replicator is not needed and this can be declared by the use of the null replicator `#!` in place of the regular replicator `#`. This may improve not only the efficiency of the program but also its complexity (see generator of integers bellow).

6.2.3 Nested Generators

A generator a is in the scope of generator b if a is a parameter (direct or indirect) of b, is evaluated in the same scope with it, and is not in the scope of another generator or a replicator which is in the scope of b. A generator which affects other generators is treated in a way similar to the way a replicator would be treated except that now the replication process is transparent and all the values that are generated appear to have been generated by the enclosing generator. The outermost generator in the program:

$$(^{c}\cdots \quad *(^{r}\cdots \quad *e_1 \ \cdots \ *e_i \ \cdots \ *e_n \ \cdots)^{r} \ \cdots)^{c}$$

is transformed to:

$$*(^{c}\texttt{let} \ \ (\ (f_1 \ (\texttt{'new factory \^{}}*e_1))$$
$$(f_i \ (\texttt{'new factory \^{}}*e_i))$$
$$\vdots \qquad\qquad \vdots$$

[4] One way to do this is to place generators as arguments of a call to function `list`. That call should then be concurrent or serial as needed.

[5] This helps to define functions simulating conditional expressions such as `when`. However, these functions would then have to take closures as arguments.

```
        (f_n ('new factory ^*e_n)) )
        (rep(^c mu (p_1 ··· p_i ··· p_n)
                #(lreturn *(^r ··· p_1 ··· p_i ··· p_n ···)^r))^c))
#(lreturn *(^r rep ('nth f_1 1)   ··· ('nth f_n 1)  )^r)
#(lreturn *(^r rep ('nth f_1 1)   ··· ('nth f_n 2)  )^r)
#(lreturn *(^r rep ('nth f_1 1)   ··· ('nth f_n m_n))^r)
   ⋮              ⋮                    ⋮
#(lreturn *(^r rep ('nth f_1 2)   ··· ('nth f_n 1)  )^r)
#(lreturn *(^r rep ('nth f_1 2)   ··· ('nth f_n 2)  )^r)
#(lreturn *(^r rep ('nth f_1 2)   ··· ('nth f_n m_n))^r)
   ⋮              ⋮                    ⋮
#(lreturn *(^r rep ('nth f_1 m_1) ··· ('nth f_n m_n))^r))^c
```

where the internal generators are those in the scope of the external one and the calls to `rep` (as well as `rep` itself) are modified, if needed, as described in the previous section.

The replications are done dynamically depending on the number of solutions m_i for each generator f_i. Note that in a parallel call, the above order of combinations may not be retained. The enclosing generator would not be called at all (as if it failed) if at least one of its generators fails to produce any result at all.

A return statement in a generator is not considered to be complete (have finished execution) until a new value is needed from that generator. This is clarified in the implementation of co-expressions (chapter 5). This of course is only useful in combination with serial execution in the body of the generator and the environment of the replicator which uses it. This mechanism can provide control backtracking.

6.2.4 Values are Generators

One more rule is needed to complete the definition of generators: Values are generators of themselves. Thus, both of these examples:

```
*5

(let ((x 5)) *x)
```

evaluate to the number `5`. The most interesting part about this rule is its exception: The generator of *failure* value generates no values at all. This allows ALLOY to define the

`when` expression. For example the following two calls are equivalent:

```
(when condition expression)

((mu (x) #(lreturn (cdr (cons *x expression)))) condition)
```

Calling a function prefixed with two (or more) * operators results in calling the function as a generator but filtering out *failure* values.

6.3 Examples

6.3.1 Two Functions/Generators

Using * to call functions as generators and # to replicate expressions one can define more powerful functions. The following function `ints` when called normally returns a value in the range `start ... stop` while as a generator returns all these numbers in a random order.

```
(setfun ints(start stop)
   (when (le start stop)
        (return start)
      #!(return *(ints (plus start 1) stop))))
```

When called normally, this function can either execute the first return statement first or attempt to replicate the replicable expression if the recursive call to `ints` succeeds. In fact, it may do both at the same time since the expressions can be evaluated in parallel.

If `(= start stop)` then the replicable expression will disappear, in which case only the first return statement can execute.

The serial version of this, written as:

```
(setfun ints-inc(start stop)
   [when (le start stop)
        (return start)
      #!(return *(ints-inc (plus start 1) stop))])
```

returns number `start` if `start` is not greater than `stop`. Note how the two return statements that form the body of `when` are executed serially, Of course, as soon as a return statement (always the first one here) is executed, function `ints-inc` commits. When called as a generator, the above function will return all values in order.

Note the use of the null generator `#!` here. It is the reason why the above two functions have computational requirements linear in the number of times they succeed. If the normal generator was used instead, the computational requirements would be quadratic in the number of times they succeed.

6.3.2 More Examples

The formal definition of a replicator at first sight looks complex. Nevertheless, after some experience with them their use becomes quite simple. The following examples will give an indication of that simplicity.

```
(print w (list #*(ints 1 5)))
```

will print some permutation of the list `(1 2 3 4 5)`, while

```
(print w (+ #*(ints 1 5)))
```

will print `15`.

Generators are controlled by using serial or parallel invocations. In the next example the values are produced by the generator and used serially, in a way which is reminiscent of backtracking. The following two calls are equivalent:

```
[let() #(print w *(ints-inc 1 5))]

[let() (print w 1) (print w 2) (print w 3)
       (print w 4) (print w 5)]
```

These will print the numbers `1 2 3 4 5` in that order. A number is not evaluated until the previous print has terminated.

A number of non-nested generators inside a replicator are defined to generate the direct product, so the following two calls are equivalent:

```
(let() #(print w (cons *(ints 1 4) *(ints 5 8))))

(let() #(let ((v1 *(ints 1 4))
              (v2 *(ints 5 8)))
          (print w (cons v1 v2))))
```

Both will print in window w some permutation of the sequence: `(1 . 5) (1 . 6) ... (1 . 8) (2 . 5) (2 . 6) ... (2 . 8) ... (4 . 8)`.

Nested generators are affected only by the enclosing generator, so the two following calls have the same behavior:

```
(let() #(print w *(ints 1 *(ints 1 5))))

(let() #(let ((v1 *(ints 1 5)))
           #(let ((v2 *(ints 1 v1)))
               (print w v2))))
```

Both will print in window `w` some permutation of the sequence: `1 1 2 1 2 3 1 2 3 4 1 2 3 4 5`.

Finally, this more complicated example:

```
(let() #(print w (cons *(ints 1 *(ints 1 5))
                      *(ints 6 *(ints 6 8)))))
```

will print $(\sum_{i=1}^{5} i) * (\sum_{i=6}^{8} i - 5) = 15 * 6 = 90$ pairs of numbers. These will be some permutation of the sequence: `(1 . 6) ... (1 . 8) (1 . 6) ... (1 . 8) (2 . 6) ... (2 . 8) (1 . 6) ... (3 . 8) ... (5 . 8)`.

6.3.3 Computing and Processing the Values of a Generator in Parallel

The following function produces either one or all permutations of a list in parallel (depending on whether it is called simply or as a generator):

```
(setfun permutations(l)
  (when (null-p l) (return nil))
  (when (cons-p l)
     #(lets (((x rest) *(delete-one l)))
               #(return (cons x *(permutations rest)))))))

(setfun delete-one(l)
  (when (cons-p l)
      (return (list (car l) (cdr l)))
     #(return (let (((x rest) *(delete-one (cdr l))))
                 (list x (cons (car l) rest))))))
```

The `let` call in the `permutation` function is called once for each value returned by generator `*(delete-one l)`, since it is the immediately enclosing generator. Furthermore, since `let` is called as a generator, it will return all results of the generator in its body.

The example which follows presents an entirely parallel algorithm that provides one or all solutions to the 8-queens problem.

```
(setfun queens(n)
   (lets ((lst (make-list n)))
      #(lets ((bd (board lst *(permutation lst))))
          (unless (unsafeall bd) (return bd)))))
```

```
(setfun unsafeall(board)
  (when (cons-p board)
    (when (unsafe (car board) (cdr board)) (return))
    (when (unsafeall (cdr board))          (return))))

(setfun unsafe(q board)
  (when (cons-p board)
      (when (unsafe1 q (car board)) (return))
      (when (unsafe  q (cdr board)) (return))))

(setfun abs(x)
  (return (if (lt x 0) (diff x) x)))

(setfun unsafe1(q p)
  (when (eq (abs (diff (car q) (car p)))
            (abs (diff (cdr q) (cdr p)))) (return)))

(setfun board(lst places)
  (return (if (null-p lst) ()
            (cons (cons (car lst) (car places))
                  (board (cdr lst) (cdr places))))))

(setfun make-list(n)
  (if (lt n 1) (return ())
    (return (cons n (make-list (diff n 1))))))
```

Here, all possible boards are generated in parallel and at the same time tested if they are acceptable.

It is possible to replace function `permutation`, above, by another which can utilize more processors:

```
(setfun permutation(l)
   #(return *(perms l (length l))))

(setfun perms(l n)
   (when (null-p l) (return nil))
   (when (cons-p l)
      #(lets (((x rest) *(delete-one l n)))
           #(return (cons x *(perms rest (diff n 1)))))))

(setfun delete-one(l n)
   #(return (delete-nth *(all-ints 1 n) l)))

(setfun all-ints(x y)
   (when (eq x y) (return x))
   (when (lt x y)
      (lets ((middle (quotient (sum x y) 2)))
         #(return *(all-ints x middle))
```

```
          #(return *(all-ints (sum middle 1) y)))))

(setfun delete-nth(n l)
   (return (let (((bc val) ('new become)))
             (list val (delete-nthx n l bc)))))

(setfun delete-nthx(n l bc)
  (when (cons-p l)
     [when (le n 1)
        ('make bc (car l)) (return (cdr l))]
     (when (gt n 1)
        (return (cons (car l)
                      (delete-nthx (diff n 1) (cdr l)
                                   bc))))))
```

We have a sketch of a preliminary implementation on a shared memory multiprocessor, which suggests that the first program for 8 queens finds all solutions in $O(\max(\frac{n!}{\text{cpus}}, n^2))$ time, while the second (the one which uses the last definition of `permutation`) in $O(\max(\frac{n!}{\text{cpus}}, n \log n))$ time. The number of available processing elements is cpus.

Even though the last program achieves higher processor utilization it also has considerably greater overhead. This is because function `delete-one` in the last program requires quadratic processing power, while in the first program only linear processing power.

Note that function `delete-nth` makes use of the "value to become" object (`become`) to take the value deleted from the input list in a more convenient way.

6.3.4 Computing the Values of a Generator using Control Backtracking

A generator that makes use of serial execution can be controlled by the replicator and be made to compute and produce values on demand. This is equivalent to control backtracking in the generator and is achieved by making some serial[6] calls in appropriate places.

For example, in order to solve the above problem of the 8 queens using control backtracking we need a different generator for the permutations and a new driver:

```
(setfun permutation(l)
  (when (null-p l) (return nil))
  [when (cons-p l)
```

[6] Calls whose parameters are evaluated serially. These are declared by placing the call in square brackets.

```
        #[lets (((x rest) *(delete-one l)))
            #(return (cons x *(permutation rest))])])

(setfun delete-one(l)
  [when (cons-p l)
      (return (car l) (cdr l))
     #(return (let (((x rest) *(delete-one (cdr l))))
                 (list x (cons (car l) rest))))])

(setfun queens(n)
   [let ((lst (make-list n)))
      #(lets ((bd (board lst *(permutation lst))))
          (when (not (unsafeall lst bd))
             (return bd)))])
```

Here, the possible positions of the queens on the board are generated and tested one after the other. For this reason, it would be possible to replace replicable single assignment variables with non-replicable multiple assignment ones. Note that a position is being tested while it is generated. To test a position only after it is completely generated, the last call to the `let` expression should be serial.

6.4 Implementation

6.4.1 Transformations and complications

The two transformations presented earlier can be used for the implementation of these structures. This fact is a proof that generators and the replicator in ALLOY have no undesirable effects on the efficiency of the rest of ALLOY.

Part of the transformations presented must be done at run time. That is the part which depends on the number of results returned by the various factories. For this reason and for efficiency, the transformation, taking place during pre-processing, would make use of some low level system functions. This is also the way these structures are implemented by the interpreter in Appendix A. The most important complications have to do with preserving the types of evaluation (i.e serial/parallel, eager/lazy) and the controllability of the `return` statements.

6.4.2 Parallel Architectures

The replicator is particularly important in parallel systems. This is due to two reasons. First and foremost, it allows the expression of highly parallel algorithms. Second, it often provides parallelism of more coarse grain in the form of generators. The first point is further explained in the following.

A high degree of parallel replication can appear in two forms. First, a generator in a replicable expression can return many results concurrently through processes it creates. Second, multiple non-nested generators in a replicable expression will have that expression replicated for a number of times equal to the product of the number of results from each generator.

In a shared memory MIMD architecture supporting a highly parallel primitive like `fetch-and-add`, replications can take place in parallel. For this to happen it is sufficient to delegate responsibility of the replication to the generators driving it.

In particular, processes in a generator directly place each result in an array, assigned to that generator, managed by a highly parallel index. Many results can be placed in this array concurrently. As soon as a generator co-driving a replicable expression realizes that replication can take place (all driving generators have generated at least one result,) it creates a *multiple* process to perform the replication or many replications concurrently. A *multiple* process is equivalent to an arbitrary number of processes and is created in constant time [FG91] The cpu schedulers will behave as if many processes have been placed in the parallel waiting queue.

A *multiple* process needs the `fetch-and-add` primitive for its realization. Each of its instances will access the arrays of values (created by the generators) using highly parallel indices. Each instance process of the *multiple* process will finally create and evaluate a replica statement concurrently with all of the other instance processes.

The array assigned to each generator should be dynamic in size. This is possible by implementing it as a two dimensional array whose one dimension contains only pointers to dynamically allocated vectors. In the beginning only the first pointer is allocated a vector. As soon as the vector pointed to by a pointer gets filled the next pointer can be made to point to a new empty vector.

6.5 Discussion

6.5.1 Comparison with other Languages

Icon provides control backtracking between a function and its parameters as well as between the parameters of a function. Making serial calls in ALLOY and nesting generators results precisely in ICON-style programs. However, in ICON a function can be either a simple function or a generator (but not both) depending on its definition. In ALLOY this depends on the way a function is called.

Prolog always provides full backtracking, unless explicitly requested not to do so (assuming no side effects take place). If side effects do take place the program has to restore their effects explicitly, as with control backtracking. Having backtracking at every point makes expressing of many *conventional* programs awkward.

Since replicators in ALLOY provide control backtracking, it is not hard to translate to ALLOY (see chapter 7) a program written in ICON or PROLOG. The transformation whether mechanical or by hand would be simple. However in the case of translating PROLOG to ALLOY the resulting program may be somewhat longer than the original one[7]. It should be noted that functional programs have much less need for data backtracking than logic programming languages, since functions can succeed repeatedly and return new solutions without having to re-assign variables to new values.

SETL provides a different solution by allowing one to define a backtracking point inside a function and return there from anywhere when a failure is registered. This means that any function is a potential backtracking point if it either has a backtracking point or makes a call to a function that may fail. Since variables whose values must be restored during such a backtracking need to be declared, the above mechanism is not more expressive than that of, say, ICON.

The continuation mechanism of Scheme can provide coroutining and control backtracking, and is more powerful than that provided by ICON, and also PROLOG and SETL (except for data backtracking), since it can express and control many threads of execution some of which may be backtracking points of others. ALLOY makes no attempt to provide continuations, which we view as almost too powerful for general use, but provides

[7]This is true when the parameters of a predicate are used both for input and output. In such a case logical variables/unification (objects) will be needed.

mechanisms which are equally expressive for most common applications of continuations, and more expressive for others. The one application of continuations which seems difficult to implement in ALLOY is that where continuations split threads of execution without prior preparation.

And-parallel logic programming languages such as PARLOG, GHC, CP etc. provide limited or-parallelism or full or-parallelism on some subset of the language. These multiple solutions must then be explicitly iterated upon. A serious disadvantage is that the mentioned limited subset is not allowed to interact or communicate with the full language except to provide its solutions. One can directly program in PARLOG style in ALLOY with the only support being a class which provides logical variables and messages for their unification.

Later research attempts to combine properties of and-parallel LPLs with or-parallel Prolog languages. Andorra Prolog [HB88], P-Prolog [YA86] and Pandora [BG89] all try to provide some form of both and-parallelism and or-parallelism with suspension and commitment mechanisms for parallel programming. These designs seem to be promising, but it remains to be seen how powerful/efficient they are.

Multilisp and Queue-Based Multiprocessing Lisp, two of the few functional programming languages which can express arbitrary parallel algorithms, make no attempt to ease implementation of programs that need to search a space of solutions. Even with the support of a powerful library, solutions to the above problems would be far from being elegant or convenient to read or write. Users would have to resort to the use of special-purpose user-defined macros to express their problems in a natural way.

6.5.2 Discussion

ALLOY employs parallel execution of calls and commitment of closures. What other languages regard as and-parallelism here is equivalent to letting processes finish execution before committing a closure. What other languages regard as or-parallelism here is equivalent to committing a closure before some of its processes have finished execution. Or-parallelism typically appears in the form of a set of `when/return` pairs.

Having functions instead of predicates makes flow of data clearer as well as reducing the need for multiple environments during or-parallel execution. The replicator creates new environments where necessary by replicating instances of function calls.

Replicas of replicable expressions do not have to be made serially or to be made by one process only. Instead, replicas can be created and executed by the generators in the replicable expression. Depending on the hardware this may involve little or no synchronization. This allows the programmer to express highly parallel algorithms and the implementation to exploit various types of hardware.

Combining results of generators is considered to be most common in a parallel environment and this is indeed what one achieves with non-nested generators. Still, calling a generator once for each of the results of another generator is also possible by nesting the latter into the former. This last ability is what makes (both or-parallel and serial) Prolog-like programs simple to write.

Collecting all solutions from a generator is possible by using a convenient container object. However, in most cases the explicit construction of this object is unnecessary as the solutions may be used directly.

Non-deterministic commitment is possible when a function is not called as a generator. It is trivial to write a function that merges two streams. A similar function can be used as a generator to produce all possible resulting streams, even though that would be of little use.

We believe that the notions of replicators and generators can be made part of most functional programming languages, with only minor effort.

6.6 Summary

This chapter introduced the replicator control operator, which allows a particularly flexible use and control of generators in a parallel environment. It is demonstrated in ALLOY how these can be used to express a wide range of algorithms for searching a space of solutions, from rigid control backtracking up to unhampered full parallelism.

Chapter 7

Logic Programming Styles

In logic programming languages it is too often the case that the programmer is forced to hide what was meant behind vague and complicated expressions using general purpose features. It is shown how selective use of functions with generators and/or special objects enhances clarity, efficiency and power. This chapter also describes the mechanisms which the language ALLOY provides for the support of full logic programming styles.

7.1 Introduction

This chapter describes techniques for programming in styles like that of Prolog, or-parallel Prolog, and Parlog. It shows also that the full requirements (and complexities) of a logic programming style (i.e. backtracking, logical variables, and unification) are often not needed.

Many program components have a functional nature, for which logic programming languages are not suitable. It is claimed that Logic Programming Languages can best be thought of as special cases of functional languages extended with searching abilities and object oriented programming. In such a case, selective use of functions with generators and/or special objects provides clarity, efficiency, or-parallelism, and the option to evaluate lazily or eagerly.

7.2 Logic Programming and Styles

This section presents programs written in Prolog, or-parallel Prolog, and Parlog. It gives examples written in various styles in each language.

7.2.1 Prolog

A Prolog [Col72,Kow74] program consists of predicates. A predicate consists of clauses. A clause (Definite Horn Clause[1]) consists of a Head and a Body. The head is a term[2] while the body is a conjunction of terms (predicate calls). These heads have the same term-name, the name of the predicate. The head of a clause is true if its body is true.

Programs in Prolog use resolution [Rob65] to find out the variable bindings for which a goal term is true. Resolution is based on unification and in Prolog proceeds in a top-down left-to-right fashion while the program clauses are selected in order of appearance (Selective Linear Definite or SLD clause resolution). The process is deterministic.

Variables in Prolog are called *logical variables* since they are objects which can be passed around or be made part of other structures. A logical variable can be bound to a value if and only if it is not bound to one already. After backtracking a variable may become unbound.

The process of backing up from a search branch that fails to resolve is called backtracking. This process involves both control and data backtracking. Backtracking is used until a successful branch is found.

A program often used to demonstrate the flexibility of programs written in Prolog is the `append` predicate:

```
append([H| T], L, [H| T2]):- append(T, L, T2).
append([], L, L).
```

This predicate can be used with variables at any argument or any position of an argument. Thus, it can append or split lists.

[1] In AI terminology, *clause* is a set of literals representing their disjunction, *Horn Clause* is a clause with at most one positive literal and *Definite Horn Clause* is a Horn Clause with exactly one positive literal.

[2] Since clauses are DHCs the head has to be a positive literal. This is what Prolog is lacking from first order logic. For the same reason treatment of negation is either incomplete (eg. works on ground terms only) or not sound (eg. negation through failure)

While the above predicate is impressive, often reversible algorithms are either hard (e.g. copy elements of a list to an array,) impossible (e.g differentiation,) or just meaningless (e.g. n-queens problem). The proliferation of the *cut* operation in Prolog Programs is another indication that multi-directional programs are the exception rather than the rule.

Predicate `fibonacci` unifies the third argument with successive elements of the fibonacci sequence with initial values the first two parameters. Though it could be made to use as output whichever parameter the caller leaves uninstantiated, in practice it would not be very useful and since it is not obvious how to implement it or what its exact behavior would be, the extra feature is ignored. Thus, predicate `fibonacci` is written in a functional style:

```
fibonacci(X, Y, Y).
fibonacci(X, Y, V):- Z is X + Y, fibonacci(Y, Z, V).
```

Finally predicate `length` returns the length of a list. For similar reasons it is written in a functional style:

```
length([],     L):- L is 0.
length([_| T], L):- length(T, L2), L is L2 + 1.
```

7.2.2 Or Parallel Prolog

Here programs take advantage of multiprocessor systems. A predicate often mentioned with these languages is the `intersection` of two lists:

```
intersection(L1, L2, L):- setof(X, common(L1, L2, X), L).

common(L1, L2, L):- member(X, L1), member(X, L2).

member(X, [X| _]).
member(X, [_| T]):- member(X, T).
```

7.2.3 Parlog

An extended description of PARLOG was given by Gregory [Gre87]. Its main objectives include the provision of stream and-parallelism, the logical variable, efficient compilation

and powerful metalevel programming (process control). PARLOG uses modes for parameters as a syntactic convenience and provides provides both serial and parallel disjunction and conjunction.

Whenever there are many producers of some values predicate `merge` is used to combine the various results, which come in streams, to one stream:

```
mode merge(in, in, out).

merge([X | U], V, [X | Z]):- merge(V, U, Z).
merge(U, [Y | V], [Y | Z]):- merge(U, V, Z).
merge([], [], []).
```

Another predicate often mentioned with these languages is `isotree` which checks if two trees are isomorphic. This program makes use of both or-parallelism and and-parallelism:

```
mode isotree(in, in).

isotree([Left1 | Right1], [Left2 | Right2]):-
        isotree(Left1, Left2), isotree(Right1, Right2) : true;
isotree([Left1 | Right1], [Left2 | Right2]):-
        isotree(Left1, Right2), isotree(Right1, Left2) : true.
isotree(T, T).
```

It is possible to use the logical variable to suspend and resume execution of a predicate and finally achieve lazy evaluation. It may require some effort until the technique is mastered. For example consider how one may write a predicate that returns the squares of all integers:

```
mode squares(?), squares(?, ?), integers_from(?, ?).

squares(List):- integers_from(1, Ints), squares(Ints, List).

squares(Ints, [S| List]):- Ints=[X| Irest], prod(X, X, S),
                           squares(Irest, List).

integers_from(I, [Int, Irest]):- Int=I, sum(I, 1, I1),
                                 integers_from(I1, Irest).
```

In this way, the square of an integer is evaluated only if it is requested. Using similar techniques it is easy enough to write a lazy predicate which evaluates all primes or one that can function as a bounded buffer to serve as an amortizing mechanism among producers and consumers.

7.3 Comparisons

This section gives implementations of the above programs in ALLOY. In some cases, library `prolog` is used. It is defined in the next section.

7.3.1 Prolog styles

In general cases where prolog programs make full use of the logical variable and backtracking the solution provided in ALLOY is to use the `prolog` library. This solution however results in verbose programs (see subsection 7.4.4 for a simpler version):

```
[setfun p-app(p1 p2 p3)
   [lets (((l) (lvs 1)))
      (snest ^!*(unify p1 ()) ^!*(unify p2 l) ^!*(unify p3 l) ^(return))]
   [lets (((l t r h) (lvs 4)))
      (snest ^!*(unify p1 (cons h t)) ^!*(unify p3 (cons h r))
             ^!*(p-app t p2 r) ^(return))]]
```

The main difference is that here logical variables have to be initialized explicitly and unifications in the head of each clause have to be moved into the body. Depending on the ALLOY implementation of generators, this code may be inefficient (by a constant factor) when compared to the Prolog code.

As explained above, the arguments of Prolog predicates are often used either for input or for output but not for both. In these cases, the functional definition can be cleaner. This is the definition of function `app` in ALLOY which appends two lists:

```
(setfun append(l1 l2)
    (return (if (cons-p l1) (cons (car l1) (append (cdr l1 l2))) l2)))
```

In complex cases, functional algorithms can be much easier to understand or produce efficient code for.

This is the definition of function `fibonacci` in ALLOY:

```
[setfun fibonacci(x y)
    (return y)
    #!(return *(fibonacci y (sum x y))]]
```

Values are generated by this function on demand. This is due to properties of the `return` statement in generators (see chapter 6).

Finally, when arguments are directional and backtracking is not needed the algorithm is clearly functional. Thus, the simplicity of function `length`:

```
(setfun length(l)
   (return (if (cons-p l) (sum 1 (length (cdr l))) 0)))
```

7.3.2 OR-parallel Prolog

Such ALLOY programs making full use of the logical variable should make use of the `prolog` library. Here is the direct OR-parallel Prolog-like implementation of predicate `intersection`. Each predicate returns the new values of its arguments in a list.

```
(setfun intersectionp(lp2 lp3)
   (return (let [((x) (lvs 1))]
             [list #[rstrip *(commonp x lp2 lp3)]])))

(setfun commonp(lp1 lp2 lp3)
    (lets [((x l1 l2) (copyloge (list lp1 lp2 lp3)))]
       #(lets [((x2 l1_2) *(memberp x l1)
                ((x3 l2_2) *(memberp x l2))
          (when (unify x2 x3) (return (list x3 l1_2 l2_2)))]

[setfun memberp(lp1 lp2)
   [lets [((p1 p2) (copyloge (list lp1 lp2)))]
      (snest ^!*(unify (cons p1 ('new logvar)) p2)
             ^(return (list p1 p2)))]
   (lets [((x p2) (copyloge (list lp1 lp2)))
          ((r h) (lvs 2))]
      (snest ^!*(unify p2 (cons h r))
             ^![lets [((X2 R2) *(memberp x r))]
                     (return (list X2 (cons h r2)))]))]
```

Again the problem is verbosity. Also, the code is not efficient.

We have some extra flexibility here. If we prefer that predicates in conjunctions do not evaluate in parallel we can replace predicate `commonp` with:

```
(setfun commonp(lp1 lp2 lp3)
    [lets [((x l1 l2) (copyloge (list lp1 lp2 lp3)))]
       #[lets [((X2 l1_2) *(memberp x l1))]
              #[lets [((x3 l2_2) *(memberp x2 l2))]
                     (return (list x3 l1_2 l2_2))]]])
```

However, since in general arguments are directional, as is the case in this example, it is more natural to use a functional style:

```
(setfun intersection(s1 s2)
```

```
  #(lets ((x *(getmember s1)) (y *(getmember s2)))
      (when (eql x y) (return x))))

(setfun getmember(l)
  (when (cons-p l) (return (car l))
                   #!(return *(getmember (cdr l)))))
```

7.3.3 Programming in Parlog Style

In PARLOG it is even more common that arguments have directionality. The few cases where this is not true usually have to do with emulation of objects or lazy evaluation. ALLOY directly supports object oriented programming and lazy evaluation.

Simple programs such as the merge function are also simple in ALLOY. The definition of function `merge` follows:

```
(setfun merge(l1 l2)
  (when (null-p l1) (return l2))
  (when (null-p l2) (return l1))
  (when (cons-p l1) (return (cons (car l1) (merge l2 (cdr l1)))))
  (when (cons-p l2) (return (cons (car l2) (merge (cdr l2) l1)))))
```

The same definition can be called lazily. If recursive calls were made replicative this function could be called as a generator to return all possible resulting streams. In PARLOG the second is not supported at all while the first would require a different program.

In PARLOG OR-parallel calls must be *safe* even though safety is an undecidable property. In ALLOY if safety is needed it is sufficient not to pass writable objects to functions called in parallel[3]. Here is function `isotree` in ALLOY:

```
(setfun isotree(t1 t2)
   (if (and (cons-p t1) (cons-p t2))
     (if (or-call ^(and-call ^(isotree (car t1) (car t2))
                             ^(isotree (cdr t1) (cdr t2)))
                  ^(and-call ^(isotree (car t1) (cdr t2))
                             ^(isotree (cdr t1) (car t2))))
         (return t1) (return %f))
     (return (eql t1 t2))))
```

[3]This does sound like Concurrent Prolog's abilities, though passing non-writable objects does not have to be done at run time. Synchronizing values are already non writable.

The functional nature of ALLOY makes lazy evaluation trivial and clean. The following function returns the squares of all integers as a lazily evaluated stream:

```
(setfun squares()
   (return (lazy (squaresl (ints 0)))))

(setfun squaresl(l)
  (return (cons (times (car l) (car l)) (squaresl (cdr l)))))

(setfun ints(x)
   (return (cons x (ints (sum x 1)))))
```

This is the same as the eager version with the exception of the `lazy` declaration at the top level. If it is not clear that the functional lazy solution is much cleaner than the logic lazy solution the reader is invited to compare the lazy implementation of prime numbers in ALLOY (chapter 2) with the PARLOG solution [CG86].

7.3.4 And and Or parallel algorithms

Problems such as n Queens (see chapter 2), can make use of both AND and OR parallelism. In the particular case of the n Queens problem, AND parallelism can be used in testing for safety at the same time a permutation of the Queens is being created. OR Parallelism can be used in creating the permutations of the Queens in parallel. This is not possible in either or-parallel Prologs or PARLOG. Though it is possible in full Concurrent Prolog no implementations of it are available while research indicates that properties of the language could make an efficient parallel implementation impossible. Attempts to translate such programs to primarily and-parallel logic programming languages appear to face complications [Ued87a,Ued87b] and often introduce serializations [Mit88a,Mar88].

Problems which can be expressed functionally can make use of both AND and OR parallel evaluation in ALLOY. Some solutions to the n Queens problem are given in chapter 6. These make depth first or breadth first search and can return one or all solutions.

7.4 Supporting Library

This section describes the library of functions providing the logical variable, one way unification, backtrackable unification, the cut operation and function `snest`. Library file

`prolog.a` makes sure all these utilities are available.

7.4.1 Prolog

This top level file defines utilities to create logical variables, make successive backtrackable calls, the `cut` operation and also makes sure everything else is loaded.

```
(provide "prolog")
(require "unify")

(setfun lvs(i)
   (if (gt i 0) (return (cons ('new logvar) (lvs (diff i 1))))
                (return nil)))

(setfun (snest . l) (snestl l))

(setfun snestl(l)
   [when (cons-p l) #[s *((car l)) (snestl (cdr l))]])

(setfun (s . x))

[setfun cut(global-fail)
   (return 1)
   (global-fail)]
```

Note the simplicity of the definition of function `cut`.

7.4.2 Unification

This file provides unification mechanisms for logical variables. The available operations are: one-way unification, full unification and backtrackable full unification. Backtrackable unification makes use of the ability to control the generation process (chapter 6). During backtracking the unification operation un-does its bindings.

```
(provide "unify")
(require "logvar")

;---------------------------
; Simple, non backtrackable unification.
(setfun s-unify(l1 l2)
   (return (simpl-unify l1 l2)))

;---------------------------
; Simple, non backtrackable left unification.
(setfun s-unifyl(l1 l2)
   (return (simpl-unifyl l1 l2)))
```

```
(setfun simpl-unifyl(l1 l2)
   (when (logvar-p l1)
      (return (if ('put l1 l2) l1 (simpl-unifyl ('get l1) l2))))
   (when (and (cons-p l1) (cons-p l2))
      (return (and-call ^(simpl-unifyl (car l1) (car l2))
                        ^(simpl-unifyl (cdr l1) (cdr l2)))))
   (when (and (scons-p l1) (scons-p l2))
      (return (and-call ^(simpl-unifyl (scar l1) (scar l2))
                        ^(simpl-unifyl (scdr l1) (scdr l2)))))
   (when (atom-p l1) (return (eql l1 l2))))

(setfun simpl-unify(l1 l2)
   (when (and (cons-p l1) (cons-p l2))
      (return (and-call ^(simpl-unify (car l1) (car l2))
                        ^(simpl-unify (cdr l1) (cdr l2)))))
   (when (and (scons-p l1) (scons-p l2))
      (return (and-call ^(simpl-unify (scar l1) (scar l2))
                        ^(simpl-unify (scdr l1) (scdr l2)))))
   (if (logvar-p l1)
      (return (if ('put l1 l2) l1 (simpl-unify ('get l1) l2)))
      (if (logvar-p l2)
         (return (if ('put l2 l1) l2 (simpl-unify ('get l2) l1)))
         (when (atom-p l1) (return (eql l1 l2))))))

;---------------------------
; Backtrackable unification.
(setfun unify(l1 l2)
   [when (and (cons-p l1) (cons-p l2))
      #(return (cons *(unify (car l1) (car l2))
                     *(unify (cdr l1) (cdr l2))))]
   [when (and (scons-p l1) (scons-p l2))
      #(return (scons *(unify (scar l1) (scar l2))
                      *(unify (scdr l1) (scdr l2))))]
   (if (logvar-p l1)
      [block #(return *(unifyput l1 l2))]
      (if (logvar-p l2)
         [block #(return *(unifyput l2 l1))]
         (when (atom-p l1) (when (eql l1 l2) (return l1))))))

(setfun unifyput(l1 l2)
   [lets [(_ok %f)]
      #(return (block1 *('put l1 l2) (set _ok %t)))
      [unless _ok #(return *(unify ('get l1) l2))]])

(setfun copyloge(e)
   (return (let ((d ('new dictionary)))
             (copy-loged e d))))

[setfun copy-loged(e d)
   (when (cons-p e)
      (return (let [(eleft  (copy-loged (car e) d))
                    (eright (copy-loged (cdr e) d))]
                (if (and (eql (car e) eleft) (eql (cdr e) eright))
                   e
                   (cons eleft eright))))))
```

```
   (when (atom-p e)
      (return (let ((es (mstrip e)))
                (if (logvar-p es)
                   (let [(esc ('get d ('age es)))]
                      (when (var-p esc) ('put d ('age es) ('new logvar)))
                      esc)
                   es)))]
```

7.4.3 Logical Variable

The logical variable is defined here in terms of a multiple assignment variable. The ability of this variable to undo its bindings appears in function `put`. The un-binding takes place when called as a generator and requested to return a second solution. No second solution is returned but the binding is undone.

```
(provide "logvar")
(require "faa")
(require "sa-var")

(class-start logvar
  (static _value order novalue)
  (import eql printnl strip)
  (common faa)
  (methods get put age var))

(setfun new(x)
   (set novalue 'novalue175342)
   (set _value novalue)
   (set order ('faa c-faa 1)))

[setfun var()
   (if (eql _value novalue) (return %t) (return %f))]

(setfun get()
   (return (if (eql novalue _value) here _value)))

(setfun put(vv)
   (when (var)
      [block (return (set _value (strip vv))) (set _value novalue)]))

(setfun age()
   (return order))

(class-end logvar)


(setfun strip(x)
   (return (if (var-p x) x (if (logvar-p x) ('get x) x))))
```

```
(setfun mstrip(x)
   [lets [(sx [strip x])]
      (if (var-p sx)
         (return sx)
         (if (eql x sx)
            (return x)
            (return [mstrip sx]))))])

(setfun rstrip(x)
   [lets [(sx (mstrip x))]
         (when (var-p sx)   (return sx))
         (when (cons-p sx)  (return (cons  [rstrip (car  sx)] [rstrip (cdr  sx)])))
         (when (scons-p sx) (return (scons [rstrip (scar sx)] [rstrip (scdr sx)])))
         (when (atom-p sx)  (return sx))])

(setfun logvar-p(v)
   (return (eql ('object-p v) 'logvar)))
```

7.4.4 Automating the Translation

Transformations from logic programming languages to ALLOY using the above library can be automated easily. For serial and and-parallel logic programming languages the transformations presented are efficient, though for or-parallel logic programming languages it would be much better, if possible, to replace the predicates by functions.

However, the transformations are not so verbose as to be very hard to do by hand. Of course, predicates which behave like functions should preferably be written directly into ALLOY.

Another way to facilitate translation of Prolog programs to ALLOY is the use of some macros. Two macros, `p-clause` and `p-cut`, could be provided for this purpose. Thus, function/predicate `p-app` would become simpler:

```
[setfun p-app(p1 p2 p3)
   [p-clause (l)
      (unify p1 ()) (unify p2 l) (unify p3 l)      (p-cut)]
   [p-clause (l t r h)
      (unify p1 (cons h t)) (unify p3 (cons h r))  (p-app t p2 r)]]
```

7.5 Summary

It has been shown that general programs written in serial Prolog, or-parallel Prolog, or Parlog can in general be translated into ALLOY. This general transformations, though inefficient (assuming simple compilers) by a constant factor can be mixed as desired. In the case of or-parallel prolog the transformation is naive. All transformation can be done automatically by a simple pre-processor.

It has been shown that a considerable number of programs written in these languages can be written in an ALLOY style. These styles are more efficient and clear since they are functional, more flexible since they can evaluate eagerly or lazily, and more powerful since they can provide single or multiple solutions. These problems (e.g the n Queens problem) can make use of both AND and OR parallel evaluation. This last ability is not available in any of the current Logic Programming Languages.

Having the synchronizing variable as the basic variable instead of the too powerful logical variable is one reason for the above advantages. Using the replicator/generators instead of either backtracking or or-parallelism is another reason.

Chapter 8

An Interpreter Simulating Parallelism

This chapter describes an interpreter of ALLOY implemented in Common Lisp. The description is followed by some benchmarks. Finally some comments on potential opportunities and problems of implementations on MIMD shared memory and distributed architectures are presented. The full source code of the interpreter is given in appendix A. The full suite of benchmarks is given in appendix B.

8.1 Introduction

An implementation of ALLOY is dominated by the need to support:

- Serial/parallel fine grain execution.
- Eager/lazy evaluation.
- Powerful commitment.
- Synchronizing variables.
- Closures.
- Objects and dynamic variables.
- Uniformity of classes, objects, and closures.
- Generators and their behavior during serial/parallel and eager/lazy evaluation.
- Replicators with nested/non-nested generators.

- Garbage collection.

Other important issues include:

- Pattern matching for function arguments.
- Ability for any function to be called as a generator.
- Built in expressions, functions and objects.
- Pre-processing for faster interpretation.
- Interactive environment.

ALLOY v2.0, described in this thesis, has been implemented fully as an interpreter in Common Lisp. This implementation has been focused on simplicity so that changes could easily be made and new features could easily be added. This implementation provides all the features described in chapter 3 including a programming environment and some libraries.

One important reason for this implementation has been assistance in making ALLOY clear, complete, and efficient. As it turned out, this implementation helped in the illumination of a number of dark corners. Lisp was selected as the language of the implementation because it allows quick changes, and provides powerfull utilites as well as a garbage collector.

The other important reason for this implementation was to test and debug ALLOY programs. These programs are very important in demonstrating ALLOY's strength.

When this interpreter is compiled using Austin Kyoto Common Lisp v 1.530 it executes with a speed of 600 function calls (or messages) per second on a SparcStation 1. Programs which use many generators may be expected to run slower since in this implementation the replicator/generator structures get replaced by many regular calls before they are executed.

8.2 The interpreter in Lisp

This section describes a full interpreter of ALLOY in Common Lisp. A short description of the logical parts for this interpreter is followed by an extensive description of the important data structures used. The complete sources of the interpreter are given in appendix A.

In this interpreter a running program is a set of processes. Of these processes those which are ready to execute are located either in a queue or a stack. Normally, new

processes are placed and taken off the stack but in cases when it is known that some processes would probably suspend for some time, they are placed in the queue.

After a (user definable) number of processes have been taken off the stack a process is moved from the stack to the queue (to intensify non-determinism). When the stack is empty processes are taken for execution from the queue.

When both the stack and the queue are empty, the prompt is returned to the user. In a parallel implementation the prompt could be returned to the user while processes were still executing.

8.2.1 Parts of the System

The important logical parts of the ALLOY system are presented here. These should be the first step towards the understanding of the implementation.

Top Level

It is defined in subsection A.1.2 and subsection A.1.3. It provides the *read-eval-execute* loop. The prompt is given to the user as soon as there are no more executing processes. The input together with a command to print its value is translated into the internal representation, is made known to the interpreter, and control is passed to the interpreter. When the interpreter has nothing to do, control is returned to the read-eval-execute loop.

Parser

The actual parser is defined in subsection A.4.1. It is an interface to the Common Lisp parser enhancing it with the ability to recognize square lists, generators, the replicator, and closures.

The function for reading an ALLOY expression is `al-read` with optional argument an input stream (default is the console). The function for printing an ALLOY expression is `al-print` with optional argument an output stream (default is the console).

Pre-processor

The pre-processor checks that built-in expressions are correctly formed, replaces expressions with lower level expressions and simpler function calls, and returns the result. The

main part of the pre-processor is defined in subsection A.3.1 and the main function is `al-prep`.

The basic part of the interpreter is defined in subsection A.3.1. Pre-processing of functions and functional expressions is done here. Expressions are checked here that they are well formed and some are translated into code which is more efficient to interpret. The most complex of these expressions is the `let` expression.

A large part of pre-processing is defined in subsection A.3.2 and deals with the pre-processing of generators and replicators. It is charged with the task of identifying replicators and the properties of the environment in which they operate. Then it places generators into groups with the appropriate drivers and flags declaring the required behavior, and creates the general form of replicable expressions. The main function for the pre-processing of generators is `gprep`, which is called by the main pre-processor.

Finally, the part of the pre-processor defined in subsection A.3.3 pre-processes class definitions, making sure they are well formed, and replacing them by expressions easier to handle. The main function for this is `oprep` which is also called by the main pre-processor.

Functional Features

The main implementation of functional features is given in subsection A.2.1. These are serial/parallel evaluation of calls, eager/lazy evaluation, calls of closures and built in functions, local and global commitment, handling of synchronizing variables, and some basic built in expressions.

Each function becomes a process. Each process knows to which synchronizing variable it should its result. Each process can suspend waiting for a synchronizing variable to get a value. Each process can suspend waiting for another process to ask that a synchronizing variable gets a value (i.e. producer of lazily evaluated value).

Object Oriented Features

The implementation of these features is given in subsection A.2.2 and even though it is lengthy, it has no complications. Much of it deals with keeping the representation of objects and classes consistent. Also, it defines the behavior of predefined messages for objects and classes.

Generators and Replicator

These are defined in subsection A.2.3 and subsection A.2.1. Their implementation is rather involved. One must keep in mind that at each point information about lazy evaluation and order of evaluation must be passed along consistently.

For efficiency, these functions as well as drivers of the generating and replicating process, have to manage internal ALLOY processes at the lowest level.

Built in Functions and Expressions

Expressions (special forms) are different from function calls in that they do not follow the normal evaluation rules. Their arguments are evaluated (if at all) in special ways. They are declared using function `add-spec` in subsection A.5.1 and the basic ones are defined in subsection A.2.1, while others which are only for internal usage from the interpreter are declared elsewhere. Since these expressions are operating at a very low level, they communicate with the interpreter at a very low level, using the following functions:

- `(give-value <value>)` makes `<value>` the returned value of the expression.
- `(waitfor-val <variable>)` declares that the current expression suspends execution waiting for `<variable>` to be set.
- `(next-the-same)` declares that the expression must be evaluated again immediately. Usually this happens after part of the process representing the expression has undergone some changes.

Built in functions are declared using function `set-noninst` whose second argument specifies whether the function should be immediately available to objects or not (in which case it must be imported). The implementation of these functions is given in common Lisp and those common Lisp functions interface with the ALLOY interpreter by returning values in the following special ways:

- `(sus <variable>)` means that the function wants to suspend waiting for that `<variable>` to be set.
- `(suc <value>)` means that the function wants to return `<value>`.

Built in Classes

Built in classes defined in Common Lisp have two forms. Classes for "regular" objects such as vectors, dictionaries etc. and classes for low level objects such as lists, numbers, strings, etc.

Classes of regular objects are declared using function `set-bclass` and the implementation of their messages and those of their instances is given in two functions whose names consist of the name of the class prefixed by `cl-c` and `cl-o` respectively.

Definition of classes for low level objects (such as numbers and strings) in addition require that an instance of their class is saved in a constant and that function `send-bi-obj` in subsection A.5.3 is informed of their existence.

8.2.2 Data Structures of the Functional Part

The following describes the main data structures used by the interpreter. These are used to describe synchronizing variables, closures, processes and environments.

Synchronizing Variables

Synchronizing variables have the responsibility of handling suspension of processes and lazy evaluation. The data structure which represents them is called `alval` and is defined in subsection A.1.3. The various fields of it are:

- `init` is true if the variable has been set to some value (which could be another variable). Otherwise it is nil.
- `value` contains the value of the synchronizing variable if it has one.
- `suspend` contains pointers to processes which cannot proceed until this variable has been set.
- `delayed` may contain a pointer to the process which is responsible for giving a value to this variable but is being delayed (or in a lazy evaluation mode).

Closure

A closure is represented by a simple data structure. That data structure is called `ext-function` and is defined in subsection A.1.3. It has the following fields:

- type gives the internal type of the corresponding function which can be:
  - `gfunction` for a global function (defined at the top level of an object).
  - `gbfunction` for a global function which is built in (cannot be redefined).
  - `noninst` for a built in function of low level defined in the same language that the interpreter is written.
  - `argone` is similar to `noninst` except that the implementation function expects all arguments in one list.
- `expr` contains a pointer to the code of the function.
- `lmu` contains a pointer to the environment of the function's definition.

Processes

The data structures used to represent processes are rather complicated. Processes need to know the code they are expected to evaluate, where to place their result, the environment of their execution, the mode of execution (eager/delayed/lazy), whether another process is waiting (among others) for it to complete, and a pointer to that other process. The data structure is called `proc` and is defined in subsection A.1.3. The various fields are:

- `call` contains a pointer to the code this process is expected to evaluate.
- `result` contains a pointer to the synchronizing variable into which the result must be stored.
- `lmu` contains a pointer to the environment in which it executes.
- `waitfor` contains an integer equal to the number of processes whose execution must complete before this process can start execution.
- `suspended` contains a pointer to a process which is waiting for this process (among others) to complete execution before it (the suspended process) can start execution. When the current process completes execution the `waitfor` number of the suspended process is decremented. When that number becomes 0 the suspended process is started.
- `lazy` contains the mode of execution (eager, delayed or lazy). The possible values are:

- `nil` Eager.
- `delay` Delayed.
- `(t)` Lazy.
- `t` Lazy but must give a value now.

Environment

The environment in which a process executes is represented by the most complicated data structure in this implementation. A new environment is created every time a function defined in ALLOY is called. For each function defined in ALLOY that executes there is an exclusive environment given to it. The data structure is called `muinfo` and is defined in subsection A.1.3. The fields of this data structure are:

- `gmu` A pointer to the environment of the call to the global (non-local) function which defined this function.
- `pmu` A pointer to the environment of the function which called this one. This is useful when some action needs to be charged to the caller (i.e. arguments of a function call).
- `chmus` the list of environments of children functions.
- `lenv` the local part of the variable bindings (deep association list).
- `die` is true if processes accessing this environment are declared dead, nil otherwise.
- `commit` true if the function call using this this environment has committed, nil otherwise.
- `result` is a pointer to the variable where the result of the called function must be given.
- `suspended` the process suspended waiting for this function to complete execution. Similar to the filed in the data structure of a process.
- `lazy` is true if the function has been called lazily. In this case the interpreter must continue executing lazily after commitment.
- `top` is true if the function has been called by the top level of the interpreter.

- `object` is a pointer to the description of the object in which this function has been defined.

8.2.3 Data Structures for Classes

These are the most important data structures of those used exclusively for the representation of classes and objects.

Classes

A class must know of the special properties given to it, provide access to variables common to all instances of the class with their values, and also a list of the dynamic variables for convenience. The structure is called `class` and is defined in subsection A.2.2. It has the following fields:

- `origdef` contains the list of properties given to this class by expression `class-define`.
- `tab` is a symbol table of variables with their values which are common to all instances of this class and no other.
- `ndynnammes` is a list containing the number of dynamic variables followed by a list of those names and a list of those names each prefixed by the string `next-`.

Objects

The data structure used to describe objects is called `object`, is defined in subsection A.2.2, and has the following fields:

- `class` is a pointer to the object's class.
- `interface` is the list of messages accepted by the object.
- `inienv` is a deep association list for binding of this object.
- `nextdyn` is the list of the `next-` dynamic variables which will be current to the next object.
- `createvar` is true if this object can create global static variables at run time (i.e. the top level object can do that).

8.2.4 Data Structures for Replication and Generators

Replicators and Generators have the most complex implementation of all features in ALLOY. One reason for the complexity, and one of the advantages of ALLOY, is that this simple control structure gives the writer of a compiler many opportunities for optimization.

Because many opportunities for optimization arise an efficient implementation of this control structure would be lengthy though not very complicated. Since this implementation is meant to be simple and short, this control structure is given a very general implementation.

The behavior of generators and the replicating process is different depending on the environment in which the generators and the replicator exist. Depending on whether parts of the structure are in a serial, parallel, eager, lazy, nested or flat environment the structure can be compiled to take full advantage of the special usage.

Fake generators

Any ALLOY value (including Numbers, atoms, closures, etc.) as well as built in functions implemented in the language in which the interpreter is written can be treated as if they were generators. Therefore a special data structure is needed to represent them. The data structure is called `fgrec` is defined in subsection A.2.3 and has the following fields:

- `result` is a list of the single result of the fake generator.
- `expr` is the fake generator itself.
- `lazy` is the mode in which the generator must evaluate.
- `act` is the action that must be taken when a value is being requested by the generator. It can be:
  - `first` if a value has not been requested yet.
  - `wait` if a value has been requested already.
  - `regular generator` if the fake generator is, in fact, a normal generator (i.e. a real ALLOY closure).

Normal Generator

A normal generator is a function defined in ALLOY, in other words a μ closure. The data structure used to represent it is called `genstate` and is defined in subsection A.2.4. Its fields are:

- `result` is the list of results is generated up to that moment. The list may end in a synchronizing variable.
- `finished` is the queue of `gval` structures each containing a process. The process is the one executing a return (i.e. generating a result). That process suspends until that value has been requested.
- `resend` is the tail of the `results` list. It is either a list or a synchronizing variable. That is where the next generated element will be placed.
- `next` is a tail of the `results` list. The car is the next element to be given.
- `rest` holds the list of values which must follow after the generated ones. Used for testing. Normally, it is always nil.
- `suspend` is a queue of processes . Every time a new value is requested from this generator a processes is dequeued and gets awaken. In the beggining the queue only contains the generator function. Later, every process returning a result for that generator places its continuation in that queue. This is needed because the `return` expressions of generators can be controled by the callers of the generators.

Multiple Generators

The results of multiple generators all of which are at the same level are combined in all possible ways and the resulting expression containing them is executed.

Again, one difficulty is to preserve order in serial execution and mode of evaluation. The data structure is called `genstate` and is defined in subsection A.2.4 of chapter A. Its fields are:

- `results` is the list of all results of the expression with the multiple generators.
- `resend` is the tail of the *results* list. Useful to trigger generation of the next element.
- `sergens` true if the generators must be called the one after the other.

8.3 Practical Considerations

This section deals with a number of practical issues.

8.3.1 Portability

Porting this interpreter to another Lisp system should be a simple task. Areas that would need changes may include input/output, hash tables, and structures. Macro definitions and variable number of arguments as well as tagged arguments may also need changes.

Porting this interpreter to C would not be quite so easy. The major difficulty would be the requirement of a garbage collector. Also, the programmer would have to write many supporting functions. However porting this interpreter in C would allow for many optimizations of both time and space.

8.3.2 Benchmarks

On Austin-Kyoto Common Lisp about 60 function calls are performed per MIPS per second. This translates to about 600 function calls per second on a Sparcsation 1. Generators can be expected to be slower than that since they are translated into many internal processes.

The results of some benchmarks are given in the following tables. A small description of these benchmarks and their code, is given in appendix B.

Basic functional:

Benchmark	*msecs*	*Cycles*	*Calls*	*Procs*	*Envs*	*Vars*	*PPS*	*EPS*	*PPE*
app3	50	56	35	52	10	22	1040	200	5.2
nrev30	5500	6374	3953	5878	994	2421	1068	180	5.9
length100	2067	2314	1505	2112	406	808	1021	196	5.2
whut80	2439	2725	1797	2420	530	1026	924	217	4.5
qsort11	927	1247	805	1119	163	1463	828	175	6.8
ssort11	2433	2865	1834	2699	514	1277	1104	211	5.3
dine3-4	4274	4336	3134	3899	1290	1847	912	301	3.0

Lazy evaluation and generators:

Benchmark	*msecs*	*Cycles*	*Calls*	*Procs*	*Envs*	*Vars*	*PPS*	*EPS*	*PPE*
lprimes5	491	558	291	467	56	170	949	113	8.3
lprimes25	15426	15602	8601	12907	1366	4430	836	87	9.4
inters33	350	623	458	537	32	300	1534	91	16.8
perms3	1927	2859	2047	2388	127	1212	1256	66	18.8
perms4	14117	12724	9157	10518	582	5340	745	41	18.0
queens4	13150	10057	6868	9222	728	4221	701	55	12.6
queens4a	28034	24603	16813	22509	1782	10219	802	63	12.6
prapp2	9517	8489	5856	7890	1017	3738	829	106	7.7
prapp2or	12217	14780	10138	13837	2001	6430	1132	163	6.9
pinters33or	55183	44219	30015	41724	5490	19228	756	99	7.6

Where:

- *msecs* is the time in milliseconds needed for execution on a Sparcstation 1.
- *cycles* is the number of times the scheduler fetched a new process.
- *calls* is the total number (normal and internal) of function calls. This includes normal function calls (appearing in the original program), and internal function calls generated by the interpreter.
- *procs* is the numbers of processes created.
- *envs* is the number of different environments created (closures called).
- *vars* is the numbers of synchronizing variables created.
- *PPS* is processes per second.
- *EPS* is environments per second.
- *PPE* is processes per environment.

8.3.3 Efficiency and Compilation

We see that the number of *EPS* for eager programs without generators is virtually stable at about 190. Since the number of ALLOY function calls per closure calls (environments created) is also stable at about 4-5 we can make general conclusions from specific programs. By examining programs `length100` and `nrev30` we see that the first does 1206 and the second 3474 function calls. Dividing these numbers by the time they take to execute, we see that about 600 functions are called per second.

Of the above table, of particular importance are values *PPS*, *EPS* and eventually their ratio *PPE*. Unless the interpreter is written in a faster language such as C or the interpreter is replaced by a compiler, the number of *PPS* created is virtually stable. The high number of *PPE* implies inefficient execution. An efficient implementation would keep the number of *PPE* to well below 2 maybe even below 1. This would reduce the number of *PPS* but probably by no more than 50 per cent.

For the above reasons, it is reasonable to assume that an efficient compiler would be at least 5 times faster than a naive compiler. It is also reasonable to assume that a naive compiler of ALLOY in C would be at least 20 times faster than this naive interpreter of ALLOY in Lisp. If these assumptions are correct, an efficient compiler for ALLOY in C would be at least 100 times faster than this naive interpreter. This would translate to an average of about 60,000 function calls per second on a SparcStation 1.

8.3.4 Ineffiecient Implementation of Features

The large number of *PPE* with lazy evaluation and the even larger number of *PPE* with replicators and generators implies that these features have not been implemented very efficiently. In particular, the implementation of replicators and generators is general and thus special cases are identified at run time. That is the main reason for the large number of processes executed per created environment.

8.3.5 Using the Interpreter

To call ALLOY after the interpreter has been loaded, execute `(ALLOY)`. This enters the user in the read-evaluate-execute loop. The user may use `^C` to terminate execution of ALLOY. That takes the user to the Lisp system. After exiting the debugger the user can re-enter ALLOY in the normal way.

For more details on the use of ALLOY see chapter 3 and chapter 2.

8.4 Implementations on Parallel Architectures

Even though ALLOY is a higher level programming language it encourages the use of features which result in programs with massive parallelism. A program can be written to take advantage of the underlying architecture.

8.4.1 Shared Memory MIMD

There are aspects of ALLOY which can be used to exploit the power of shared memory MIMD architectures. There is also support for the fetch and add primitive.

One such aspect is the highly parallel nature of replication especially in relation to the combination of values returned by generators. This, and its utilization of the highly parallel `fetch-and-add` operation are described further in chapter 6. Also, a large number of (often heavy weight) processes with the same or different environment can be generated swiftly.

Another important aspect is the dynamic variables of objects. An object can make heavy use of these variable for inter-message communication and synchronization. Availability of the fetch and add primitive from the underlying architecture makes non-serializing creation of many copies of these variables possible. This makes possible a completely parallel processing of concurrent messages. This is described further in chapter 5.

ALLOY provides in effect both coarse and fine grain parallelism. The large number of light weight processes becomes of particular importance in a shared memory multiprocessor where fine grain processes can be implemented more efficiently.

Data Flow Architectures

It has been brought to our attention recently, that ALLOY may have efficient implementations on Data Flow architectures. This would be because ALLOY creates very fine grain parallelism, has only single assignment variables, and is functional enough (clear flow of data) to support lazy evaluation directly.

8.4.2 Distributed Architectures

The synchronizing value has a clear directionality, which can simplify efficient implementation on a distributed system compared to that of a system based on the logical variable.

Object Oriented programming helps program effectively distributed systems. ALLOY additionally exploits this by encouraging the user to write small objects. This is a side-effect of the dynamic variables in objects. Since different messages to the same object are processed by different processes spawned by the object itself, the object may be able to

utilize CPUs distributed into clusters.

The replicator can also help detect coarse grain parallelism for distribution to different CPUs. This is because replication is a natural way to express similar (often complex) processing of many data.

8.5 Summary

A full implementation of an ALLOY interpreter has been written in Common Lisp. Together with a programming environment and a set of libraries this system is written in less that 5000 lines of code.

This interpreter is focused on simplicity and flexibility and not on efficiency. It evaluates an average of about 60 function calls for each million of (VAX 11/780 equivalent) CPU instructions executed. Lazy evaluation is half as fast and generators a third as fast.

This interpreter is a significant step in the direction of an efficient implementation on a serial computer by improving understanding of the complexities and opportunities involved. It indicates some of the possibilities for optimization and efficient exploitation of either distributed or shared memory MIMD architectures.

Chapter 9

Conclusion

ALLOY, has been shown to be a simple yet powerful programming language. It is simple enough to have a kernel of only 29 primitives and expressions without restrictions in their usage. The set of primitives is more powerful than those of Lisp, Multilisp, Icon, Parlog, Vulcan, Actors, Linda-Lisp, and in some cases Prolog.

ALLOY closures provide full or-parallelism. Evaluating modes support serial or and-parallel execution, eager or lazy evaluation, non-determinism or multiple solutions. These modes can be combined freely.

There is no conflict between the implementation of the various features of ALLOY. Many special uses of features can easily be identified and be compiled very efficiently. In the worst case, a non-optimizing compiler could provide each feature with an efficiency within a constant factor of the best possible implementation of that feature on a language.

Parallel implementations of ALLOY can exploit massively parallel architectures. There are features of this languages which make good use of shared memory MIMD systems and features which make good use of distributed memory MIMD systems. This makes it possible for the programmer to write programs in a way suitable to the underlying architecture.

ALLOY owes its strength in a number of simple, clean, and efficient features. During their selection the power of their combinations was constantly considered. These features can be included in other programming languages.

A full implementation of ALLOY is given in Common Lisp. The implementation consists of an interpreter, libraries and some programming environment. While there has

been no attempt at efficiency, this interpreter when run in AKCL it performs about 60 function calls per million of (VAX equivalent) CPU instructions. The full system takes less that 5000 lines of code. An compiler in C is expected to be 20-100 times faster depending on the degree of optimizations.

Appendix A

The Interpreter of ALLOY in Common Lisp

This appendix lists the interpreter of ALLOY described previously. It has been implemented in Common Lisp and tested with Golden Common Lisp and Austin Kyoto Common Lisp compilers.

A.1 Top level

A.1.1 alloy.l

The root file loads the whole system in, except of the higher level libraries which are written in ALLOY itself:

```
;-------------------------------------------------------------------------------
; ALLOY interpreter.
; By Thanasis Mitsolides                (mitsolid@cs.nyu.edu)
; Written in Common Lisp
; Tested in Austin Kyoto Common Lisp and Alegro Common Lisp.
;-------------------------------------------------------------------------------
; Features:
; - Serial/parallel execution.
; - Eager/Lazy evaluation.
; - Flexible commitment mechanism.
; - Static scoping. Nested procedures.
; - Higher order functions.
; - Generators. Replicators.
; - Object Oriented Programming.
; - Parallel ADTs.
; - Delegation.
; - SIMPLICITY!
;-------------------------------------------------------------------------------
; Speed
;       - On a Sparcstation 1 (12 VAX MIPS) using Austin-Kyoto Common Lisp
;       - An average of 700 function calls per second.
;       - A minimum of 500 fcps.
;       - A maximum of 1200 fcps.
```

```
24  ;-------------------------------------------------------------------------------
25  ; Tested in Austin Kyoto Common Lisp and Alegro Common Lisp.
26  ;-------------------------------------------------------------------------------
27
28  ; Main file. Loads everything else.
29  ;-------------------------------------------------------------------------------
30
31  (proclaim '(optimize (speed 3) (safety 0) (space 0)))
32
33  (load "util.l")
34  (load "alutil.l")
35  (load "prep.l")
36  (load "gprep.l")
37  (load "oprep.l")
38  (load "load.l")
39  (load "values.l")
40  (load "io.l")
41  (load "inter.l")
42  (load "fcall.l")
43  (load "gcall.l")
44  (load "rcall.l")
45  (load "ocall.l")
46  (load "build.l")
47  (load "build2.l")
48  (load "obuild.l")
49
50  (feval '(load "alloylib.a"))
51
52  ; the next command may need to be given again from the terminal.
53  (alloy)
54
55
```

A.1.2 load.l

Top level and loader. Handles prompting and printing of results:

```
;------------------------------------------------------------------------------
; The begin and the end.
; Outermost Loop.

(defvar exit-alloy-loop nil "Alloy loop exits as soon as possible.")
(defvar exit-alloy-lisp nil "Lisp and Alloy loop exits as soon as possible.")
(defvar al-trace        nil "Alloy execution is traced.")
(defvar al-time-prompt  nil "Time of execiton is printed on every prompt.")
(defvar al-stats-prompt nil "Statistics are printed on Prompt.")
(defvar class-define    nil "When a class is being defined, holds its name.")
(defvar warning-exec    t   "Gives warnings in some situations during exec.")
(defvar warning-load    t   "Gives warnings in some situations during load.")
(defvar create-vars     t   "Allows creation of new variables.")

(defun alloy()
  (init-structs)
  (setq al-trace nil)
  (setq tab global-tab)
  (setq class-define nil)
  (setq al-readers 0)
  (setq exit-alloy-loop nil) (gcl-terpri)
  (princ "ALLOY version 2.0 5/21/91.") (terpri) (terpri)
  (prog (e)
    start
    (when exit-alloy-loop (go exit))
    (princ "ALLOY > ")  (force-output)
    (setq e (al-read)) (gcl-terpri)
    (when (eq e al-eof) (go exit))
    (setq e (al-prep e)) (when (eq e al-f)   (go start))
    (when al-trace (princ "--G-> ") (al-print e) (terpri) (write e) (terpri))
    (if al-time-prompt (time (feval e al-trace))
      (feval e al-trace))
    (when al-stats-prompt (pstatistics) (terpri))
    (go start)
    exit
    (princ "Bye.")
    (when exit-alloy-lisp (exit)))
  t)

(defvar gcl-p t "True if gcl is used.")

(defun non-interactive()
  (setq gcl-p nil))

;-----
; Comment next command if system replies before a <cr> is given.

(non-interactive)

(defun gcl-terpri(&optional stream)
  (when gcl-p
    (unless stream (setq stream *standard-output*))
```

```
53      (unless (streamp stream) (al-error "Stream expected in: ~A" stream))
54      (terpri stream)))
55
56  (defun pure-list(e)
57    (if (slist-p e) (slist-expr e) e))
58
59
60  ;-------------------------------------------------------------------------------
61  ; Loader.
62
63  (defun al-load(fname out)
64    (setq fname (sformat "~A" fname))
65    (let ((f (my-open-read-file fname)))
66      (if (not f)
67          (prog1 al-f (p-error "Cannot open file for reading: ~A" fname))
68        (prog1 t (al-load-file f out)))))
69
70  (defun al-load-file(f out)
71    (prog (e)
72      start
73      (force-output)
74      (setq e (al-read f)) (when (eq e al-eof) (go exit))
75      (setq e (al-prep e)) (when (eq e al-f)   (go start))
76      (when al-trace (princ "--L-> ") (al-print e) (terpri) (write e) (terpri))
77      (feval (if out e (list '$a$no-result e)) al-trace)
78      (go start)
79      exit
80      (close f)))
```

A.1.3 inter.l

The main interpreter loop, fundamental structures and initialization:

```
;-------------------------------------------------------------------------------
; ALLOY Interpreter.
; Data Structures, Initialization, Read-eval-execute Loop, Tracing.

;-------------------------------------------------------------------------------
; Data Structures.

(proclaim '(inline make-alval))
(proclaim '(inline make-proc))
(proclaim '(inline make-muinfo))

;_____
; All first class functions (i.e. not expressions) are represented as:

(defstruct (ext-function)
   (type   'al-error)
   (expr   "No function in ext-function structure.")
   (lmu    p-lmu))

;_____
; Synchronizing variables are represented as:

(defstruct (alval)
  "ALLOY logical value"
  (init      nil)
  (suspended nil)
  (value     nil)
  (delayed   nil))

;_____
; A process is represented as:

(defstruct (proc)
  (call      p-call)
  (result    p-result)
  (lmu       p-lmu)
  (waitfor   p-waitfor)
  (suspended p-suspended)
  (lazy      (if p-lazy lazyd p-lazy)))

;_____
; The environment for a process is represented as:

(defstruct (muinfo)
  (gmu       dumb-muinfo)
  (pmu       dumb-gmu)
  (chmus     nil)
  (lenv      nil)
  (die       nil)
  (commit    nil)
  (result    dumb-alval)
  (suspended dumb-proc)
```

```
53    (lazy      nil)
54    (top       nil)                        ; At top level.
55    (object    (muinfo-object p-lmu)))     ; Copy object from last mu
56
57  ;_____
58  ; Declaring a process to be the child of a parent environment.
59
60  (defun make-child(ch mu)
61    (setf (muinfo-pmu ch) mu)
62    (unless (muinfo-top mu) (push ch (muinfo-chmus mu))))
63
64  ;_____
65  ; Holds global values. They can be changed by anyone.
66
67  (defstruct (gval)
68    (val nil))
69
70  ;_____
71  ; fast structures.
72
73  (proclaim '(inline make-alval))
74  (proclaim '(inline make-proc))
75  (proclaim '(inline make-muinfo))
76
77  ;-------------------------------------------------------------------------------
78  ; Declarations. Top level variables.
79
80  (defvar al-cons     nil "Used by fcall")
81  (defvar al-scons    nil "Used by fcall")
82  (defvar proc-stack  nil "The stack of processes.")
83  (defvar proc-queue  nil "The queue of processes.")
84  (defvar proc        nil "Current process.")
85  (defvar tab         nil "Current hashing table. Current global environment.")
86  (defvar global-tab  nil "Holds the global table during class definitions.")
87  (defvar globale-tab nil "Holds the global exported table.")
88  (defvar spec-table  nil "Table of special expressions and their code.")
89  (defvar dumb-proc   nil "Dumb process.")
90  (defvar dumb-alval  nil "Dumb alloy variable.")
91  (defvar dumb-gmu    nil "Dumb global environment.")
92  (defvar dumb-muinfo nil "Dumb local environment.")
93  (defvar global-class  nil "Current global class.")
94  (defvar global-object nil "Current global object.")
95
96  (defvar gtq-max-org 1000
97                      "Original value.")
98  (defvar gtq-max gtq-max-org
99                  "Move a process from the stack to the queue every steps.")
100 (defvar gtq-cycle gtq-max "Do the move when counter becomes negative.")
101
102 (proclaim '(type (or t nil) p-same))
103 (defvar p-same nil "Execute the same proc structure again.")
104
105 (proclaim '(type fixnum     p-waitfor))
106 (proclaim '(type muinfo     p-lmu))
107 (proclaim '(type (or t nil) p-serial))
```

```
108  (proclaim '(type alval      p-result))
109  (proclaim '(type cons       p-call))
110
111  (proclaim '(type proc       dumb-proc))
112  (proclaim '(type muinfo     dumb-muinfo))
113  (proclaim '(type alval      dumb-alval))
114  (proclaim '(type muinfo     dumb-gmu))
115  (proclaim '(type class      global-class))
116  (proclaim '(type object     global-object))
117
118  (defvar p-serial        nil     "Serial-parallel     part of current process")
119  (defvar p-call          nil     "Function Call       part of current process")
120  (defvar p-result        nil     "Result variable     part of current process")
121  (defvar p-lmu           nil     "Local mu closure    part of current process")
122  (defvar p-waitfor       nil     "Waiting for process part of current process")
123  (defvar p-suspended     nil     "Suspended process   part of current process")
124  (defvar p-lazy          nil     "Lazy not eager      part of current process")
125
126  ;-------------------------------------------------------------------------------
127  ; Main loop
128
129  (defun feval0(&rest args)
130     (init-structs)
131     (apply 'al-eval args))
132
133  (defun feval(&rest args)
134     (reset-stats)
135     (apply 'al-eval args))
136
137  (defun al-eval(start-expr &optional trace)
138    (let ((proc (al-eval3 start-expr
139                          (if class-define (nth 2 class-define) dumb-gmu))))
140      (push proc proc-stack)
141      (al-eval-loop trace)
142      t))
143
144  (defun top-level-proc(start-expr)
145    (al-eval4 start-expr dumb-gmu))
146
147  (defun al-eval3(start-expr gmu)
148    (al-eval4 (make-slist :expr (list '$a$finish-eval start-expr)) gmu))
149
150  (defun al-eval4(expr gmu)
151    (let* ((d-mu   gmu)
152           (d-proc (make-proc-m :lazy nil :waitfor 100))
153           (proc   (make-proc-m :call expr
154                                :result (make-alval-m)
155                                :lmu d-mu
156                                :suspended d-proc
157                                :lazy nil
158                                :waitfor 0)))
159      proc))
160
161  (defvar cycles-nosx 0)
162  (defvar proc-nosx   0)
```

```
163  (defvar alval-nosx  0)
164  (defvar muinfo-nosx 0)
165  (defvar funcall-nosx 0)
166  (defvar specall-nosx 0)
167
168  (defun al-eval-loop(trace)
169     (prog ()
170           start
171           (incf cycles-nosx)
172
173           (when (minusp (decf gtq-cycle))
174              (setq gtq-cycle gtq-max)
175              (when proc-stack (queue-put (pop proc-stack) proc-queue)))
176
177           (cond (p-same (setq p-same nil))
178                 (proc-stack                 (setq proc (pop proc-stack))
179                                             (expand-proc))
180                 ((queue-nonempty proc-queue) (setq proc (queue-get proc-queue))
181                                             (expand-proc))
182                 (t (go exit)))
183           (when trace (pstatus))
184           (when (muinfo-die p-lmu) (go start))
185           (when (lazy-delay1-p p-lazy)
186              (setq p-lazy lazy1) (delay-me-expanded) (go start))
187           (when (lazy-delay-p p-lazy)
188              (setq p-lazy lazyt) (delay-me-expanded) (go start))
189           (unless (spec-eval)
190              (if p-serial (do-scall) (do-pcall)))
191           (go start)
192           exit))
193
194  (defun expand-proc()
195     (let ((c (proc-call proc)))
196        (cond ((slist-p c) (setq p-serial t)   (setq p-call (slist-expr c)))
197              (t           (setq p-serial nil) (setq p-call c))))
198     (setq p-result    (proc-result    proc))
199     (setq p-lmu       (proc-lmu       proc))
200     (setq p-waitfor   (proc-waitfor   proc))
201     (setq p-lazy      (proc-lazy      proc))
202     (setq p-suspended (proc-suspended proc))
203     (setq tab (class-tab (object-class (muinfo-object p-lmu)))))
204
205  ;_____
206  ; Reconstruct process taking serial parallel info from previous call.
207
208  (defun construct-proc()
209    (setq proc (let ((c (if p-serial (make-slist :expr p-call) p-call)))
210                 (make-proc-m :call c                :result p-result
211                              :lmu p-lmu             :waitfor p-waitfor
212                              :suspended p-suspended :lazy p-lazy))))
213
214  ;_____
215  ; Reconstruct process taking serial parallel info from current call.
216
217  (defun construct-procc()
```

```
218   (check-p-serial)
219   (construct-proc))
220
221 (defun check-p-serial()
222   (cond ((slist-p p-call) (setq p-serial t) (setq p-call (slist-expr p-call)))
223         (t                (setq p-serial nil))))
224
225 (defun p-ser-to-p-call()
226   (when p-serial (setq p-serial nil ) (setq p-call (make-slist-t p-call t))))
227
228
229 ;-------------------------------------------------------------------------------
230 ; Initialization functions.
231 ; Set up environment.
232
233 (defun init()
234   (init-structs)
235   (setq *print-circle* t)
236   (setq *print-length* 14)
237   (setq *print-level* 6)
238   (setq spec-table (make-sym-table))
239   (setq tab (setq global-tab (make-sym-table)))
240   (setq globale-tab (make-sym-table))
241   (setq dumb-alval (make-alval))
242
243   (setq global-class
244         (make-class :origdef '((name . global) (static)(dynamic)(common)
245                                (inherit)(import)(methods))
246                     :tab global-tab :ndynnames `(0 () ())))
247   (setq global-object (make-object :class global-class :createvar t))
248
249   (setq dumb-gmu                (def-dumb-gmu))
250   (setf (muinfo-gmu    dumb-gmu) dumb-gmu)
251   (setf (muinfo-pmu    dumb-gmu) dumb-gmu)
252
253   (setq dumb-proc  (make-proc-m :lazy nil :waitfor 9999999
254                                 :result dumb-alval :suspended dumb-proc
255                                 :call '( (quote toptop) )   :lmu dumb-gmu))
256
257   (setq p-lmu       dumb-gmu)
258   (setq dumb-muinfo dumb-gmu))
259
260 (defun def-dumb-gmu()
261   (make-muinfo-m :commit t   :suspended dumb-proc   :top t
262                  :chmus  nil :result dumb-alval :die nil     :object global-object
263                  :lenv `(((original-msg alloy-loop ,global-object)
264                           (here ,@ global-object) (self ,@global-object)))))
265
266 (defun init-structs()
267   (reset-stats)
268   (setq gtq-cycle  gtq-max)
269   (setq proc-stack nil)
270   (setq proc-queue (make-queue))
271   (setq proc nil))
272
```

```
273  (defun reset-stats()
274    (setq cycles-nosx 0)
275    (setq muinfo-nosx 0)
276    (setq alval-nosx  0)
277    (setq proc-nosx   0)
278    (setq funcall-nosx 0)
279    (setq specall-nosx 0)
280  )
281
282  ;-------------------------------------------------------------------------------
283  ; Tracing Functions.
284
285  (defun pstatistics()
286    (pprint (list (cons "Cycles  : " cycles-nosx)
287                  (cons "Proc    : " proc-nosx)
288                  (cons "Muinfo  : " muinfo-nosx)
289                  (cons "Alval   : " alval-nosx)
290                  (cons "Funcall : " funcall-nosx)
291                  (cons "Specallo: " (- specall-nosx funcall-nosx))
292                  (cons "FScallt : " specall-nosx))))
293
294  (defun pstatus()
295    (print-reg p-call)
296    (princ
 "================================================================================")
297    (terpri))
298
299  (defmacro print-reg(reg)
300    `(let()
301       (princ (quote ,reg)) (princ ":")
302       (princ "------------------------------------------------------------------")
303       (terpri) (write ,reg :length 8 :level 4 :pretty t :circle t)  (terpri)))
304
```

A.2 Main Interpreter

A.2.1 fcall.l

Interpretation of function calls serial and parallel. Commitment and Lazy evaluation are handled here. Basic expressions are also executed here.

```
;-------------------------------------------------------------------------------
; Main interpretation of calls.
; Interprets closure applications.
; Interprets Basic Special expressions.
; Delegates interpretation of build in functions and messages.

;-------------------------------------------------------------------------------
; Make parallel call a set of processes. Process is in "proc".

(defun do-pcall()
   (let ((g-ci 0)
         (g-proc (make-naddproc nil p-result p-lmu 0 p-suspended))
         (g-calls nil))
     (push g-proc proc-stack)
     (setq p-call  (nreverse (rm-dot-list p-call)))
     (setq g-calls (do ((stk nil) expr)
                       ((null p-call) stk)
                     (setq expr (pop p-call))
                     (push (valprocp) stk)))
     (incf (proc-waitfor p-suspended) g-ci)
     (nput-dot-list-nc g-calls)
     (setf (proc-call g-proc) (cons '$a$funcall-vs g-calls))))

(defmacro valprocp()
  '(cond ((alcall-p expr) (let ((varres (make-alval-m)))
                            (incf g-ci)
                            (make-addproc expr varres p-lmu 0 p-suspended)
                            varres))
         ((get-alsym-value expr))))

(defmacro make-addproc(expr result lmu waitfor suspended)
   ` (push (make-proc-m :call ,expr :result ,result :lmu ,lmu
                        :waitfor ,waitfor :suspended ,suspended)
                  proc-stack))

(defun make-pprocess(e)
   (make-addproc (to-al-call e) (make-alval-m)
                 p-lmu 0 p-suspended)
   (incf (proc-waitfor p-suspended))
   (car proc-stack))

(defun to-al-call(e)
   (if (alcall-p e) e (list '$a$pass1 e)))

;-------------------------------------------------------------------------------
; Make serial call a set of processes. Process is in "proc".

(defun do-scall()
   (let ((g-fc    (make-naddproc nil p-result p-lmu 1 p-suspended))
```

```
50          (g-calls nil))
51      (setq p-call  (nreverse (rm-dot-list p-call)))
52      (setq g-calls (do ((stk nil) expr (prev g-fc))
53                        ((null p-call) (setf (proc-waitfor prev) 0)
54                                       (push prev proc-stack) stk)
55                      (setq expr (pop p-call))
56                      (push (valprocs) stk)))
57      (nput-dot-list-nc g-calls)
58      (setf (proc-call g-fc) (make-slist :expr (cons '$a$funcall-vs g-calls)))))
59
60
61  (defmacro valprocs()
62    '(cond ((alcall-p expr) (let ((varres (make-alval-m)))
63                              (setq prev (make-naddproc expr varres p-lmu 1 prev))
64                              varres))
65           ((get-alsym-value expr))))
66
67  (defmacro make-naddproc(expr result lmu waitfor suspended)
68     ` (make-proc-m :call ,expr :result ,result :lmu ,lmu
69                    :waitfor ,waitfor :suspended ,suspended) )
70
71  ;-----------------------------------------------------------------------------
72  ; Apply a function after its arguments have been computed.
73
74  (defvar alfvs-f nil  "Function name of current external function.")
75  (defvar alfvs-mu nil "When t mus are not spawned but given to alfvs-mu.")
76
77  (defun al-funcall-vs()
78    (setq alfvs-mu nil)
79    (al-funcall-vs-main))
80
81  ;_____
82  ; Executes call unless it results in a closure application (associated mu).
83  ; In the last case returns the created closure (mu), otherwise atom nomu.
84  ; The actual result is given to the first argument to al-funcall-nomu.
85
86  (defun al-funcall-nomu()
87    (setq alfvs-mu p-result)
88    (setq p-result (cadr p-call))
89    (when (slist-p (cddr p-call)) (setq p-serial t))
90    (setq p-call (cons (car p-call) (rm-slist (cddr p-call))))
91    (al-funcall-vs-main)
92    (when (alval-p alfvs-mu) (alset-alval 'nomu alfvs-mu)))
93
94  (defun al-funcall-vs-main()
95     (let* ((args (cdr p-call))
96            (f    (strip-alval (pop args))))
97       (setq args (strip-alval args))
98       (cond ((alval-p f) (waitfor-val f))
99             ((eq f al-f) (argone 'cl-failfunc nil))
100            ((ext-function-p f)
101             (incf funcall-nosx)
102             (setq alfvs-f f)
103             (funcall (ext-function-type f) (ext-function-expr f) args))
104            ((and (eql f 'eval) (alval-p args)) (waitfor-val args))
```

```
105             ((and (or (eql f 'eval) (eql f 'new)) (cons-p args)
106                   (setq args (cons (setf (car args) (strip-alval (car args)))
107                                    (cdr args)))
108                   (alval-p (car args)))
109              (waitfor-val (car args)))
110             ((and (eql f 'eval) (cons-p args) (ext-function-p (car args)))
111              (setq p-call (cons (car p-call) args))
112              (al-funcall-vs-main))
113             ((and (eql f 'new) (cons-p args) (round-cons-p (car args)))
114              (setq p-call (cons (car p-call) (cons al-cons (cdr args))))
115              (al-funcall-vs-main))
116             ((and (eql f 'new) (cons-p args) (square-cons-p (car args)))
117              (setq p-call (cons (car p-call) (cons al-scons (cdr args))))
118              (al-funcall-vs-main))
119             ((symbol-p f) (message-other f args))
120             ((class-p f)
121              (setq p-call (cons (car p-call) (cons 'new (cdr p-call))))
122              (al-funcall-vs-main))
123             (t
124              (setq p-call (cons (car p-call) (cons 'eval (cdr p-call))))
125              (al-funcall-vs-main))
126             (t (when warning-exec
127                      (p-warning "Function or message expected in: ~A" f))
128                (argone 'cl-failfunc nil)))))
129
130  (defun cl-failfunc(&rest dumb)     (declare (ignore dumb))
131    (cons 'suc al-f))
132
133  (defun pfunction(cl-fun)
134     (funcall cl-fun))
135
136  (defun gfunction(args-body vals)
137    (let ((current-gmu (select-lmu alfvs-f)))
138      (lfunction-lmu args-body vals current-gmu t)))
139
140  (defun gbfunction(args-body vals)
141    (let ((current-gmu (select-lmu alfvs-f)))
142      (lfunction-lmu args-body vals current-gmu t)))
143
144  (defun select-lmu(f)
145    (let ((lmu (ext-function-lmu f)))
146      (if (eq (object-class (muinfo-object p-lmu))
147              (object-class (muinfo-object lmu)))
148          p-lmu lmu)))
149
150  (defun lfunction(args-body vals)
151    (lfunction-lmu args-body vals (ext-function-lmu alfvs-f) nil))
152
153  ;_____
154  ; Creates a process from the application of a closure.
155
156  (defun lfunction-lmu(args-body vals lmu globalp)
157     (let ((body (cond ((consp   args-body) (body-clist nil (cdr args-body)))
158                       ((slist-p args-body) (body-clist t   (cdr (slist-expr
 args-body))))
```

```
159                     ((al-error "Unexpected function definition: ~A" args-body))))
160          (args (xcar args-body)))
161       (let* ((call-env (new-env args vals (muinfo-lenv lmu)))
162              (mu    (make-muinfo-m :gmu       (muinfo-gmu lmu)
163                                    :lenv      (cdr call-env)
164                                    :result    p-result :lazy p-lazy
165                                    :suspended p-suspended
166                                    :object (muinfo-object lmu)))
167              (final (make-proc-m :call (list '$a$alfun-fail)
168                                  :lmu mu :waitfor 1
169                                  :lazy nil))
170              (proc2 (make-proc-m :call body
171                                  :result (make-alval-m)
172                                  :lmu mu    :suspended final :lazy nil
173                                  :waitfor 0)))
174         (make-child mu p-lmu)
175         (when globalp (setf (muinfo-gmu mu) mu))
176
177         (when (car call-env) (setq proc2 (lfl-carg (car call-env) proc2)))
178
179         (if alfvs-mu (progn (alset-alval proc2 alfvs-mu) (setq alfvs-mu nil))
180           (push proc2 proc-stack)))))
181
182  (defun lfl-carg(call proc2)
183    (let ((proga (make-proc-m :call call
184                              :result dumb-alval :lazy nil
185                              :waitfor 0 :suspended dumb-proc)))
186      (push proga proc-stack)
187
188      (when p-serial
189            (if alfvs-mu
190                (let* ((vproc (make-alval))
191                       (prest (make-proc-m :call `($a$pass-v ,proc2)
192                                           :result vproc
193                                           :lazy nil
194                                           :waitfor 1 :suspended dumb-proc)))
195                  (setf (proc-suspended proga) prest) (setq proc2 vproc))
196              (progn (setf (proc-suspended proga) proc2) (incf (proc-waitfor proc2) 1)
197                     (pop proc-stack) (setq proc2 proga)))))
198    proc2)
199
200  (defun body-clist(serp body)
201    (if (and (cons-p body) (null (cdr body)) (alcall-p (car body))) (car body)
202      (make-slist-t (cons 'list body) serp)))
203
204  ;_____
205  ; Build in function call which accepts all args in a list.
206
207  (defun argone(cl-func args)
208     (let ((v (funcall cl-func args)))
209       (cond ((eq (car v) 'suc) (give-value  (cdr v)))
210             ((eq (car v) 'sus) (waitfor-val (cdr v)))
211             ((al-error "argone: Succeed or suspend cons expected: ~A" v)))))
212
213  ;_____
```

```
214  ; Build in function call which accepts all args separately.
215
216  (defun noninst(cl-func args)
217     (let ((v (apply cl-func args)))
218       (cond ((eq (car v) 'suc) (give-value  (cdr v)))
219             ((eq (car v) 'sus) (waitfor-val (cdr v)))
220             ((al-error "noninst: Succeed or suspend cons expected: ~A" v)))))
221
222
223  ;-----------------------------------------------------------------------------
224  ; Operations on ALLOY values variables.
225
226  ;_____
227  ; If sym is a symbol return the value given to it else return it (value).
228
229  (defun get-alsym-value(sym)
230     (cond ((null sym) sym)
231           ((symbolp sym)
232            (strip-alval (prog2 (check-ncvars)
233                                (get-val sym (muinfo-lenv p-lmu) tab globale-tab)
234                                (check-cvars))))
235           (sym)))
236
237  ;_____
238  ; During evaluation of a special expression.
239  ; Sleep waiting for alval to be set.
240
241  (defun waitfor-val(alval)
242    (wake-up-delayed alval)
243    (construct-proc)
244    (push proc (alval-suspended alval)))
245
246  (defun wake-up-delayed(alval)
247    (when (alval-delayed alval)
248      (wakeups (alval-delayed alval)) (setf (alval-delayed alval) nil)))
249
250  ;_____
251  ; During evaluation of a special expression.
252  ; Return as a result val.
253
254  (defun give-value(val)
255     (alset-alval val p-result)
256     (check-suspended))
257
258  ;_____
259  ; Set alloy value alval to value val.
260
261  (defun alset-alval(val alval)   ; value variable
262     (unless (or (not (alval-p alval)) (alval-init alval))
263        (setq val (strip-alval val))
264        (setf (alval-init  alval) t)
265        (setf (alval-value alval) val)
266        (let ((susp1 (alval-suspended alval)))
267           (setf (alval-suspended alval) nil)
268           (if (alval-p val)
```

```
269              (when susp1 (push susp1 (alval-suspended val)) (wake-up-delayed val))
270            (wakeups susp1)) t)))
271
272  (defun wakeups(l)
273     (cond ((null  l))
274           ((consp l) (wakeups (car l)) (wakeups (cdr l)))
275           ((push l proc-stack))))
276
277  ;_____
278  ; Inform waiting process (continuation) that current process has finished.
279  ; If waiting process is ready to execute make it ready.
280
281  (defun check-suspended()
282     (check-suspended1 p-suspended))
283
284  (defun check-suspended1(proc2)
285     (when (zerop (decf (proc-waitfor proc2)))
286        (push proc2 proc-stack)))
287
288
289  ;-------------------------------------------------------------------------------
290  ; Evaluation of Special Expressions.
291  ; Treated as macros.
292
293  (defun spec-eval()
294     (let ((fun-name (get-spec-value (car p-call))))
295        (when fun-name
296           (incf specall-nosx)
297           (funcall (car fun-name)) t)))
298
299  (defun spec-quote()
300     (let ((qarg (cdr p-call)))
301        (cond ((consp qarg) (give-value (car qarg)))
302              ((null  qarg) (give-value qarg))
303              (t (p-error "Quote has dotted argument.")))))
304
305  (defun spec-$a$pass-v()
306     (let ((arg (cadr p-call)))
307        (give-value arg)))
308
309  (defun spec-lazy()
310    (setq p-lazy lazyd)
311    (setq p-call (cadr p-call))
312    (check-proc-call)
313    (next-the-same))
314
315  (defun spec-delay()
316    (setq p-call (cadr p-call))
317    (check-proc-call)
318    (delay-me))
319
320  (defun spec-eager()
321    (setq p-call (cadr p-call))
322    (setq p-lazy nil)
323    (check-proc-call)
```

```
324    (next-the-same))
325
326  (defun delay-me()
327    (if (alval-suspended p-result)
328        (next-the-same)
329      (push (construct-procc) (alval-delayed p-result))))
330
331  (defun delay-me-expanded()
332    (p-ser-to-p-call)
333    (delay-me))
334
335  (defun spec-mu()
336     (let ((mu-call (make-slist-t (cdr p-call) p-serial)))
337       (give-value (if (muinfo-top p-lmu)         ; Global level.
338                       (make-gfunction mu-call)
339                     (make-lfunction mu-call)))))
340
341   (defun spec-function()
342     (setq p-call (list () (cons 'return (cdr p-call))))
343     (spec-mu))
344
345  (defun spec-$a$set-v1()
346     (let ((var-val (cadr   p-call)))
347        (setq p-call (caddr p-call))
348        (unless (alset-alval p-result var-val)
349           (alset-alval al-f p-result))
350        (check-proc-call)
351        (next-the-same)))
352
353  (defun spec-set()
354    (spec-set-forse nil))
355
356  (defun spec-when()
357     (let ((condition (cadr p-call)))
358        (cond ((alcall-p condition)
359               (spec-xcond-wait '$a$when-v condition))
360              ((let ((c (get-alsym-value condition)))
361                  (setq p-call (cons '$a$when-v (cons c (cddr p-call))))
362                  (spec-$a$when-v))))))
363
364  (defun spec-xcond-wait(code condition)
365    (if p-serial
366        (let ((alval (make-alval-m)))
367          (setq p-call (cons code (cons alval (cddr p-call))))
368          (incf p-waitfor)
369          (construct-proc)
370          (make-addproc condition alval p-lmu 0 proc))
371      (let ((alval (proc-result (make-pprocess condition))))
372        (setq p-call (cons code (cons alval (cddr p-call))))
373        (waitfor-val alval))))
374
375  (defun spec-if-wait(alval)
376     (setq p-call (cons '$a$if-v (cons alval (cddr p-call))))
377     (waitfor-val alval))
378
```

```
379 (defun spec-$a$when-v()
380    (let* ((args (strip-alval (cdr p-call)))
381           (c    (strip-alval (pop args))))
382       (cond ((alval-p c) (waitfor-val c))
383             ((eq c al-f) (argone 'cl-failfunc nil))
384             ((null args) (give-value c))
385             (t
386              (setq p-call
387                    (make-slist-t
388                     (cons 'blockn (if (rlist-p args)
389                                       (cons `($a$pass-v ,c) args)
390                                     args))
391                           p-serial))
392              (next-the-same)))))
393
394 (defun spec-if()
395    (let ((condition (cadr p-call)))
396       (cond ((alcall-p condition)
397              (spec-xcond-wait '$a$if-v condition))
398             ((let ((c (get-alsym-value condition)))
399                 (setq p-call (cons '$a$if-v (cons c (cddr p-call))))
400                 (spec-$a$if-v))))))
401
402 (defun spec-$a$if-v()
403    (let* ((args (cdr p-call))
404           (c    (strip-alval (pop args))))
405       (cond ((alval-p c) (waitfor-val c))
406             ((and (eq c al-f) (null (cdr args)))
407              (give-value c))
408             ((eq c al-f)
409              (setq p-call (cadr args))
410              (check-proc-call)
411              (next-the-same))
412             (t
413              (setq p-call (car args))
414              (check-proc-call)
415              (next-the-same)))))
416
417
418 ;-----------------------------------------------------------------------------
419 ; Commitment mechanisms. Return and fail. Local and global.
420
421 (defun spec-return()
422    (spec-lreturn-lmu (muinfo-gmu p-lmu) t))
423
424 (defun spec-lreturn()
425    (spec-lreturn-lmu p-lmu nil))
426
427 (defun spec-lreturn-lmu(lmu globalp)
428   (let ((r (muinfo-result lmu)))
429     (spec-lr-lmus (if (muinfo-p r) r lmu) lmu globalp)))
430
431 (defun spec-lr-lmus(comgen-lmu lmu globalp)
432   (cond ((muinfo-commit comgen-lmu)
433          (if globalp
```

```
434              (p-error "return statement executed after commitment.")
435            (p-error "lreturn statement executed after commitment."))
436          (give-value al-f))
437         ((check-ser-return))
438         ((let ((expr (cadr p-call)))
439            (gen-or-commit comgen-lmu)
440            (cond ((alcall-p expr)
441                   (let ((com-mu (make-muinfo-m :gmu       (muinfo-gmu lmu)
442                                                :chmus     nil
443                                                :lenv      (muinfo-lenv p-lmu)
444                                                :die       nil
445                                                :commit    t
446                                                :result    dumb-alval
447                                                :suspended dumb-proc)))
448                     (make-child com-mu (muinfo-pmu lmu))
449                     (when globalp (setf (muinfo-gmu com-mu) com-mu))
450                     (setq p-lmu       com-mu)
451                     (setq p-call      expr)
452                     (setq p-lazy      (muinfo-lazy lmu))
453                     (next-the-same)))
454                  (t (give-value (get-alsym-value expr))))))))
455
456  (defun gen-or-commit(lmu)
457    (if (genstate-p (muinfo-result lmu))
458        (gen-lreturn-lmu lmu)
459      (progn (terminate-fun lmu)
460             (alset-alval p-result (muinfo-result lmu)))))
461
462  (defun terminate-fun(lmu)
463    (setq p-suspended (muinfo-suspended lmu))
464    (kill-mus lmu))
465
466  (defmacro check-ser-return()
467    '(when p-serial
468           (let* ((v    (make-alval))
469                  (prest (make-proc-m :call (list (if globalp 'return 'lreturn) v)
470                                      :result p-result :lazy nil
471                                      :lmu p-lmu :waitfor 1 :suspended p-suspended)))
472             (setq p-suspended prest) (setq p-result v)
473             (setq p-call (cadr p-call))
474             (check-proc-call)
475             (next-the-same))
476           t))
477
478  (defun alcall-p(e)
479     (or (consp e) (slist-p e)))
480
481  (defun kill-mus(mu)
482     (declare (type muinfo mu))
483     (setf (muinfo-commit mu) t)
484     (setf (muinfo-die    mu) t)
485     (let ((chmus (muinfo-chmus mu)))
486        (while chmus (kill-mus (pop chmus)))
487        (setf (muinfo-chmus mu) nil)))
488
```

```
489  (defun spec-$a$no-result()
490    (alset-alval '$a$no-result p-result)
491    (setq p-result (make-alval))
492    (setq p-call (cadr p-call))
493    (make-eager-self)
494    (check-proc-call)
495    (next-the-same))
496
497  (defun spec-alfun-fail()
498     (alset-alval al-f (muinfo-result    p-lmu))
499     (check-suspended1 (muinfo-suspended p-lmu)))
500
501  (defun spec-fail()
502    (spec-fail-lmu (muinfo-gmu p-lmu) t))
503
504  (defun spec-lfail()
505    (spec-fail-lmu p-lmu nil))
506
507  (defun spec-fail-lmu(lmu globalp)
508    (let ((r (muinfo-result lmu)))
509      (spec-f-lmus (if (muinfo-p r) r lmu) lmu globalp)))
510
511  (defun spec-f-lmus(comgen-lmu lmu globalp)    (declare (ignore lmu))
512    (cond ((muinfo-commit comgen-lmu)
513           (if globalp
514               (p-error "fail statement executed after commitment.")
515             (p-error "lfail statement executed after commitment.")))
516          (t
517           (terminate-fun comgen-lmu)
518           (setq p-lmu comgen-lmu)
519           (if (genstate-p (muinfo-result comgen-lmu))
520               (spec-alfun-gfail)
521             (spec-alfun-fail))))
522    (give-value al-f))
523
524  (defun make-eager-self()
525    (setq p-lazy nil))
526
527
528  ;-------------------------------------------------------------------------------
529  ; Utilities.
530
531  (defun next-the-same()
532    (check-p-serial)
533    (setq p-same t))
534
535  (defun check-proc-call()
536     (unless (alcall-p p-call)
537        (setq p-call (list '$a$pass1 p-call))))
538
```

A.2.2 ocall.l

Run time execution of objects and classes:

```
;-------------------------------------------------------------------------------
; Object Oriented Programming. Hooks to Interpreter.

;-------------------------------------------------------------------------------
; Actual code for special expression set. Depended on current class.

(defun spec-$a$setfun1()
  (let ((v (get-alsym-value (cadr p-call))))    ; Striped
    (cond ((alval-p v)
           (when warning-load (p-msg "Defining function  : ~A" (cadr p-call)))
           (setq p-call (cons '$a$reset1 (cdr p-call)))
           (next-the-same))
          ((and (ext-function-p v) (member (ext-function-type v)
                                           '(gfunction lfunction)))
           (if (get-val-env (cadr p-call) (muinfo-lenv p-lmu))
               (let()
                 (p-error "Cannot redefine local function: ~A" (cadr p-call))
                 (give-value al-f))
             (let()
               (when warning-load (p-msg "Redefining function: ~A" (cadr p-call)))
               (setq p-call (cons '$a$reset1 (cdr p-call)))
               (next-the-same))))
          (t
           (p-error "Can not define function. Symbol is not a user function: ~A"
                  (cadr p-call))
           (give-value al-f)))))

(defun spec-$a$reset1()
  (spec-set-forse t))

(defun check-ncvars()
  (unless (object-createvar (muinfo-object p-lmu)) (setq create-vars nil)))

(defun check-cvars()
  (setq create-vars t))

(defun spec-set-forse(forse)
  (check-ncvars)
  (let ((var-name (cadr p-call)))
    (setq p-call (caddr p-call))
    (cond (p-serial
           (let* ((v (make-alval-m))
                  (prest (make-proc-m :call `(set ,var-name ,v) :result p-result
                                      :lazy nil :lmu p-lmu
                                      :waitfor 1 :suspended p-suspended)))
             (setq p-suspended prest) (setq p-result v)))
          (t (unless (set-val var-name p-result (muinfo-lenv p-lmu) tab)
                     (if forse
                         (if (get-val-env var-name (muinfo-lenv p-lmu))
                             (let()
                               (p-error "Cannot redefine local variable: ~A"
                                        var-name)
```

```
53                              (alset-alval al-f p-result))
54                          (set-sym-value tab var-name p-result))
55                      (alset-alval al-f p-result))))))
56    (check-cvars)
57    (check-proc-call)
58    (next-the-same))
59
60
61  ;-------------------------------------------------------------------------------
62  ; Class defginition.
63
64  (defstruct (class)
65    (origdef      nil)            ; name and assoc list of definition in start
66    (tab          nil)            ; Symbol table of set/common inherited vars
67    (ndynnames    nil)            ; 1 #. 2 name. 3 next-names.
68    )
69
70  (defun class-name(class)
71     (get-feature 'name (class-origdef class)))
72
73  (defun object-name(object)
74     (get-feature 'name (class-origdef (object-class object))))
75
76  (defstruct (object)
77    (class        nil)                  ; The class
78    (interface    nil)                  ; List of current methods.
79    (inienv       nil)                  ; Multi alist list of static vars
80    (nextdyn      (make-gval :val nil))   ; List of next dynamic variables.
81    (createvar    nil)                  ; Can create static variables.
82    )
83
84  (defun spec-$a$class-start()
85    (let* ((defs (cdr p-call)) (cname (car defs)) (c (make-class)))
86      (setq defs (cadr defs))
87      (if (not (check-set-cdef cname)) (give-value al-f)
88        (let()
89          (let ((obj-gmu (make-muinfo-m :top t :commit t)))
90            (setf (muinfo-gmu obj-gmu) obj-gmu)
91            (make-child obj-gmu p-lmu)
92            (setf (muinfo-object obj-gmu) (make-object :class c :createvar t))
93            (setq class-define (list cname c obj-gmu)))
94          (setf (class-origdef   c) (cons (cons 'name cname) defs))
95          (setf (class-tab       c) (make-sym-table))
96          (let ((dyns (get-feature 'dynamic defs)))
97            (setf (class-ndynnames c)
98                  (list (length dyns) dyns (ren-list dyns 'next-))))
99          (set-sym-value tab cname c)
100         (set-sym-value tab (atom-app cname '-p)
101                        (make-gbfunction `((e) (return (eql ('object-p e)
102                                                            (quote ,cname))))))
103         (setq tab (class-tab c))
104         (set-sym-value tab 'new
105                        (make-gfunction '(() (return))))
106         (give-value c)))))
107
```

```
108  (defun check-set-cdef(cname)
109    (let ((v (get-alsym-value cname)))    ; Striped
110      (cond (class-define
111             (p-error "Nested class definition ignored: ~A" cname) nil)
112            ((alval-p v)
113             (when warning-load (p-msg "~%Defining class  : ~A" cname)) t)
114            ((class-p v)
115             (when warning-load (p-msg "~%Redefining class: ~A" cname)) t)
116            (t
117             (p-error "Can not define class. Symbol is not a user class: ~A" cname)
 nil))))
118
119  (defun ren-list(names prefix)
120    (cond ((null names) nil)
121          ((cons (atom-app prefix (car names)) (ren-list (cdr names) prefix)))))
122
123  (defun create-objs(mytab names prefix)
124    (cond ((null names))
125          (t
126           (let* ((name  (car names))
127                  (args  nil)
128                  (oname (atom-app prefix name)) (v (make-alval)))
129             (set-sym-value mytab oname v)
130             (make-pprocessv `('new ,(get-sym-value global-tab name) ,@args) v))
131           (create-objs mytab (cdr names) prefix))))
132
133  (defun spawn-obj-list(names prefix)
134    (when names
135      (let* ((name  (car names))
136             (args  nil)
137             (oname (atom-app prefix name)) (v (make-alval)))
138        (make-pprocessv `('new ,(get-sym-value global-tab name) ,@args) v)
139        (cons (cons oname v) (spawn-obj-list (cdr names) prefix)))))
140
141  (defun spec-$a$class-end()
142    (let ((cname (cadr p-call)))
143      (if (not (eq (car class-define) cname))
144          (if (car class-define)
145            (p-msg "ERROR: end of class definition expected: ~A" (car class-define))
146            (p-msg "ERROR: unexpected class conclusion: ~A" cname))
147        (let* ((c (cadr class-define)) (defs (class-origdef c)))
148          (check-methods (get-feature 'methods defs) tab)
149          (when warning-load (p-msg "Defined class: ~A" cname))
150          (setq class-define nil)
151          (create-objs tab (get-feature 'common  defs) 'c-)))
152      (give-value cname)))
153
154  (defun get-feature(name alist)
155    (cdr (assoc name alist)))
156
157  (defun check-methods(mlist tab)
158    (cond ((null mlist) nil)
159          (t
160           (let* ((m (car mlist)) (def (get-sym-value tab m)))
161             (unless (ext-function-p (strip-alval def))
```

```
162                   (p-warning "Undefined method: ~A" m)))
163          (check-methods (cdr mlist) tab))))
164
165  (defun link-vars(names to td)
166    (cond ((null names))
167          (t
168           (let ((name (car names)))
169             (set-sym-value td name (get-sym-value to name)))
170           (link-vars (cdr names) to td))))
171
172  ;-------------------------------------------------------------------------------
173  ; Actual processing of messages (sent to class or object).
174
175  (defun message-other(f args)
176    (setq args (strip-alval args))
177    (let ((obj nil))
178      (cond ((null args)
179             (p-error "incorrect message passing in: ~A" (cons f nil))
180             (argone 'cl-failfunc nil))
181            ((alval-p args)      (waitfor-val args))
182            ((not (listp args))
183             (p-error "incorrect message passing in: ~A" (cons f args))
184             (argone 'cl-failfunc nil))
185            ((alval-p (setq obj (strip-alval (pop args)))) (waitfor-val obj))
186            ((message-other4 f obj args obj)))))
187
188  (defun message-other4(msg obj args orig-obj)
189    (cond ((eql msg 'delegate) (delegate-message obj args))
190          ((eql msg 'resend)   (resend-message obj args))
191          ((object-p obj) (send-message msg obj args orig-obj))
192          ((class-p  obj) (send-m-class msg obj args orig-obj))
193          (t              (send-bi-obj  msg obj args orig-obj))))
194
195  (defun send-m-class(msg c args orig-obj)
196     (case msg
197        (class-p   (give-value (class-name c)))
198        (object-p  (give-value al-f))
199        (class     (give-value al-f))
200        (name      (give-value (class-name c)))
201        (new       (create-instance c args orig-obj))
202        (otherwise
203         (when warning-exec
204               (p-warning "Message ~A unknown to class ~A" msg (class-name c)))
205         (argone 'cl-failfunc nil))))
206
207  (defun send-message(msg obj args orig-obj)
208    (case msg
209       (object-p  (give-value (object-name obj)))
210       (class-p   (give-value al-f))
211       (class     (give-value (object-class obj)))
212       (interface (get-interface obj args))
213       (otherwise
214        (cond ((member msg (object-interface obj))        (send-message-x msg obj args
 orig-obj))
```

```
215             ((member msg (current-interface-names obj)) (get-interface obj (list
 msg)))
216             (t (when warning-exec
217                      (p-warning "Message ~A unknown to object of class ~A"
218                                 msg (object-name obj))
219                      (p-warningc "Current interface: ~A" (object-interface obj)))
220                (argone 'cl-failfunc nil))))))
221
222  (defun send-message-x(msg obj args obj-self)
223    (if (bobject-p obj)
224        (send-msg-b msg obj args obj-self)
225      (send-message-x2 msg obj args obj-self)))
226
227  (defun send-message-x2(msg obj args obj-self)
228    (let* ((tabo      (class-tab (object-class obj)))
229           (ndn       (class-ndynnames (object-class obj)))
230           (curdyn    (gval-val (object-nextdyn obj)))
231           (nextdyn   (make-alvals-l (caddr ndn)))
232           (gmu  (make-muinfo-m :object obj
233                                :lenv   `(((original-msg ,msg ,obj-self ,@ args)
234                                           (here ,@ obj) (self ,@ obj-self))
235                                          ,(pairlis (cadr  ndn) curdyn)
236                                          ,(pairlis (caddr ndn) nextdyn)
237                                          ,@(object-inienv obj))
238                                :commit t)))
239      (make-child gmu p-lmu)
240      (setf (gval-val (object-nextdyn obj)) nextdyn)
241      (setf (muinfo-gmu gmu) gmu)
242      (setq p-lmu  gmu)
243      (setq p-call `($a$funcall-nomu ,p-result ,(get-sym-value tabo msg) ,@args))
244      (setq p-result alfvs-mu)
245      (setq alfvs-mu nil)
246      (setq tab tabo)
247      (next-the-same)))
248
249  (defun delegate-message(obj args) (declare (ignore args))
250     (let ((last-msg (get-val 'original-msg (muinfo-lenv p-lmu) tab)))
251        (message-other4 (car last-msg) obj (cddr last-msg) (cadr last-msg))))
252
253  (defun resend-message(obj args) (declare (ignore args))
254     (let ((last-msg (get-val 'original-msg (muinfo-lenv p-lmu) tab)))
255        (message-other4 (car last-msg) obj (cddr last-msg) obj)))
256
257  (defun get-interface(obj args)
258    (cond ((null args)
259           (p-error "Name expected for interface.")
260           (argone 'cl-failfunc nil))
261          ((alval-p args)      (waitfor-val args))
262          ((not (listp args))
263           (p-error "Name expected for interface.")
264           (argone 'cl-failfunc nil))
265          ((alval-p (setq args (strip-alval (car args)))) (waitfor-val args))
266          ((let ((int (get-interface-list obj args)))
267             (cond ((null int)
268                    (p-error "Unknown interface: ~A of a: ~A object"
```

```
269                          args (object-name obj))
270                   (p-errorc "Defined interfaces: ~A"
271                          (current-interface-names obj))
272                   (argone 'cl-failfunc nil))
273                  ((prog1 nil (pop int)))
274                  ((set-difference int (object-interface obj))
275                   (p-error "New interface ~A is not a subset of: ~A"
276                          int (object-interface obj))
277                   (argone 'cl-failfunc nil))
278                  ((let ((obj2 (copy-object obj)))
279                     (setf (object-interface obj2) int)
280                     (give-value obj2)))))))
281
282 (defun get-interface-list(obj name)
283   (let ((is (get-feature 'interfaces (class-origdef (object-class obj)))))
284     (assoc name is)))
285
286 (defun current-interface-names(obj)
287   (let ((is (get-feature 'interfaces (class-origdef (object-class obj)))))
288     (mapcar 'car is)))
289
290 ;_____
291 ; Process message new send to class.
292
293 (defun create-instance(cls args orig-obj)
294   (if (bclass-p cls)
295       (send-m-class-b 'new cls args)
296     (create-instance2 cls args orig-obj)))
297
298 (defun create-instance2(cls args obj-self)
299   (let* ((def  (class-origdef   cls))
300          (tabo (class-tab       cls))
301          (ndn  (class-ndynnames cls))
302          (curdyn    (make-alvals-l (cadr ndn)))
303          (nextdyn   (make-alvals-l (caddr ndn)))
304          (snames    (get-feature 'static def))
305          (static    (make-alvals-l snames))
306
307          (obj  (make-object
308                  :class cls
309                  :interface (get-feature 'methods def)
310                  :inienv    (list (pairlis snames static)
311                                   (spawn-obj-list (get-feature 'part def) 'p-))
312                  :nextdyn   (make-gval :val nextdyn)))
313          (gmu  (make-muinfo-m :object obj
314                               :lenv   `(((original-msg new ,obj-self ,@ args)
315                                          (here ,@ obj) (self ,@ obj-self))
316                                         ,(pairlis (cadr  ndn) curdyn)
317                                         ,(pairlis (caddr ndn) nextdyn)
318                                         ,@(object-inienv obj))
319                               :commit t)))
320     (setf (muinfo-gmu gmu) gmu)
321     (setq p-lmu  gmu)
322     (setq tab    tabo)
323     (link-vars (get-feature 'import  def) global-tab tab)
```

```
324     (setq p-lazy nil)  ; New function is eager.
325     (let ((prest (make-proc-m :call `($a$pass-v ,obj) :result p-result
326                               :lazy nil
327                               :lmu p-lmu :waitfor 1 :suspended p-suspended)))
328       (setq p-suspended prest))
329     (setq p-result dumb-alval)
330     (setq p-call `($a$funcall-vs ,(get-sym-value tabo 'new) ,@args))
331     (next-the-same)))
332
333 (defun make-alvals-l(lo)
334   (let ((l nil))
335     (while lo (push (if (sassg-p (car lo)) (make-alval) nil) l) (pop lo))
336     (nreverse l)))
337
338 (defun make-assoc-t(names mytab)
339   (let ((l nil))
340     (while names (push (cons (car names) (get-sym-value mytab (car names))) l)
341                  (pop names))
342     l))
```

A.2.3 rcall.l

Handling of replicators and nested generators:

```
;-------------------------------------------------------------------------------
; Handling of replicators and nested generators.

;-------------------------------------------------------------------------------
; General generator call.
; Run-time check for generator type (i.e. fake generator, lazy etc.).

(defstruct (fgrec)
  (result nil)
  (expr   nil)
  (lazy   nil)
  (act    nil))

(defun spec-$a$gen-call-gq()
  (let (fname)
    (setq fname (caddr p-call))
    (case fname
          (delay (setq p-lazy 'delay)(gcgq-popcall) (next-the-same))
          (lazy  (setq p-lazy lazyd) (gcgq-popcall) (next-the-same))
          (eager (setq p-lazy nil)   (gcgq-popcall) (next-the-same))
          (otherwise
           (cond ((get-spec-value fname)
                  (give-value (make-fgen-call (cadr p-call)
                                              (caddr p-call) (cdddr p-call))))
                 ((setq p-call
                        (make-slist-t (cons '$a$cgen-call (cddr p-call))
                                      (cadr p-call)))
                  (next-the-same)))))))

(defun gcgq-popcall()
  (let* ((call (cadddr p-call)) serp fname args)
    (cond ((slist-p call) (setq call (rm-slist call))
           (setq serp t)   (setq fname (car call)) (setq args (cdr call)))
          ((listp call)
           (setq serp nil) (setq fname (car call)) (setq args (cdr call)))
          (t
           (setq serp nil) (setq fname 'passtrue)  (setq args (list call))))
    (setq p-call `($a$gen-call-gq ,serp ,fname ,@args))))

; All lowest level generator calls are made here.
(defun cl-$a$cgen-call(fnargs)
  (let ((fname (car fnargs)) (args (cdr fnargs)))
    (cond ((alval-p (setq fname (strip-alval fname)))
           (cons 'sus fname))
          ((cons 'suc (make-fgen-specmu-call p-serial fname args))))))

(defun make-fgen-call(serp fname args)
  (make-fgrec :expr   (make-slist-t (cons fname args) serp)
              :result (make-alval-m)  :lazy p-lazy))

(defun make-fgen-specmu-call(serp fname args)
  (let ((s (make-fgen-call serp fname args)))
```

```
53      (setf (fgrec-act s) 'first)
54      s))
55
56  (defun cl-$a$fgen-next(fgrec)
57    (cond ((alval-p  (setq fgrec (strip-alval fgrec))) (cons 'sus fgrec))
58          ((not (fgrec-p fgrec))
59           (al-error "$a$fgen-next: fgrec expected in: ~A" fgrec))
60          ((fgrec-act fgrec)
61           (case (fgrec-act fgrec)
62                 (first     (fgen-next-act fgrec))
63                 (wait      (cons 'sus (fgrec-result fgrec)))
64                 (otherwise
65                  (if (genstate-p (fgrec-act fgrec))
66                      (cl-$a$gen-next (fgrec-act fgrec))
67                    (al-error "Internal. Unknown fake generator: ~A" fgrec)))))
68          ((fgen-next fgrec))))
69
70  (defun fgen-next(fgrec)
71    (let ((result (strip-alval (fgrec-result fgrec))))
72      (cond ((not (alval-p result))               (cons 'suc al-f))
73            ((let* ((gval (make-gval))
74                    (ps   (make-proc-m :call `($a$csuspended-gl , gval)
75                                       :result dumb-alval :lazy nil
76                                       :lmu p-lmu :waitfor 1
77                                       :suspended p-suspended))
78                    (px   (make-pprocess (fgrec-expr fgrec))))
79               (setf (proc-lazy px) (fgrec-lazy fgrec))
80               (setf (proc-suspended px) ps)
81               (alset-alval (cons (cons gval (proc-result px)) nil) result)
82               (cons 'suc (car (strip-alval result))))))))
83
84  (defun fgen-next-act(fgrec)
85    (setf (fgrec-act fgrec) 'wait)
86    (let* ((gval     (make-gval))
87           (res      (make-alval))
88           (ps   (make-proc-m :call `($a$csuspended-gl , gval)
89                              :result dumb-alval :lazy nil
90                              :lmu p-lmu :waitfor 1
91                              :suspended p-suspended))
92           (px   (make-pprocess `($a$funcall-nomu ,res ,@(fgrec-expr fgrec))))
93           (pf   (make-pprocess `($a$fcnmu-final ,(proc-result px) ,gval ,res ,fgrec))))
94      (setf (proc-lazy px) (fgrec-lazy fgrec))
95      (make-eager pf)
96      (setf (proc-suspended px) ps)
97      (cons 'suc (proc-result pf))))
98
99  (defun cl-$a$fcnmu-final(typ gval res fgrec)
100   (cond ((alval-p (setq typ   (strip-alval typ)))   (cons 'sus typ))
101         ((alval-p (setq fgrec (strip-alval fgrec))) (cons 'sus fgrec))
102         ((alval-p (setq gval  (strip-alval gval)))  (cons 'sus gval))
103         ((proc-p typ)
104          (when (zerop (decf (proc-waitfor p-suspended)))
105                  (al-error "$a$fcnmu-final: Internal."))
106          (let ((g (cdr (apply-gproc typ nil))))
107            (setf (fgrec-act fgrec) g)
```

```
108             (alset-alval (genstate-result g) (fgrec-result fgrec))
109             (cons 'suc (proc-result (make-pprocess `($a$gen-next ,g))))))
110           (t
111            (alset-alval (list (cons gval res)) (fgrec-result fgrec))
112            (setf (fgrec-act fgrec) nil)    (cons 'suc (cons gval res)))))
113
114  ;-------------------------------------------------------------------------------
115  ; FAST linear-time generators when #! is in use.
116
117  (defun cl-$a$directg(gl f &rest args)
118    (let* ((res (make-alval))
119           (px  (make-pprocess (make-slist-t `($a$funcall-nomu ,res ,f ,@args)
 p-serial)))
120           (pf  (make-pprocess `($a$dirg-final ,(proc-result px) ,gl ,res))))
121      (make-eager px) (make-eager pf)
122      (cons 'suc (proc-result pf))))
123
124  (defun cl-$a$dirg-final(typ gl res)
125    (cond ((alval-p (setq typ (strip-alval typ))) (cons 'sus typ))
126          ((alval-p (setq gl  (strip-alval gl)))  (cons 'sus gl))
127          ((proc-p typ)
128           (let ((lmu (if gl (muinfo-gmu p-lmu) p-lmu)))
129             (setf (muinfo-result (proc-lmu typ))
130                   (if (muinfo-p (muinfo-result lmu)) (muinfo-result lmu) lmu)))
131           (push typ proc-stack)
132           (cons 'suc 88))
133          (t
134           (gen-or-commit (if gl (muinfo-gmu p-lmu) p-lmu))
135           (cons 'suc res))))
136
137  ;-------------------------------------------------------------------------------
138  ; Run-time replicator.
139
140  (defun cl-$a$rep-driver(serp clex ggen rest)
141    (cond ((alval-p (setq serp (strip-alval serp))) (cons 'sus serp))
142          ((alval-p (setq clex (strip-alval clex))) (cons 'sus clex))
143          ((alval-p (setq ggen (strip-alval ggen))) (cons 'sus ggen))
144          ((alval-p (setq rest (strip-alval rest))) (cons 'sus rest))
145          (t
146           (make-eager (make-pprocess `($a$ggen-next ,ggen)))
147           (let ((pl (make-pprocess
148                       (list '$a$rd-loop (ggen-results ggen)
149                             ggen serp clex
150                             (if serp rest
151                               (proc-result (make-pprocess rest)))))))
152             (cons 'suc (proc-result pl))))))
153
154  (defun make-eager(lproc)
155    (setf (proc-lazy lproc) nil))
156
157  (defun reset-lazy-self()
158    (when (eq lazyt p-lazy) (setq p-lazy lazyd)))
159
160  (defun spec-$a$rd-loop()
161    (let* ((args    (cdr p-call))
```

```
162           (results (strip-alval (pop args))))
163      (cond ((alval-p results) (waitfor-val  results))
164            (t                 (apply 'rd-loop-cont (cons results args))))))
165
166  (defun rd-loop-cont(results ggen serp clex rest)
167    (cond ((null results)
168           (give-value (if serp (proc-result (make-pprocess rest)) rest)))
169          ((let ((newend (make-alval-m))
170                 (res    (make-alval-m)))
171             (alset-alval (cons res newend) p-result)
172             (reset-lazy-self)
173             (setq p-result newend)
174             (setq p-call (cons (car p-call) (cons (cdr results) (cddr p-call))))
175             (if serp
176                 (let ((pexec nil))
177                   (incf p-waitfor)
178                   (construct-procc)
179                   (setq pexec (make-proc-m :call `($a$rd-exec ,(car results)
180                                                               ,ggen ,serp ,clex ,p-lazy)
181                                            :result res :lmu p-lmu :waitfor 0
182                                            :suspended proc))
183                   (make-eager pexec)
184                   (push pexec proc-stack))
185               (let ()
186                 (make-eager (make-pprocessv `($a$rd-exec ,(car results)
187                                                          ,ggen ,serp ,clex ,p-lazy)
188                                             res))
189                 (next-the-same)))))))
190
191  (defun spec-$a$rd-exec()
192    (let* ((args (cdr p-call))
193           (gval-res (pop args)) (ggen   (pop args))
194           (serp     (pop args)) (clex   (pop args)) (lazy   (pop args)))
195      (cond ((not serp)
196             (make-eager (make-pprocess `($a$ggen-next ,ggen)))
197             (setq p-call (funcall clex (cdr gval-res)))
198             (setq p-lazy lazy) (reset-lazy-self)
199             (check-proc-call)
200             (next-the-same))
201            ((if-gval-sus-proc (car gval-res) proc)
202             (incf (proc-waitfor proc)))
203            ((let ((pn (make-proc-m :call `($a$ggen-next ,ggen)
204                                    :result dumb-proc :lazy nil
205                                    :lmu p-lmu :waitfor 1
206                                    :suspended p-suspended)))
207               (setq p-suspended pn)
208               (setq p-call (funcall clex (cdr gval-res)))
209               (setq p-lazy lazy) (reset-lazy-self)
210               (check-proc-call)
211               (next-the-same))))))
212
213  ;-------------------------------------------------------------------------------
214  ; Nested generators driver.
215
216  (defstruct (mngrec)
```

```
217   (results nil)
218   (resend  nil)
219   (serp    nil)
220   (ggen    nil)
221   (clex  nil))
222
223  ;Takes as parameters a general generator ggen and a cl expr producing a
224  ;  general generator ggeni.
225  ;  Produces all possible ggenis, calls ggenis and returns all their values.
226  (defun cl-$a$gen-driver(serp clex ggen)
227    (cond ((alval-p (setq serp (strip-alval serp)))   (cons 'sus serp))
228          ((alval-p (setq clex (strip-alval clex)))   (cons 'sus clex))
229          ((alval-p (setq ggen (strip-alval ggen)))   (cons 'sus ggen))
230          (t
231           (let* ((v (make-alval-m))
232                  (mngrec (make-mngrec :results v :resend v
233                            :ggen ggen :clex clex :serp serp))
234                  (pl (make-proc-m
235                        :call (list '$a$gd-loop (ggen-results ggen)
236                                    mngrec nil)
237                        :result v :lmu p-lmu :waitfor 0
238                        :suspended dumb-proc)))
239             (delay-proc pl) (push pl proc-stack)
240             (cons 'suc mngrec)))))
241
242  (defun spec-$a$gd-loop()   ; global-gen-results mngrec rest
243    (let ((mngrec  (caddr p-call)))
244      (make-eager (make-pprocess `($a$ggen-next ,(mngrec-ggen mngrec))))
245      (setq p-call (cons '$a$gd-loop-cont (cdr p-call)))
246      (next-the-same)))
247
248  (defun spec-$a$gd-loop-cont()   ; global-gen-results mngrec rest
249    (let* ((args (cdr p-call))
250           (results (strip-alval (pop args)))
251           (mngrec  (pop args)) (rest (pop args)))
252      (cond ((alval-p results) (waitfor-val results))
253            ((null-p  results) (give-value  rest))
254            ((let ((lgen (make-alval-m)) (fend   (make-alval-m)) cont g2)
255               (setq cont (make-pprocessv
256                           `($a$gd-loop ,(cdr results) ,mngrec ,rest) fend))
257               (delay-proc cont)
258
259               (setq g2 (funcall (mngrec-clex mngrec) (cdar results)))
260               (make-eager
261                 (make-pprocessv
262                   (make-slist-t
263                     (cons '$a$gen-call-gq (cons (slist-p g2) (rm-slist g2)))
264                     (slist-p g2)) lgen))
265
266               (setq p-call `($a$genllist ,lgen ,(caar results) ,fend))
267               (cond ((mngrec-serp mngrec)
268                      (construct-procc)
269                      (if (if-gval-sus-proc (caar results) proc)
270                          (incf (proc-waitfor proc))
271                        (next-the-same)))
```

```
272                   (t
273                    (next-the-same))))))))
274
275  (defun spec-$a$genllist()   ; coex gval rest
276    (let ((coex (strip-alval (cadr p-call))))
277      (cond ((alval-p coex)          (waitfor-val coex))
278            (t (setq p-call (list '$a$genlmake (ggen-results coex)
279                                  coex (cddr p-call)))
280               (delay-me)))))
281
282  (defun spec-$a$genlmake()   ; local-results coex gval-res
283    (let ((coex (caddr p-call)) pnext)
284      (setq pnext (make-pprocess (list '$a$ggen-next coex)))
285      (make-eager pnext)
286      (setq p-call (cons '$a$genlmakew (cdr p-call)))
287      (next-the-same)))
288
289  (defun spec-$a$genlmakew()   ; local-results coex gval-res
290    (let ((next       (strip-alval (cadr p-call)))
291          (gval-rest  (car (last p-call))))
292      (cond ((alval-p next)   (waitfor-val next))
293            ((null    next)   (give-value  (cadr gval-rest)))
294            ((let ((newend (make-alval-m))
295                   (gval (ngen-gval (car gval-rest) (caar next))))
296               (alset-alval (cons (cons gval (cdar next)) newend) p-result)
297               (setq p-result newend)
298               (setq p-call (cons '$a$genlmake (cons (cdr next) (cddr p-call))))
299               (delay-me))))))
300
301  (defun cl-$a$gdgen-next(mngrec)
302    (cond ((alval-p  (setq mngrec (strip-alval mngrec))) (cons 'sus mngrec))
303          ((not (mngrec-p mngrec))
304           (al-error "$a$gdgen-next: mngrec expected in: ~A" mngrec))
305          ((mdgen-next mngrec))))
306
307  (defun mdgen-next(mngrec)
308    (let ((resend (strip-alval (mngrec-resend mngrec))))
309      (cond ((alval-p resend)                    (cons 'sus resend))
310            ((null-p  resend)                    (cons 'suc al-f))
311            (t                                   ; Can now take a result.
312             (when (if-gval-sus-proc (caar resend) p-suspended)
313               (incf (proc-waitfor p-suspended)))
314             (setf (mngrec-resend mngrec) (cdr resend))
315             (cons 'suc (car resend))))))
316
317  (defun ngen-gval(gv1 gv2)
318    (cond ((and (wake-up-gval-p gv1) (wake-up-gval-p gv2)) gv1)
319          ((wake-up-gval-p gv1)                          gv2)
320          ((wake-up-gval-p gv2)                          gv1)
321          ((let ((gval (make-gval)) pwake)
322             (setq pwake (make-proc-m :call `($a$csuspended-gl ,gval)
323                                      :result dumb-alval :lmu p-lmu :lazy nil
324                                      :waitfor 2 :suspended dumb-proc))
325             (push pwake (gval-val gv1))
326             (push pwake (gval-val gv2))
```

```
327            gval))))
328
329  ;-----------------------------------------------------------------------------
330  ; General stuff.
331
332  (defun spec-$a$ggen-next()
333    (let ((gen (cadr p-call)))
334      (cond ((alval-p (setq gen (strip-alval gen)))
335             (waitfor-val gen))
336            ((genstate-p gen)
337             (setq p-call (list '$a$gen-next   gen)) (next-the-same))
338            ((mgrec-p gen)
339             (setq p-call (list '$a$mgen-next  gen)) (next-the-same))
340            ((and (fgrec-p gen) (genstate-p (fgrec-act gen)))
341             (setq p-call (list '$a$gen-next   (fgrec-act gen))) (next-the-same))
342            ((fgrec-p gen)
343             (setq p-call (list '$a$fgen-next  gen)) (next-the-same))
344            ((mngrec-p gen)
345             (setq p-call (list '$a$gdgen-next gen)) (next-the-same))
346            ((al-error "Generator expected in: ~A" p-call)))))
347
348  (defun ggen-results(gen)
349    (cond ((genstate-p gen)   (genstate-result gen))
350          ((fgrec-p    gen)   (fgrec-result    gen))
351          ((mgrec-p    gen)   (mgrec-results   gen))
352          ((mngrec-p   gen)   (mngrec-results  gen))
353          ((al-error "Generator expected in: ~A" gen))))
354
355  ;-----
356  ; Create association list for environment.
357
358  (defun make-blist(names values)
359    (make-blist2 names (strip-alval values)))
360
361  (defun make-blist2(names values)
362    (cond ((null-p  names) nil)
363          ((symbolp names) (list (cons names values)))
364          ((cons-p  names) (append (make-blist2 (car names) (car values))
365                                   (make-blist2 (cdr names) (cdr values))))
366          ((al-error "make-blist2: Unexpected name structure: ~A" names))))
367
368  (defun replace-syms(e blist)
369    (cond ((cons-p   e) (cons (replace-syms (car e) blist)
370                              (replace-syms (cdr e) blist)))
371          ((slist-p  e) (make-slist :expr (replace-syms (slist-expr e) blist)))
372          ((symbol-p e) (let ((v (assoc e blist)))
373                          (if v `(quote ,(cdr v)) e)))
374          (e)))
375
```

A.2.4 gcall.l

Handling of basic generators.

```
;-------------------------------------------------------------------------------
; Actual Generator calls.

(defstruct (genstate)
  (result    nil)
  (finished  (make-queue))
  (resend    nil)
  (next      nil)
  (rest      nil)
  (suspended (make-queue)))

(defstruct (mgrec)
  (results   nil)
  (resend    nil)
  (sergens   nil))

;-------------------------------------------------------------------------------
; Generators. Calls and interpretation.

(defun apply-gproc(a-proc rest)
  (let* ((a-mu   (proc-lmu a-proc))
         (res    (make-alval-m))
         (coex   (make-genstate :result    res
                                :resend    res
                                :next      res
                                :rest      rest)))
    (incf (proc-waitfor a-proc))
    (queue-put a-proc (genstate-suspended coex))
    (setf (muinfo-result    a-mu) coex)
    (setf (muinfo-suspended a-mu) dumb-proc)
    (setf (proc-call (proc-suspended a-proc)) (list '$a$alfun-gfail))
    (cons 'suc coex)))

(defun gen-lreturn-lmu(lmu)
  (let ((new-end  (make-alval-m))
        (coex     (muinfo-result lmu)))
    (let* ((gval     (make-gval))
           (signaler (make-proc-m :call  `($a$csuspended-gl ,gval)
                                  :result dumb-alval :lazy nil
                                  :lmu p-lmu :waitfor 1
                                  :suspended p-suspended)))
      (alset-alval (cons (cons gval p-result) new-end) (genstate-resend coex))
      (setf (genstate-resend coex) new-end)
      (queue-put gval        (genstate-finished  coex))
      (queue-put p-suspended (genstate-suspended coex))
      (incf (proc-waitfor p-suspended))
      (setq p-suspended signaler))))

(defun spec-alfun-gfail()
  (let ((gstate (muinfo-result p-lmu)))
    (alset-alval (genstate-rest gstate)  (genstate-resend gstate))
    (setf (genstate-resend gstate) nil)))          ; completed
```

```
53
54  ;-----------------------------------------------------------------------------
55  ; Functions on Generators.
56
57  (defun cl-$a$gen-results(e)
58    (cl-$a$gen-results-u e))
59
60  (defun cl-$a$gen-results-u(e)
61    (let ((sval (strip-alval e)))
62      (cond ((alval-p sval)             (cons 'sus sval))
63            ((not (genstate-p    sval)) (cons 'suc al-f))
64            (t                         (cons 'suc (genstate-result sval))))))
65
66  (defun cl-$a$gen-next(e)
67    (cl-$a$gen-next-u e))
68
69  (defun cl-$a$gen-next-u(e)
70    (let ((sval (strip-alval e)))
71      (cond ((alval-p sval)             (cons 'sus sval))
72            ((not (genstate-p    sval)) (cons 'suc al-f))
73            (t                         (gen-next sval)))))
74
75  (defun gen-next(coex)
76    (let ((next (strip-alval (genstate-next coex)))
77          (q    (genstate-suspended coex)))
78      (setf (genstate-next coex) next)
79      (if (queue-empty q)
80          (cond ((alval-p next) (cons 'sus next))
81                ((null-p  next) (cons 'suc al-f))
82                ((al-error "gen-next: unacounted generated values in: ~A" next)))
83        (progn (check-suspended1 (queue-get q))
84               (gen-next-coex next coex)))))
85
86  (defun cl-$a$gen-nextw(e)
87    (cl-$a$gen-nextw-u e))
88
89  (defun cl-$a$gen-nextw-u(coex)
90    (gen-next-coex (strip-alval (genstate-next coex)) coex))
91
92  (defun gen-next-coex(next coex)
93    (cond ((alval-p next)
94           (setq p-call  `($a$gen-nextw ,coex))
95           (cons 'sus next))
96          ((null next) (cons 'suc al-f))
97          ((let ((res (car next))
98                 (sig (queue-get (genstate-finished coex))))
99             (setf (genstate-next coex) (cdr next))
100            (check-gval-sus sig)
101            (cons 'suc res)))))
102
103 (defun if-gval-sus-proc(sig proc)
104   (when (listp (gval-val sig))
105     (push proc (gval-val sig))))
106
107 (defun check-gval-sus(sig)
```

```
108   (when (if-gval-sus-proc sig p-suspended)
109     (setq p-suspended dumb-proc)))
110
111 (defun spec-$a$csuspended-gl()
112   (wake-up-gval (cadr p-call))
113   (check-suspended))
114
115 (defun wake-up-gval-p(gval)
116   (eq (gval-val gval) t))
117
118 (defun wake-up-gval(gval)
119   (let ((l (gval-val gval)))
120     (while (cons-p l) (check-suspended1 (pop l))))
121   (setf (gval-val gval) t))
122
123 ;-------------------------------------------------------------------------------
124 ; Multiple coexpressions list producing coexpression.
125
126 (defun cl-$a$mgen-call-sr(rest sergens &rest lcoex)
127   (cl-$a$mgen-call-u rest sergens lcoex))
128
129 (defun cl-$a$mgen-call(sergens &rest lcoex)
130   (cl-$a$mgen-call-u () sergens lcoex))
131
132 (defun cl-$a$mgen-call-u(rest sergens lcoex)
133   (cond ((alval-p (setq sergens (strip-alval sergens))) (cons 'sus sergens))
134         ((my-member-ng
135            (setq lcoex (strip-alval-e lcoex)))          (cons 'sus mmng-var))
136         ((null-p lcoex)                                 (cons 'suc ()))
137         ((let ((v (make-alval-m)) mgrec pmg)
138            (setq mgrec (make-mgrec
139                          :results v :resend v :sergens sergens))
140            (setq pmg (make-proc-m :call (list '$a$mgmaker (makegens lcoex)
141                                               () () () mgrec rest)
142                                   :result v :lmu p-lmu :waitfor 0
143                                   :suspended dumb-proc))
144            (delay-proc pmg) (push pmg proc-stack)
145            (cons 'suc mgrec)))))
146
147 (defun delay-proc(proc)
148   (unless p-lazy (setf (proc-lazy proc) 'delay)))
149
150 (defun makegens(clist)
151   (when clist (cons (cons (car clist) (ggen-results (car clist)))
152                     (makegens (cdr clist)))))
153
154 (defun spec-$a$mgmaker()
155   (cond ((cadr p-call) (apply 'mgmaker-recs (cdr p-call)))
156         (t             (apply 'mgmaker-ret  (cdr p-call)))))
157
158 (defun mgmaker-ret(lcoex lres lresok lresfin mgrec rest)
159   (declare (ignore lcoex))
160   (setq lresok (strip-alval-e lresok))
161   (cond ((my-member-ng lresok)     (waitfor-val mmng-var))
162         ((not (my-andl lresok))    (give-value rest))
```

```
163         ((let ((newres (make-alval-m)) (gval (make-gval)))
164            (alset-alval (cons (cons gval newres) rest) p-result)
165            (alset-alval (strip-alval-e (reverse lres)) newres)
166            (setq p-result dumb-alval)
167            (cond ((mgrec-sergens mgrec) (mgreturn-fin gval))
168                  (t (setq lresfin (strip-alval-e lresfin))
169                     (setq p-call (list '$a$mgreturn gval))
170                     (construct-procc)
171                     (wait-on-gvals lresfin proc)
172                     (when (zerop (proc-waitfor proc)) (mgreturn-fin gval))))))))
173
174  (defun wait-on-gvals(lgs proc)
175    (while lgs
176      (when (if-gval-sus-proc (pop lgs) proc) (incf (proc-waitfor proc)))))
177
178  (defun spec-$a$mgreturn()
179    (mgreturn-fin (cadr p-call)))
180
181  (defun mgreturn-fin(gval)
182    (wake-up-gval gval)
183    (check-suspended))
184
185  (defun mgmaker-recs(lcoex lres lresok lresfin mgrec rest)
186    (cond ((null       (cdar lcoex))    (give-value rest))
187          ((not (alval-p (cdar lcoex)))
188           (mgm-ok  lcoex lres lresok lresfin mgrec rest))
189          ((alval-init (cdar lcoex))
190           (mgm-ok  lcoex lres lresok lresfin mgrec rest))
191          ((mgm-gen lcoex lres lresok lresfin mgrec rest))))
192
193  (defun mgm-ok(lcoexrs lres lresok lresfin mgrec rest)
194    (let ((midrest (make-alval-m))
195          (coexrs  (strip-alval-e (pop lcoexrs))) gvalsus p1 p2)
196      (setq gvalsus (caadr coexrs))
197      (setq coexrs (cons (car coexrs) (cons (cdadr coexrs) (cddr coexrs))))
198      (incf (proc-waitfor p-suspended))
199      (setq p2 (make-proc-m
200                 :call (list '$a$mgmaker
201                             (cons (cons (car coexrs) (cddr coexrs)) lcoexrs)
202                             lres lresok lresfin mgrec rest)
203                 :result midrest :lmu p-lmu
204                 :waitfor   0
205                 :suspended p-suspended))
206      (push p2 proc-stack)
207      (delay-proc p2)
208      (setq p1 (make-proc-m
209                 :call (list '$a$mgmaker
210                             lcoexrs
211                             (cons (cadr coexrs) lres)
212                             (cons t lresok) (cons gvalsus lresfin)
213                             mgrec midrest)
214                 :result p-result :lmu p-lmu :waitfor 0
215                 :suspended p-suspended))
216      (if (and (mgrec-sergens mgrec) (if-gval-sus-proc gvalsus p1))
217          (incf (proc-waitfor p1))
```

```
218       (push p1 proc-stack))))
219
220  ;_____
221  ; Executes before mgm-gen and makes sure the call is needed.
222
223  (defun spec-$a$mgmaker-wait()
224    (let* ((args   (cdr p-call))
225           (coex (pop args)) (rres (pop args)) (lcoexrsres (pop args)) (lrest (pop
 args)))
226      (setq rres (strip-alval rres))
227      (cond ((alval-p rres) (waitfor-val rres))
228            ((null-p  rres) (give-value (car (last lrest))))
229            ((mgm-gen (cons (cons coex (cdr rres)) lcoexrsres)
230                      (pop lrest) (pop lrest) (pop lrest) (pop lrest) (pop lrest))))))
231
232  ; Actions taken as soon as generator starts producing a result.
233  ;=====
234  (defun spec-$a$mgmaker-res()
235    (let* ((args   (cdr p-call))
236           (okres  (pop args)) (oldres (pop args))
237           (finres (pop args)) (rres   (pop args)))
238      (setq rres (strip-alval rres))
239      (cond ((alval-p rres) (waitfor-val rres))
240            ((null-p  rres) (alset-alval nil  okres)    ; No result
241                            (alset-alval al-f oldres)
242                            (alset-alval al-f finres)
243                            (check-suspended))
244            (t         (alset-alval t           okres)
245                       (alset-alval (cdar rres) oldres)
246                       (alset-alval (caar rres) finres)
247                       (check-suspended)))))
248
249  (defun mgm-gen(lcoexrs lres lresok lresfin mgrec rest)
250    (let ((midrest (make-alval-m))
251          (serr    (mgrec-sergens  mgrec))
252          (coexrs  (pop lcoexrs))
253          (okres   (make-alval-m)) (oldres  (make-alval-m)) (finres  (make-alval-m))
254          p0 p1 p2)
255      (incf (proc-waitfor p-suspended))
256      (unless serr (incf (proc-waitfor p-suspended)))
257      (make-pprocessv (list '$a$mgmaker-res okres oldres finres (cdr coexrs))
258                      dumb-alval)
259      (setf (proc-lazy (car proc-stack)) nil)
260      (setq p2 (make-proc-m
261                 :call (list '$a$mgmaker-wait
262                             (car coexrs) (cdr coexrs) lcoexrs
263                             (list lres lresok lresfin mgrec rest))
264                 :result midrest :lmu p-lmu
265                 :waitfor   0
266                 :suspended p-suspended))
267      (push p2 proc-stack)
268      (delay-proc p2)
269      (setq p1 (make-proc-m
270                 :call (list '$a$mgmaker
271                             lcoexrs
```

```
272                            (cons oldres lres)
273                            (cons okres lresok) (cons finres lresfin)
274                            mgrec midrest)
275                 :result p-result :lmu p-lmu
276                 :waitfor   (if serr 1 0)
277                 :suspended p-suspended))
278     (unless serr (push p1 proc-stack))
279     (setq p0 (make-proc-m
280                 :call (list '$a$ggen-next (car coexrs))
281                 :result dumb-alval :lmu p-lmu :waitfor 0  :lazy nil
282                 :suspended (if serr p1 p-suspended)))
283     (push p0 proc-stack)))
284
285  (defun cl-$a$mgen-next(mgrec)
286    (cond ((alval-p (setq mgrec (strip-alval mgrec))) (cons 'sus mgrec))
287          ((not (mgrec-p mgrec))
288           (al-error "cl-$a$mgen-next: Not multiple gens record: ~A" mgrec))
289          (t                                          (mgen-next mgrec))))
290
291  (defun mgen-next(mgrec)
292    (let ((resend (strip-alval (mgrec-resend mgrec))))
293      (cond ((alval-p resend)                    (cons 'sus resend))
294            ((null-p  resend)                    (cons 'suc al-f))
295            (t                                   ; Can now take a result.
296             (when (if-gval-sus-proc (caar resend) p-suspended)
297               (incf (proc-waitfor p-suspended)))
298             (setf (mgrec-resend mgrec) (cdr resend))
299             (cons 'suc (car resend))))))
300
301  ;-------------------------------------------------------------------------------
302  ; Results
303
304  (defun make-pprocessv(e v)
305     (make-addproc (to-al-call e) v p-lmu 0 p-suspended)
306     (incf (proc-waitfor p-suspended))
307     (car proc-stack))
308
309  (defun my-and(&rest args)
310    (my-andl args))
311
312  (defun my-andl(l)
313    (if l (and (car l) (my-andl (cdr l))) t))
314
```

A.3 Pre-processor

A.3.1 prep.l

Basic pre-processor. Handles read macro expressions

```
;-------------------------------------------------------------------------------
; ALLOY preprocessor. Main part.
; Read macros. Syntax checking of special expressions.

(defun al-perror(&rest msgs)
  (throw 'al-prep (apply 'sformat msgs)))

(defun al-rpp()
  (let ((e (al-prep (al-read))))
;    (al-print e)
    (pprint e)
    t))

;-------------------------------------------------------------------------------
; Mail preprocessoring call. Takes care of error conditions.

(defun al-prep(e)
  (let ((pe (catch 'al-prep (list (al-prep-main e)))))
    (if (cons-p pe) (car pe)
      (progn (p-error pe) al-f))))

(defun al-prep-main(e)
  (oprep (gprep (prep e nil))))

(defun prep(e gok)
  (cond ((rep-p e)           (prep-rep0 e gok))
        ((gen-p e)           (prep-gen0 e gok))
        ((not (xcons-p e))   e)
        ((eq (xcar e) 'lets) (prep (prep-lets e) gok))
        ((eq (xcar e) 'let)  (prep (prep-let  e) gok))
        ((cons-p  e)         (prep-call e gok nil))
        ((scons-p e)         (setf (slist-expr e)
                                   (prep-call (slist-expr e) gok t)) e)))

(defun prep-rep0(e gok)                 (declare (ignore gok))
  (let ((pe (prep-rep (rep-expr e))))
    (if (outer-gen-p pe)
        (progn (setf (rep-expr e) pe) e)
      pe)))

(defun prep-rep(e)
  (prep e t))

(defun prep-gen0(e gok)
  (unless gok
    (al-perror "Unexpected generator call: ~A" e))
  (when (rep-p (gen-expr e))
    (al-perror "Unexpected replicator call: ~A" (gen-expr e)))
  (let ((pe (prep-gen (gen-expr e))))
    (when (and (gen-p pe) (gen-p (gen-expr pe)))
```

```
51        (setq pe (gen-expr pe)))
52      (when (and (gen-p pe) (not (xcons-p (gen-expr pe))))
53        (setq pe (gen-expr pe)))
54      (setf (gen-expr e) pe) e))
55
56  (defun prep-gen(e)
57    (prep (if (xcons-p e) (case (xcar e)
58                            (let       (prep-glet e))
59                            (letrec    (prep-gletrec  (rm-slist e) (slist-p e)))
60                            (letrecs   (prep-gletrecs (rm-slist e) (slist-p e)))
61                            (otherwise e))
62            e) t))
63
64  (defun prep-list(e gok)
65    (cond ((null e) nil)
66          ((atom e) (prep e gok))
67          ((cons (prep (car e) gok) (prep-list (cdr e) gok)))))
68
69  (defun prep-call(e gok serp)
70    (case (car e)
71      (letrecs  (pcheck-length e 2 9999) (prep (prep-letrecs e serp) gok))
72      (letrec   (pcheck-length e 2 9999) (prep (prep-letrec  e serp) gok))
73      (function (pcheck-length e 2 2) (no-rep-the (cadr e))
74                `(mu () ,(prep (make-rep :expr `(lreturn ,(cadr e))) nil)))
75      (quote    (pcheck-length e 2 2) (no-rep-the (cadr e)) (prep-list e gok))
76      (delay    (pcheck-length e 2 2) (no-rep-the (cadr e)) (prep-list e gok))
77      (eager    (pcheck-length e 2 2) (no-rep-the (cadr e)) (prep-list e gok))
78      (lazy     (pcheck-length e 2 2) (no-rep-the (cadr e)) (prep-list e gok))
79      (return   (pcheck-length e 1 2) (no-rep-the (cadr e))
80                (prep-list e gok))
81      (lreturn  (pcheck-length e 1 2) (no-rep-the (cadr e))
82                (prep-list e gok))
83      (setfun   (pcheck-length e 2 9999)
84                (if (cons-p (cadr e))
85                    (progn (pcheck-length e 2 9999) (no-rep-the (cadr e)))
86                    (progn (pcheck-length e 3 9999) (no-rep-the (cadr e))))
87                (setq e (check-fdef (cdr e)))
88                (list '$a$setfun1 (car e)
89                      (make-slist-t
90                        (cons 'mu (cons (cadr e) (prep-list (cddr e) nil))) serp)))
91      (mu       (pcheck-length e 2 9999) (no-rep-the (cadr e))
92                (setq e (check-fdef (cons 'none (cdr e))))
93                (cons 'mu (cons (cadr e) (prep-list (cddr e) nil))))
94      (set      (pcheck-length e 3 3) (no-rep-the (caddr e))
95                (no-rep-the (cadr e))
96                (prep-set (cdr e) gok serp))
97      (when     (pcheck-length e 2 9999)
98                (when (rep-p (cadr e))
99                  (al-perror "Condition cannot be a replicator in: ~A" e))
100               (cons 'when (prep-list (cdr e) gok)))
101     (unless   (pcheck-length e 2 9999)
102               (when (rep-p (cadr e))
103                 (al-perror "Condition cannot be a replicator in: ~A" e))
104               (cons 'unless (prep-list (cdr e) gok)))
105     (if       (pcheck-length e 3 4)
```

```
106               (when (rep-p (cadr e))
107                 (al-perror "Condition cannot be a replicator in: ~A" e))
108               (when (rep-p (caddr e))
109                 (al-perror "Simple expression expected in then part: ~A" e))
110               (when (and (cons-p (cdddr e)) (rep-p (caddr e)))
111                 (al-perror "Simple expression expected in else part: ~A" e))
112               (cons 'if (prep-list (cdr e) gok)))
113     (otherwise
114      (cond ((rep-p (car e)) (prep-list (cons '$a$funcall e) gok))
115            ((prep-list e gok))))))
116
117  (defun no-rep-the(e)
118    (when (rep-p e)
119          (al-perror "Unexpected replicator in: ~A" e)))
120
121  (defun prep-return(local e serp)
122    (let ((rets (pr-makers local (cdr e) serp)))
123      (cond ((null   rets)       e)
124            ((atom-p rets)       rets)
125            ((cons-p (cdr rets)) (cons 'list rets))
126            (t                   e))))
127
128  (defun pr-makers(local e serp)
129    (cond ((null-p e) nil)
130          ((atom-p e)
131           (make-slist :expr
132                       (list '$a$xreturn-many (true-zero local) e)))
133          ((cons (make-slist-t (list (if local 'lreturn 'return) (car e)) serp)
134                 (pr-makers local (cdr e) serp)))))
135
136  (defun pcheck-length(e minsize maxsize)
137    (unless (full-list-p e)
138            (al-perror "Complete expression expected in: ~A" e))
139    (let ((s (length e)))
140      (cond ((< s minsize)
141             (al-perror "Incomplete expression in: ~A" e))
142            ((> s maxsize)
143             (al-perror "Overcomplete expression in: ~A" e)))))
144
145  (defun prep-set(e gok serp)             (declare (ignore serp))
146    (unless (= (length e) 2) (al-perror "set expects two arguments in: ~A" e))
147    (let ((bl (x-sacc (car e) 'x644824))
148          (v  (prep (cadr e) gok)))
149      (cond ((symbol-p (car e)) (list 'set (car e) v))
150            ((list `(mu (x644824) (lreturn (block1 x644824 ,@bl)))
151                   v)))))
152
153  (defun x-sacc(e how)
154    (cond ((null e)     nil)
155          ((symbol-p e) (list (list 'set e how)))
156          ((cons-p e)   (append (x-sacc (car e) `(car ,how))
157                                (x-sacc (cdr e) `(cdr ,how))))
158          ((al-perror "List structure of names expected in: ~A" e))))
159
160  (defun check-fdef(fdef)
```

```
161    (cond ((and (symbol-p (car fdef)) (al-args-p (cadr fdef)))
162           (cons (car fdef) (cons (cadr fdef) (cddr fdef))))
163          ((and (cons-p (car fdef)) (symbol-p (caar fdef)) (al-args-p (cdar fdef)))
164           (cons (caar fdef) (cons (cdar fdef) (cdr fdef))))
165          ((al-perror "Function definition expected in: ~A" fdef))))
166
167
168  (defun al-args-p(args)
169    (cond ((symbol-p args))
170          ((not (cons-p args))   nil)
171          ((al-args-p (car args)) (al-args-p (cdr args)))))
172
173
174  ;-----------------------------------------------------------------------------
175  ; Preprocessing of letrec and letrecs expressions.
176
177  (defun prep-letrecs(e serp)
178    (prep-letrecx e 'lets serp nil))
179
180  (defun prep-letrec(e serp)
181    (prep-letrecx e 'let serp nil))
182
183  (defun prep-gletrecs(e serp)
184    (prep-letrecx e 'lets serp t))
185
186  (defun prep-gletrec(e serp)
187    (prep-letrecx e 'let serp t))
188
189  (defun prep-letrecx(e inlet serp genp)
190    (let* ((parbind (slist-p (cadr e)))
191           (bind    (rm-slist (cadr e)))
192           (args    (plx-args bind))
193           (inflet `(,inlet ,(make-slist-t (plx-aparset bind) parbind) ,@(cddr e))))
194      (make-slist-t `(let ,args
195                       ,(make-gen-t (make-slist-t inflet serp) genp)) serp)))
196
197  (defun plx-aparset(e)
198    (cond ((null-p e) nil)
199          ((symbol-p  (car e))       (plx-aparset (cdr e)))
200          ((list-l1-p (cdr (car e))) (cons (list (caar e) `(set ,(caar e) ,(cadar e)))
201                                           (plx-aparset (cdr e))))
202          ((list-l1-p (car e))       (plx-aparset (cdr e)))
203          ((al-perror "Binding in <let> is not a list of one or two elements: ~A" (car
 e)))))
204
205  ;-----------------------------------------------------------------------------
206  ; Preprocessing of let and lets expressions.
207
208  (defun prep-let(e)
209    (prep-letx e 'let))
210
211  (defun prep-lets(e)
212    (prep-letx e 'lets))
213
214  (defun prep-glet(e)
```

```
215    (prep-letx e 'glet))
216
217  (defun prep-letx(e kind)
218    (let ((parb t) (parp t) (body nil) (bind nil))
219      (when (slist-p e) (setq parb nil) (setq e (slist-expr e)))
220      (setq e (cdr e))
221      (setq bind (car e))
222      (when (slist-p bind) (setq parp nil) (setq bind (slist-expr bind)))
223      (setq body (cdr e))
224      (let ((mbody (plx-body body kind parb))
225            (margs (plx-args bind))
226            (mpars (plx-pars bind)) (mcall nil) (mfun nil))
227        (setq mfun (cons 'mu (cons margs mbody)))
228        (unless parb (setq mfun (make-slist :expr mfun)))
229        (setq mcall (cons mfun mpars))
230        (unless parp (setq mcall (make-slist :expr mcall)))
231        mcall)))
232
233  (defun plx-body(e kind parb)
234    (case kind
235      ('lets (if e (pls-body e)               nil))
236      ('let  (if e (pl-body e (not parb) nil) '(lreturn)))
237      ('glet (if e (pl-body e (not parb)  t)  '(lreturn)))))
238
239  (defun pls-body(e)
240    (cond ((null-p e) nil)
241          ((atom-p e) (al-perror "Unexpected end of <lets> as: ~A" e))
242          ((cons (car e) (pls-body (cdr e))))))
243
244  (defun plx-args(e)
245    (cond ((null-p e) nil)
246          ((atom-p e) (al-perror "Unexpected end of bindings in <let> as: ~A" e))
247          ((cons-p   (car e)) (cons (caar e) (plx-args (cdr e))))
248          ((symbol-p (car e)) (cons (car  e) (plx-args (cdr e))))
249          ((al-perror "Binding in <let> is not a list of two elements: ~A" (car e)))))
250
251  (defun plx-pars(e)
252    (cond ((null-p e) nil)
253          ((symbol-p  (car e))       (cons '%non-value (plx-pars (cdr e))))
254          ((list-l1-p (cdr (car e))) (cons (cadar e) (plx-pars (cdr e))))
255          ((list-l1-p (car e))       (cons '%non-value (plx-pars (cdr e))))
256          ((al-perror "Binding in <let> is not a list of one or two elements: ~A" (car
 e)))))
257
258  ;-----
259  ; Preprocessing body of let.
260
261  (defun pl-body(e serb gen)
262    (unless (full-list-p e) (al-perror "Unexpected end of <let> as: ~A" e))
263    (when (null e) (setq e '(())))
264    (let ((lst (car (last e))) (blst (butlast e)))
265      (let ((x (outer-gen-p blst)))
266        (when x (al-perror "Unexpected generator: ~A" x)))
267      (if blst (let* ((the-mu
268                       (make-slist-t `(mu dumb-2645
```

```
269                                        ,(make-rep :expr (list 'lreturn lst)))
270                                   serb))
271                       (the-call
272                         (make-gen-t (make-slist-t (cons the-mu blst) serb) gen)))
273                 (list (make-rep-t (make-slist-t
274                                    (list 'lreturn the-call) serb) gen)))
275         (list (make-rep :expr (list 'lreturn lst))))))
276
277  ;          (the-call (make-gen-t (make-slist-t (cons the-mu blst) serb) gen)))
278  ;      (list (make-rep-t (make-slist-t (list 'lreturn the-call) serb) gen)))))
279
280  (defun outer-gen-p(e)
281    (cond ((xcons-p e) (or (outer-gen-p (xcar e)) (outer-gen-p (xcdr e))))
282          ((gen-p e) t)
283          ((rep-p e) nil)
284          ((atom  e) nil)))
285
286
287  ;-------------------------------------------------------------------------------
288  ; Testing
289
290  (defun rpp-file(fname)
291    (let ((f (my-open-read-file fname)))
292      (unless f (al-error "Cannot open file for reading: ~A" fname))
293      (prog (e)
294        start
295        (setq e (al-read f)) (when (eq e al-eof) (go exit))
296        (pprint "<--") (al-print e)
297        (setq e (al-prep e)) (when (eq e al-f)   (go start))
298        (pprint "-->") (al-print e) (go start)
299        exit
300        (close f))
301      t))
302
303  (defun make-slist-t(e ser)
304    (if ser (make-slist :expr e) e))
305
306  (defun make-gen-t(e gen)
307    (if gen (make-gen :expr e) e))
308
309  (defun make-rep-t(e rep)
310    (if rep (make-rep :expr e) e))
311
312  (defun rm-slist(e)
313    (if (slist-p e) (slist-expr e) e))
```

A.3.2 gprep.l

Pre-processing of generator/replicator related forms:

```
;-------------------------------------------------------------------------------
; Generator and Replicator Preprocessor.

(defvar trace-prep nil "Trace preprocessor out put if true.")


;-------------------------------------------------------------------------------
; Main call for generator/replicator preprocessing.

(defun gprep(e)
  (reset-gen-temp)
  (let ((final (gprep1 `(list ,e))))
    (if (cons-p (cdr final))
        (setq final (cadr final)))
    (when trace-prep (pprint final))
    final))

(defun gprep1(e)
  (cond ((cons-p  e)                      (gprep-list nil e))
        ((scons-p e) (setf (slist-expr e)  (gprep-list t   (slist-expr e))) e)
        ((gen-p   e) (setf (gen-expr   e)  (gprep1         (gen-expr   e))) e)
        ((rep-p   e) (al-perror "Unexpected Replicator at: ~A" e))
        (e)))

(defun check-sharp!(l)
  (when (and (cons-p (rep-expr l)) (eql (car (rep-expr l)) '$a$sharp!))
        (let ((et l))
          (setq et (cadr (rep-expr et)))
          (if (and (cons-p et) (member (car et) '(return lreturn))
                   (gen-p (cadr et)))
              (setf (gen-expr (cadr et))
                    `(passnever ,(make-slist-t
                                  `($a$directg ,(when (eql (car et) 'return) 0)
                                    ,@(rm-slist (gen-expr (cadr et))))
                                  (slist-p (gen-expr (cadr et))))))
            (setq et (make-gen :expr `(passnever ,et))))
          (setf (rep-expr l) et))
        t))

(defun gprep-list(serp l)
  (cond ((cons-p l)
         (if (rep-p (car l))
             (let ((e (pop l)) sh)
               (setq sh (check-sharp! e))
               (setq e (gprep1 (rep-expr e)))
               (gprep-rep serp e (gprep-list serp l) sh))
           (cons (gprep1 (pop l)) (gprep-list serp l))))
        ((null   l) nil)
        ((gprep1 l))))

;_____
; Found replicator in front of e followed by rest.
```

```
53
54  (defun gprep-rep(serp e rest &optional eager)
55    (setq e (list e))
56    (let ((gens (remove-gens e)))
57      (setq e (car e))
58      (setq gens (simpl-gens gens))
59      (let* ((names  (sg-names gens))
60             (cl `(lambda (l) (replace-syms (quote ,e) (make-blist (quote ,names) l))))
61             (gexec  (sg-exec-m gens))
62             (exrest (if (null rest) nil
63                       (make-slist-t (cons 'list rest) serp)))
64             (fres (make-rlist
65                     :expr `($a$rep-driver ,(true-zero serp) (quote ,cl)
66                                           ,(if eager (list 'eager gexec) gexec)
67                                           (quote ,exrest)))))
68        fres)))
69
70  (defun sg-names(gens)
71    (cond ((null gens) nil)
72          ((eq (car gens) '$sym$) (cadr gens))
73          (t (when (atom (car gens)) (pop gens))
74             (cons (sg-names (car gens)) (sg-names (cdr gens))))))
75
76  (defun sg-exec-m(gens)
77    (case (car gens)
78      ($sym$           (car (sg-exlist (list gens))))
79      (($ser$ $par$)
          (let ((gexec (sg-exlist (cdr gens)))
81              (serp (case (car gens) ($ser$ t) ($par$ nil))))
82                `($a$mgen-call ,(true-zero serp) ,@gexec)))
83      (otherwise (al-perror "Unkown. Generator preprocessing failed."))))
84
85  (defun sg-exlist(gens)
86    (cond ((null gens)                  nil)
87          ((eq (caar gens) '$sym$)
88           (cons (caddar gens)          (sg-exlist (cdr gens))))
89          ((cons (sg-exec-m (car gens)) (sg-exlist (cdr gens))))))
90
91  (defun simpl-gens(gens)
92    (cond ((null gens)            nil)
93          ((eq (car gens) '$sym$) gens)
94          ((let ((r (simpl-glist (car gens) (cdr gens))))
95             (case (length r)
96               (0         nil)
97               (1         (al-error "simpl-gens"))
98               (2         (cadr r))
99               (otherwise r))))))
100
101 (defun simpl-glist(mode l)
102   (when l (let ((sg1 (simpl-gens  (car l)))
103                 (sgr (simpl-glist mode (cdr l))))
104             (cond ((and (null sg1) (null sgr)) nil)
105                   ((null sg1) sgr)
106                   ((null sgr) sg1)
107                   ((and (eq mode (car sg1)) (eq mode (car sgr)))
```

```
108                    (append sg1 (cdr sgr)))
109                   (t
110                    (unless (eq mode (car sg1)) (setq sg1 (list mode sg1)))
111                    (append sg1 (if (eq mode (car sgr))
112                                    (cdr sgr) (list sgr))))))))
113
114  (defun remove-gens(e)
115    (rg-main e))
116
117  (defun rg-main(e)
118    (cond ((cons-p   e) (cons '$par$ (rg-list e)))
119          ((scons-p  e) (cons '$ser$ (rg-list (slist-expr e))))))
120
121  (defun rg-list(e)
122    (when (cons-p e)
123      (cons (if (gen-p (car e))
124                (let ((ece (gen-expr (car e)))
125                      (s (gen-temp)))
126                  (setf (car e) s)
127                  (list '$sym$ s (rem-from-gen ece)))
128              (rg-main (car e)))
129            (if (gen-p (cdr e))
130                (let ((ece (gen-expr (cdr e)))
131                      (s (gen-temp)))
132                  (setf (cdr e) s)
133                  (list '$sym$ s (rem-from-gen ece)))
134              (rg-list (cdr e))))))
135
136  (defun rem-from-gen(ingen)
137    (cond ((xcons-p ingen)
138           (remove-ggens ingen))
139          ((remove-ggens `(passtrue ,ingen)))))
140
141
142  (defun remove-ggens(e)
143    (let ((serp (slist-p e)) gens)
144      (setq gens (simpl-gens (rg-main e)))
145      (when serp (setq e (slist-expr e)))
146      (if (null gens) `($a$gen-call-gq ,(true-zero serp) ,@e)
147        (let* ((names  (sg-names gens))
148               (cl `(lambda (l) (replace-syms (quote ,(make-slist-t e serp) )
149                                              (make-blist (quote ,names) l))))
150               (gexec  (sg-exec-m gens))
151               (fres `($a$gen-driver ,(true-zero serp) (quote ,cl) ,gexec)))
152          fres))))
153
154  ;-------------------------------------------------------------------------------
155  ; Utilities
156
157  (defun chcar-of(e)
158    (function (lambda (x) (rplaca e x))))
159
160  (defun chcdr-of(e)
161    (function (lambda (x) (rplacd e x))))
162
```

```
163  (defvar gen-temp-v 0)
164
165  (defun gen-temp()
166    (intern (sformat "$a$-~A" (incf gen-temp-v))))
167
168  (defun reset-gen-temp()
169    (setq gen-temp-v 0))
170
171  (defun true-zero(c)
172    (when c 0))
```

A.3.3 oprep.l

Pre-processing of object declarations:

```
 1  ;-------------------------------------------------------------------------------
 2  ; Object Oriented Programming. Preprocessor hooks.
 3
 4  ;-------------------------------------------------------------------------------
 5  ; Main call for object oriented preprocessing.
 6
 7  (defun oprep(e) (oprep1 e))
 8
 9  (defun oprep1(e)
10    (cond ((xcons-p e)
11           (case (xcar e)
12             (class-start (oprep-cs (cdr (rm-slist e))))
13             (class-end   (oprep-ce (cdr (rm-slist e))))
14             (otherwise   e)))
15          (e)))
16
17  (defun oprep-ce(defs)
18    (unless (cons-p defs)         (al-perror "Name of class expected in car of: ~A"
defs))
19    (unless (symbol-p (car defs)) (al-perror "Name of class expected in: ~A" (car
defs)))
20    (let ((cname (pop defs)))
21      (when defs (al-perror "Unknown features of class in: ~A" defs))
22      (list '$a$class-end cname)))
23
24  (defun oprep-cs(defs)
25    (unless (cons-p defs)         (al-perror "Name of class expected in: ~A" defs))
26    (unless (symbol-p (car defs)) (al-perror "Name of class expected in: ~A" (car
defs)))
27    (let ((cname (pop defs)) (description nil) (tmp nil))
28  ;    (take-item 'inherit)
29      (take-item 'common)
30      (take-item 'static)
31      (take-item 'dynamic)
32      (take-item 'import)
33      (take-item 'part)
34      (take-item 'methods)
35      (take-interfaces)
36      (when defs (al-perror "Unexpected feature of class ~A in: ~A" cname defs))
37      (list '$a$class-start cname (reverse description))))
38
39  (defmacro take-interfaces()
40    '(let ((d1 description))
41       (setq description nil)
42       (take-item 'interface)
43       (while (cdar description) (take-item 'interface))
44       (pop description)
45       (push `(interface all-methods ,@(cdr (assoc 'methods d1))) description)
46       (setq description (cons (cons 'interfaces (join-inters description)) d1))))
47
48  (defun join-inters(dl)
49    (when dl (cons (cdar dl) (join-inters (cdr dl)))))
```

```
50
51  (defmacro take-item(type)
52    `(progn (setq defs (cdr (setq tmp (assoc-rmf defs ,type))))
53            (push (car tmp) description)
54            (check-symlist (car tmp))))
55
56  (defun check-symlist(l)
57    (cond ((null   l))
58          ((atom-p l)              (al-perror "Unexpected end of list in: ~A" l))
59          ((not (symbol-p (car l))) (al-perror "Symbol expected in: ~A" (car l)))
60          ((check-symlist (cdr l)))))
61
62  ;_____
63  ;Find key in al. Create it if it does not exist and assoc it with nil.
64
65  (defun assoc-rmf(al key)
66    (let ((v (assoc-rm al key)))
67      (if (car v) v (cons (cons key nil) (cdr v)))))
68
69  (defun assoc-rm(al key)
70    (let* ((val (cons nil nil))
71           (al2 (assoc-rm-v al key val)))
72      (cons (car val) al2)))
73
74  (defun assoc-rm-v(al key val)
75    (cond ((null   al)               nil)
76          ((atom-p al)               (al-perror "End of list expected in: ~A" al))
77          ((atom-p (car al))         (al-perror "List expected in: ~A" (car al)))
78          ((not (symbol-p (caar al))) (al-perror "Symbol expected in: ~A" (caar al)))
79          ((eq (caar al) key)        (setf (car val) (car al)) (cdr al))
80  ;       (al)))
81          ((cons (car al) (assoc-rm-v (cdr al) key val)))))
82
```

A.4 Utilities

A.4.1 io.l

Input output utilities:

```
 1  ;-------------------------------------------------------------------------------
 2  ; Input/Output of ALLOY expressions.
 3
 4  (defstruct (rep)
 5    "A Repicator Data Structure"
 6    (expr nil))
 7
 8  (defstruct (gen)
 9    "A generator Data Structure"
10    (expr nil))
11
12  (defstruct (slist)
13    "A Square List"
14    (expr nil))
15
16  (defstruct (rlist)
17    "A Round List"
18    (expr nil))
19
20  (defmacro scons-p(e)
21    `(slist-p ,e))
22
23  (defun s-list(&rest args)
24    (make-slist :expr args))
25
26  (defconstant spc   (character '| |))
27  (defconstant excla (character '|!|))
28  (defconstant opar  (character '|(|))
29  (defconstant cpar  (character '|)|))
30  (defconstant ospar (character '|[|))
31  (defconstant cspar (character '|]|))
32  (defconstant power (character '|^|))
33  (defconstant sharp (character '|#|))
34  (defconstant star  (character '|*|))
35  (defconstant dot   (character '|.|))
36  (defconstant quo   (character '|'|))
37
38  (defun get-cl-table()
39    (eval '(copy-readtable nil)))
40
41  (defconstant cl-table (get-cl-table))
42  (defconstant al-table (get-cl-table))
43
44  (defun makenospec()
45    (let ((n 255) (aa (character '|A|)))
46      (while (>= n 0) (reset-mc (int-char n) aa) (decf n))))
47
48  (defun reset-mc(c &optional from)
49    (set-syntax-from-char c (if from from c) *readtable* cl-table))
50
```

```
(defun make-al-table()
  (setq *readtable* al-table)
  (makenospec)
  (reset-mc #\') (reset-mc #\")  (reset-mc #\;) (reset-mc #\|) (reset-mc #\\)
  (reset-mc #\Newline) (reset-mc #\Return) (reset-mc #\Space) (reset-mc #\Tab)
  (set-macro-character quo   'read-quo t)
  (set-macro-character dot   'read-dot t)
  (set-macro-character power 'read-power nil)
  (set-macro-character star  'read-star nil)
  (set-macro-character sharp 'read-sharp nil)
  (set-macro-character opar  'read-opar)
  (set-macro-character ospar 'read-ospar)
  (set-macro-character cpar  'read-cpar)
  (set-macro-character cspar 'read-cspar)
  (setq *readtable* (get-cl-table)))

(make-al-table)

(defun read-pass(stream char)
  (declare (ignore stream))
  char)

(defun read-power(stream char)
  (declare (ignore char))
  (let ((c (read-char stream nil spc nil)) e)
    (unless (eql c excla) (unread-char c stream))
    (setq e (list 'mu () (make-rep :expr (list 'lreturn (read-one stream)))))
    (if (eql c excla) (make-slist :expr e) e)))

(defun read-sharp(stream char)
  (declare (ignore char))
  (let ((c (read-char stream nil spc nil)) e)
    (unless (eql c excla) (unread-char c stream))
    (setq e (read-one stream))
    (make-rep :expr (if (eql c excla) `($a$sharp! ,e) e))))

(defun read-star(stream char)
  (declare (ignore char))
  (make-gen :expr (read-one stream)))

(defun read-quo(stream char)
  (declare (ignore char))
  (list 'quote (read-one stream)))

(defun read-opar(stream char)
  (declare (ignore char))
  (al-read-list stream cpar cspar))

(defun read-ospar(stream char)
  (declare (ignore char))
  (let ((e (al-read-list stream cspar cpar)))
    (if e (make-slist :expr e) nil)))

(defun al-read-list(stream end other)
  (let ((e (read-any stream)))
```

```
106     (cond ((eq e end)   nil)
107           ((eq e other) (throw 'al-read "Unmached parenthesis."))
108           ((eq e dot)   (let ((le1 (read-one stream))
109                               (le2 (read-any stream)))
110                           (when (eq other le2)
111                             (throw 'al-read "Mismathed Parenthesis."))
112                           (unless (eq end le2)
113                             (throw 'al-read "Parenthesis expected."))
114                           le1))
115           ((cons e (al-read-list stream end other))))))
116
117 (defun read-dot(stream char)
118   (declare (ignore stream))
119   char)
120
121 (defun read-cpar(stream char)
122   (declare (ignore stream))
123   char)
124
125 (defun read-cspar(stream char)
126   (declare (ignore stream))
127   char)
128
129 (defun read-one(stream)
130   (let ((e (read-any stream)))
131     (when (or (eq e cpar) (eq e cspar))
132       (throw 'al-read "Unexpected closing Parenthesis."))
133     e))
134
135 (defun read-any(stream)
136   (let ((e (read stream nil al-f nil)))
137     (when (eq e al-f) (throw 'al-read "Unexpected End of Input."))
138     e))
139
140 ;-----
141 ; Main reader function.
142
143 (defvar al-readers 0 "How many processes read at the moment.")
144
145 (defun al-read(&optional stream quiet)
146   (when (null stream) (setq stream *standard-input*))
147   (let ((expr nil))
148     (when (= (incf al-readers 1) 1) (setq *readtable* al-table))
149     (setq expr (catch 'al-read (list (read stream nil al-eof nil))))
150     (when (= (decf al-readers 1) 0) (setq *readtable* cl-table))
151     (if (consp expr) (car expr)
152       (progn (unless quiet (p-error expr)) al-f))))
153
154 ;-----
155 ; Main printer function.
156
157 (defun al-print(e &optional stream)
158   (when (null stream) (setq stream *standard-output*))
159   (cond ((numberp e) (princ e stream))
160         ((symbolp e) (princ e stream))
```

```
161         ((stringp e) (princ e stream))
162         ((cons-p  e) (al-print-list e opar cpar stream))
163         ((slist-p e) (al-print-list (slist-expr e) ospar cspar stream))
164         ((rep-p   e) (princ '|#|  stream) (al-print (rep-expr e) stream))
165         ((gen-p   e) (princ '|*|  stream) (al-print (gen-expr e) stream))
166         ((alval-p e) (princ '|%w| stream))
167         ((eq al-f e) (princ '|%f| stream))
168         ((ext-function-p e) (princ '|%(object of class CLOSURE)| stream))
169         ((class-p e) (princ '|%(class | stream)
170                      (princ (class-name e) stream ) (princ '|)| stream))
171         ((object-p e) (princ '|%(object of class | stream)
172                       (princ (object-name e) stream ) (princ '|)| stream))
173         (t (princ '|%u| stream))))
174
175  (defun al-print-list(e op cl stream)
176    (cond ((null e) (princ '|nil| stream))
177          (t        (princ op stream)
178                    (al-print (car e) stream) (al-print-listx (cdr e ) stream)
179                    (princ cl stream))) t)
180
181  (defun al-print-listx(e stream)
182    (cond ((null-p e) nil)
183          ((cons-p e) (princ '| | stream) (al-print (car e) stream)
184                      (al-print-listx (cdr e) stream))
185          (t          (princ '| . | stream) (al-print e stream))))
```

A.4.2 values.l

Local and global environment handling functions:

```
;-------------------------------------------------------------------------------
; Local and global environment handling.

(defconstant hassvstr '**1*noval*1**)

(defun get-val(var env tab &optional tab2)
  (let ((v nil))
    (cond ((setq v (get-val-env var env))
           (cdr v))
          ((and tab2 (not (eq (setq v (has-sym-value tab2 var)) hassvstr))) v)
          ((get-sym-value tab var)))))

(defun set-val(var val env tab)
   (if (sassg-p var) (let ((v (get-val var env tab)))
                       (when (alset-alval val v) t))
                     (let ((v (get-val-env var env)))
                       (if v (setf (cdr v) val)
                             (set-sym-value tab var val)) t)))

(defun get-val-env(var e)
  (cond ((null e) nil)
        ((assoc var (car e)))
        ((get-val-env var (cdr e)))))

(defconstant char-und (char "____" 1))

(defmacro massg-p(varname)
  `(eq (char (string ,varname) 0) char-und))

(defmacro sassg-p(varname)
  `(not (massg-p ,varname)))


; ALLOY local environments.

(defvar pb-bs nil "Contains bind list.")
(defvar pb-ss nil "Contains set list.")

(defun new-env(vars vals e)
  (setq pb-bs nil) (setq pb-ss nil)
  (new-env2 vars vals)
  (cons (when pb-ss (cons 'list pb-ss))
        (cons pb-bs e)))

(defun makev-fail(vars)
  (cond ((null     vars))
        ((symbol-p vars) (push (cons vars al-f) pb-bs))
        ((cons-p   vars) (makev-fail (car vars)) (makev-fail (cdr vars)))
        ((al-error "Invalid parameters in: ~A" vars))))

(defun new-env2(vars vals)
  (cond ((null vars))
```

```
53          ((symbol-p vars)              ; end of vars is a var
54           (new-bind vars vals))
55          ((null vals) (makev-fail vars))
56          ((atom-p vals)                ; end of vals is a val
57           (struct-bind vars vals))
58          ((cons-p vars)
59           (new-env2 (car vars) (car vals))
60           (new-env2 (cdr vars) (cdr vals)))
61          ((al-error "Invalid parameters in: ~A" vars))))
62
63  (defun new-bind(name val)
64    (push (cons name (if (eq val al-n) (make-alval) val)) pb-bs))
65
66  (defun struct-bind(l how)
67    (cond ((null l))
68          ((symbol-p l) (let ((var (make-alval)))
69                          (push (cons l var) pb-bs)
70                          (push (list '$a$set-v1 var how) pb-ss)))
71          ((cons-p l) (struct-bind (car l) `(car ,how))
72                      (struct-bind (cdr l) `(cdr ,how)))
73          ((al-error "Invalid parameter structure in: ~A" l))))
74
75
76  ;-----------------------------------------------------------------------------
77  ; Hash tables.
78
79  (defun make-sym-table()
80    (make-hash-table :test 'eq :size 10))
81
82  (defun has-sym-value(sym-table name)
83     (gethash name sym-table hassvstr))
84
85  (defun has-sym-value-p(sym-table name)
86     (not (eq (gethash name sym-table hassvstr) hassvstr)))
87
88  (defun get-sym-value(sym-table name)
89     (let ((hval (gethash name sym-table hassvstr)))
90        (if (eq hval hassvstr) (set-sym-valuec sym-table name
91                                               (if (massg-p name) nil (make-alval)))
92                               hval)))
93
94  (defun set-sym-value(sym-table name value)
95    (if (has-sym-value-p sym-table name)
96        (set-sym-valuex sym-table name value)
97      (set-sym-valuec sym-table name value)))
98
99  ; name is known not to exist in table. Must be created.
100 (defun set-sym-valuec(sym-table name value)
101   (if create-vars (setf (gethash name sym-table) value)
102     (progn (p-warning "Cannot define new variable: ~A" name)
103            al-f)))
104
105 ; name is known to exist in table.
106 (defun set-sym-valuex(sym-table name value)
107   (setf (gethash name sym-table) value))
```

```
108
109 ;-------------------------------------------------------------------------------
110 ; Interpreters special function table.
111
112 (defun get-spec-value(name)
113   (let ((hval (gethash name spec-table al-t)))
114      (if (eq hval al-t) nil (list hval))))
115
116 (defun set-spec-value(name value)
117   (setf (gethash name spec-table) value))
118
119 (defun add-spec(name value)
120   (set-spec-value name value))
121
122 (defun add-specs(l)
123   (when l (add-spec (caar l) (cdar l))
124           (add-specs (cdr l))))
```

A.4.3 alutil.l

This file contains utilities more specific to ALLOY:

```
;------------------------------------------------------------------------------
; General utilities to ALLOY interpreter.

;------------------------------------------------------------------------------
; Special values.

(defconstant al-eof '***end-of-file*** "End of file starting ALLOY Reading.")

(defstruct (al-failure))
(defstruct (al-success))
(defstruct (al-nonvalue))
(defconstant al-f (make-al-failure))
(defconstant al-t (make-al-success))
(defconstant al-n (make-al-nonvalue))

(defconstant lazyd '(t))
(defconstant lazyt  t)
(defconstant lazy1  nil)

(defun lazy-delay-p(e)
  (consp e))

(defun lazy-delay1-p(e)
  (eq e 'delay))

(defstruct (eerror)
  (expr "Some error occured."))


;------------------------------------------------------------------------------
; Utilities.

(defun strip-alval(e)
  (cond ((and (alval-p e) (alval-init e))
         (setf (alval-value e) (strip-alval (alval-value e))))
        (e)))

(defun strip-alval-e(e)
  (cond ((consp   e) (setf (car e) (strip-alval-e (car e)))
                     (setf (cdr e) (strip-alval-e (cdr e))))
        ((slist-p e) (setf (slist-expr e) (strip-alval-e (slist-expr e))))
        ((let ((es (strip-alval e)))
           (unless (eql es e) (setq e (strip-alval-e es))))))
  e)

(defun strip-alval-l(e)
  (cond ((consp   e) (setf (car e) (strip-alval (car e)))
                     (setf (cdr e) (strip-alval-l (cdr e))))
        ((slist-p e) (setf (slist-expr e) (strip-alval-l (slist-expr e))))
        ((let ((es (strip-alval e)))
           (unless (eql es e) (setq e (strip-alval-l es))))))
  e)
```

```
53
54  (defun ground-p(e)
55    (cond ((xcons-p e) (and (ground-p (xcar e)) (ground-p (xcdr e))))
56          ((and (alval-p e) (not (alval-init e))) nil)
57          (t)))
58
59  (defun member-ng(l)
60    (cond ((xcons-p l) (or (member-ng (xcar l)) (member-ng (xcdr l))))
61          ((and (alval-p l) (not (alval-init l))) l)))
62
63  ;-----
64  ; Macros
65
66  (defmacro noteq(x y)
67    `(not (eq ,x ,y)))
68
69  (defun compile-all-alloy()
70    (shell (str-app4 "cat safe.l util.l alutil.l prep.l gprep.l oprep.l "
71                     "load.l values.l io.l inter.l "
72                     "fcall.l gcall.l rcall.l ocall.l "
73                     "build.l build2.l obuild.l >> allal.l"))
74    (compile-file "allal.l"))
75
76  (defun compile-all-alloy-f()
77    (shell (str-app4 "cat fast.l util.l alutil.l prep.l gprep.l oprep.l "
78                     "load.l values.l io.l inter.l "
79                     "fcall.l gcall.l rcall.l ocall.l "
80                     "build.l build2.l obuild.l >> allal.l"))
81    (compile-file "allal.l"))
82
83  (defun compile-alloy()
84    (mapc 'compile-file '("util.l" "alutil.l" "prep.l" "gprep.l"
85                          "oprep.l" "load.l"
86                          "values.l" "io.l" "inter.l" "fcall.l" "gcall.l"
87                          "rcall.l" "ocall.l" "build.l" "build2.l" "obuild.l")))
88
89  (defun load-calloy()
90    (mapc 'load '("util" "alutil" "prep" "gprep" "oprep" "load"
91                  "values" "io" "inter" "fcall" "gcall"
92                  "rcall" "ocall" "build" "build2" "obuild")))
93
94  (defun compile-all-alloy-gcl()
95  ;  (sys:dos "makesafe")
96    (compile-file "allal.l"))
97
98  (defun compile-all-alloy-f-gcl()
99  ;  (sys:dos "makefast")
100   (compile-file "allal.l"))
101
102 (defmacro make-proc-m(&rest args)
103   `(progn (incf proc-nosx) (make-proc ,@args)))
104
105 ; Create muinfo. Preserve counter.
106 (defmacro make-muinfo-m(&rest args)
107   `(progn (incf muinfo-nosx) (make-muinfo ,@args)))
```

```
108
109  (defmacro make-alval-m(&rest args)
110    `(progn (incf alval-nosx) (make-alval ,@args)))
111
112  (defun xcons-p(e)
113    (or (cons-p e) (scons-p e)))
114
115  (defun xcar(e)
116    (cond ((cons-p e)  (car e))
117          ((scons-p e) (car (slist-expr e)))
118          ((al-error "xcar expected cons or scons in ~A" e))))
119
120  (defun xcdr(e)
121    (cond ((cons-p e)  (cdr e))
122          ((scons-p e) (let ((e (cdr (slist-expr e))))
123                         (if e (make-slist :expr e) nil)))
124          ((al-error "xcdr expected cons or scons in ~A" e))))
125
126
127  (defvar rm-dot-t nil "True if last rm-dot-list acted.")
128
129  (defun rm-dot-list(e)
130    (setq rm-dot-t nil)
131    (rm-dot-list1 e))
132
133  (defun rm-dot-list1(e)
134    (cond ((null   e) nil)
135          ((atom   e) (setq rm-dot-t t)
136                      (list (if (rlist-p e) (rlist-expr e) e)))
137          ((cons-p e) (cons (car e) (rm-dot-list1 (cdr e))))))
138
139  (defun nput-dot-list-nc(e)
140    (when rm-dot-t
141      (cond ((cddr e) (nput-dot-list-nc (cdr e)))
142            ((setf (cdr e) (cadr e))))))
143
144  (defun p-error(&rest msgs)
145    (terpri) (princ "ALLOY ERROR -- ")
146    (princ (apply 'sformat msgs) *error-output*)
147    (terpri) nil)
148
149  (defun p-errorc(&rest msgs)
150    (princ "            -- ")
151    (princ (apply 'sformat msgs) *error-output*)
152    (terpri) nil)
153
154  (defun p-warning(&rest msgs)
155    (terpri) (princ "ALLOY WARNING -- ")
156    (princ (apply 'sformat msgs) *error-output*)
157    (terpri) nil)
158
159  (defun p-warningc(&rest msgs)
160    (princ "              -- ")
161    (princ (apply 'sformat msgs) *error-output*)
162    (terpri) nil)
```

```
163
164  (defun al-error(str &rest msgs)
165    (apply 'error (cons (sformat "~A~A" "ALLOY ERROR -- " str) msgs))
166    (terpri) nil)
167
168
169
170
171
172
```

A.4.4 util.l

This file contains general utilities used by the interpreter:

```
;------------------------------------------------------------------------------
; General Utilities for lisp.

(defun loadl(fname)
  (load (format nil "~A~A" fname ".l")))

(defmacro ll(fname)
  `(loadl (quote ,fname)))

(defmacro loadq(fname)
  (load fname :verbose nil))

(defun atom-app(x y)
  (intern (format nil "~A~A" x y)))

(defun str-app4(x y z w)
  (format nil "~A~A~A~A" x y z w))

(defun sformat(&rest args)
  (apply 'format (cons nil args)))

(defmacro xreturn(value)
  `(return-from 'exit ,value))

(defmacro rpush(stack val)
  `(push ,val ,stack))

(defmacro set-list-end(l end)
  `(append ,l ,end))

(defstruct (queue (:constructor make-queue-simpl))
  (head nil)
  (tail nil))

(defun make-queue()
  (let ((q (cons nil nil)))
    (make-queue-simpl :head q :tail q)))

(defun queue-empty(q)
  (eq (queue-head q) (queue-tail q)))

(defun queue-nonempty(q)
  (not (eq (queue-head q) (queue-tail q))))

(defun queue-get(q)
  (when (queue-empty q) (al-error "Attempt to get from an empty queue."))
  (let ((x (cadr (queue-tail q))))
    (setf (queue-tail q) (cdr (queue-tail q)))
    x))

(defun queue-put(x q)
  (setf (cdr (queue-head q)) (cons x nil))
```

```
53    (setf (queue-head q) (cdr (queue-head q))) x)
54
55  (defmacro while(cond &rest args)
56    `(do ()
57           ((not ,cond))
58        ,@args))
59
60  (defmacro until(cond &rest args)
61     `(do ()
62            (,cond)
63         ,@args))
64
65  (defmacro cons-p(e)
66    `(consp ,e))
67
68  (defmacro atom-p(e)
69    `(atom ,e))
70
71  (defmacro null-p(e)
72    `(null ,e))
73
74  (defmacro number-p(e)
75    `(numberp ,e))
76
77  (defmacro symbol-p(e)
78    `(symbolp ,e))
79
80  (defun list-l1-p(e)
81    (and (cons-p e) (null-p (cdr e))))
82
83  (defun full-list-p(e)
84    (cond ((null e))
85          ((cons-p e) (full-list-p (cdr e)))))
86
87  (defun my-open-read-file(fname)
88    (open fname :direction :input :if-does-not-exist nil))
89
90  (defun my-open-write-file(fname)
91    (open fname :direction :output))
92
93  (defun my-close-file(fname)
94    (close (open fname)))
95
96  (defun p-msg(&rest msgs)
97    (princ (apply 'sformat msgs) *error-output*)
98    (terpri) nil)
99
100 (defun remainder(x y)
101   (* (cadr (multiple-value-list (floor (/ x y)))) y))
102
103 (defun last-p(e)
104   (cond ((cons-p e) (atom-p (cdr e)))
105         ((al-error "last-p: List expected in: ~A" e))))
106
107 (defun compilel-file(fname)
```

```
108   (compile-file (sformat "~A.l" fname))
109   (load fname))
110
111 (defun explode(s)
112   (when (stringp s)
113         (let ((l (length s)) (ls nil))
114           (while (>= (decf l) 0) (push (string (char s l)) ls))
115           ls)))
116
117 (defun implode(ls)
118   (format nil "~{~A~}" ls))
119
120 (defun someend(pred seq)
121   (cond ((not (cons-p seq))       seq)
122         ((funcall pred (car seq)) (list (car seq)))
123         ((someend pred (cdr seq)))))
```

A.5 Built in functions and objects

A.5.1 build.l

Utilities for the creation of built in expressions, declarations, and basic functions: built in functions:

```
;-------------------------------------------------------------------------------
; Built in functions of ALLOY.
; Basic Library

;-----
; Initialize structures.

(init)

;-------------------------------------------------------------------------------
; Utilities for function declarations.

(defun make-power(p-func)
   (make-ext-function :type 'pfunction :expr p-func))

(defun make-gfunction(args-body)
   (make-ext-function :type 'gfunction :expr args-body))

(defun make-gbfunction(args-body)
   (make-ext-function :type 'gbfunction :expr args-body))

(defun make-lfunction(args-body)
   (make-ext-function :type 'lfunction :expr args-body))

(defun make-argone(al-name)
   (make-ext-function :type 'argone :expr (atom-app 'cl- al-name)))

(defun make-noninst(al-name)
   (make-ext-function :type 'noninst :expr (atom-app 'cl- al-name)))

(defmacro set-in-table(val)
  `(set-sym-value (if export globale-tab global-tab) al-name ,val)
)

(defun set-argone(al-name &optional export)
  (set-in-table (make-argone al-name)))

(defun set-noninst(al-name &optional export)
   (set-in-table (make-noninst al-name)))

(defun set-power(al-name &optional export)
   (set-in-table (make-power   al-name)))

(defun set-global(al-name args-body &optional export)
   (set-in-table (make-gfunction args-body)))

(defun set-globalb(al-name args-body &optional export)
   (set-in-table (make-gbfunction args-body)))

```

```
(defun add-spec1(al-name)
   (add-spec al-name (atom-app 'spec- al-name)))

(defun global2globalb(al-name)
  (let ((args-body (strip-alval (get-sym-value tab al-name))))
    (set-globalb al-name (ext-function-expr args-body)))
  t)

;-------------------------------------------------------------------------------
; Build in Values.

(set-sym-value globale-tab '%non-value al-n)
(set-sym-value globale-tab '%t 't)
(set-sym-value globale-tab '%f al-f)

;-------------------------------------------------------------------------------
; special function declarations & noninstant declared elsewhere.

(add-spec '$a$funcall-vs  'al-funcall-vs)
(add-spec '$a$alfun-fail  'spec-alfun-fail)
(add-spec '$a$alfun-gfail 'spec-alfun-gfail)
(add-spec '$a$if-v        'spec-$a$if-v)
(add-spec '$a$set-v1      'spec-$a$set-v1)
(add-spec '$a$pass-v      'spec-$a$pass-v)

(add-spec 'quote         'spec-quote)
(add-spec 'function      'spec-function)
(add-spec 'set           'spec-set)
(add-spec 'mu            'spec-mu)
(add-spec 'lreturn       'spec-lreturn)
(add-spec 'return        'spec-return)
(add-spec 'when          'spec-when)
(add-spec '$a$when-v     'spec-$a$when-v)
(add-spec 'if            'spec-if)
(add-spec 'delay         'spec-delay)
(add-spec 'eager         'spec-eager)
(add-spec 'lazy          'spec-lazy)
(add-spec1 'unless)
(add-spec1 'fail)
(add-spec1 'lfail)
(add-spec1 '$a$no-result)
(add-spec1 '$a$reset1)
(add-spec1 '$a$setfun1)
(add-spec1 '$a$class-start)
(add-spec1 '$a$class-end)

; Generator specific.

(add-spec1 '$a$csuspended-gl)
(set-noninst '$a$gen-call-sr t)
(set-noninst '$a$gen-call t)
(set-noninst '$a$gen-next t)
(set-noninst '$a$gen-nextw t)
(set-noninst '$a$gen-results t)

```

```
105  (add-spec1 '$a$genlmakew)
106  (add-spec1 '$a$genlmake)
107  (add-spec1 '$a$genllist)
108  (add-spec1 '$a$gen-call-dl)
109
110  (set-noninst '$a$mgen-call-sr t)
111  (set-noninst '$a$mgen-call t)
112  (set-noninst '$a$mgen-next t)
113  (add-spec1   '$a$mgmaker)
114  (add-spec1   '$a$mgmaker-wait)
115  (add-spec1   '$a$mgmaker-res)
116  (add-spec1   '$a$mgreturn)
117
118  (add-spec1 '$a$gen-call-gq)
119  (add-spec1 '$a$rd-loop)
120  (add-spec1 '$a$rd-exec)
121
122  (add-spec1 '$a$ggen-next)
123  (add-spec1 '$a$gd-loop)
124  (add-spec1 '$a$gd-loop-cont)
125
126  (set-noninst '$a$rep-driver t)
127  (set-noninst '$a$gen-driver t)
128  (set-noninst '$a$gdgen-next t)
129
130  (set-argone  '$a$cgen-call t)
131  (set-noninst '$a$fgen-next t)
132  (add-spec    '$a$funcall-nomu 'al-funcall-nomu)
133  (set-noninst '$a$fcnmu-final t)
134  (set-noninst '$a$directg t)
135  (set-noninst '$a$dirg-final t)
136  (add-spec1 '$a$sharp!)
137
138
139  ;-------------------------------------------------------------------------------
140  ; Build in basic non-instant Functions.
141
142  (defmacro list-l1(l)
143    `(endp (cdr ,l)))
144
145  (set-noninst '$a$finish-eval t)
146  (defun cl-$a$finish-eval(arg)
147    (let ((val (strip-alval-e arg)))
148      (unless (and (symbol-p val) (eq val '$a$no-result))
149        (princ "==> ") (al-print val) (terpri))
150      (cons 'suc t)))
151
152  (set-noninst '$a$pass1 t)
153  (defun cl-$a$pass1(arg &rest args)              (declare (ignore args))
154    (cons 'suc arg))
155
156  (defun spec-unless()
157    (setq p-call `(when (not ,(cadr p-call)) ,@(cddr p-call)))
158    (next-the-same))
159
```

```
160  (set-noninst 'caar t)
161  (set-noninst 'cadr t)
162  (set-noninst 'cdar t)
163  (set-noninst 'cddr t)
164
165  (defun cl-caar(arg &rest args)              (declare (ignore args))
166    (let ((p (make-pprocess `(car (car ($a$pass-v ,arg))))))
167      (cons 'suc (proc-result p))))
168
169  (defun cl-cadr(arg &rest args)              (declare (ignore args))
170    (let ((p (make-pprocess `(car (cdr ($a$pass-v ,arg))))))
171      (cons 'suc (proc-result p))))
172
173  (defun cl-cdar(arg &rest args)              (declare (ignore args))
174    (let ((p (make-pprocess `(cdr (car ($a$pass-v ,arg))))))
175      (cons 'suc (proc-result p))))
176
177  (defun cl-cddr(arg &rest args)              (declare (ignore args))
178    (let ((p (make-pprocess `(cdr (cdr ($a$pass-v ,arg))))))
179      (cons 'suc (proc-result p))))
180
181  (set-globalb 'passtrue
182     '((x) (when x (return x))) t)
183
184  (set-globalb 'passnever
185     '(()) t)
186
187  (defun spec-$a$sharp!()
188    (setq p-call (cadr p-call))
189    (check-proc-call)
190    (next-the-same))
191
192  ;-------------------------------------------------------------------------------
193  ; argone system functions.
194
195  (defmacro condsval(&rest args)
196    `(let ((sval nil))
197       (cond ((alval-p (setq args (strip-alval args)))       (cons 'sus args))
198             ((atom-p  args)                                 (cons 'suc al-f))
199             ((alval-p (setq sval (strip-alval (car args)))) (cons 'sus sval))
200             ,@ args)))
201
202  (defmacro condsvar(&rest args)
203    `(let ((sval nil))
204       (cond ((alval-p (setq args (strip-alval args)))       (cons 'sus args))
205             ((atom-p  args)                                 (cons 'suc al-f))
206             ((prog1 nil (setq sval (strip-alval (car args)))))
207             ,@ args)))
208
209
210  (set-argone '$a$call-lisp t)
211  (defun cl-$a$call-lisp(args)
212    (condsval ((cons 'suc (eval sval)))))
213
214  (set-argone '$a$make-bi t)
```

```
215  (defun cl-$a$make-bi(args)
216    (condsval ((cons 'suc (global2globalb sval)))))
217
218  (set-argone 'load t)
219  (defun cl-load(args)
220    (condsval ((cons 'suc (suc-al-load sval nil)))))
221
222  (set-argone 'include t)
223  (defun cl-include(args)
224    (condsval ((cons 'suc (suc-al-load sval t)))))
225
226  (defun suc-al-load(sval output)
227    (let ((p1 (construct-proc))
228          (v  (al-load sval output)))
229      (setq proc p1) (expand-proc)
230      v))
231
232  (set-argone 'provide t)
233  (defun cl-provide(args)
234    (condsval ((cons 'suc (if (or (symbolp sval) (stringp sval) (numberp sval))
235                              (let* ((fname (atom-app sval ".a"))
236                                     (key   (atom-app "$a$loaded-" fname)))
237                                (set-sym-value globale-tab key t))
238                            al-f)))))
239
240  (defvar require-path `("") "Path to search for -required- files")
241
242  (defun exist-pathf(paths fname)
243    (cond ((null paths) fname)
244          ((let* ((pfname (atom-app (car paths) fname))
245                  (f (my-open-read-file pfname)))
246             (if f (prog1 pfname (close f))
247               (exist-pathf (cdr paths) fname))))))
248
249
250  (set-argone 'add-require-path)
251  (defun cl-add-require-path(args)
252    (let ((sval nil))
253      (cond ((alval-p (setq args (strip-alval args)))      (cons 'sus args))
254            ((null-p  args)                          (cons 'suc require-path))
255            ((atom-p  args)                                (cons 'suc al-f))
256            ((alval-p (setq sval (strip-alval (car args)))) (cons 'sus sval))
257            ((or (symbolp sval) (numberp sval) (stringp sval))
258             (setq require-path (append require-path (list sval)))
259                                                   (cons 'suc require-path))
260            (t                                             (cons 'suc al-f)))))
261
262  (set-argone 'require t)
263  (defun cl-require(args)
264    (condsval ((cons 'suc (if (or (symbolp sval) (stringp sval) (numberp sval))
265                              (let* ((fname (atom-app sval ".a"))
266                                     (key   (atom-app "$a$loaded-" fname)))
267                                (if (has-sym-value-p globale-tab key) t
268                                  (progn (suc-al-load (exist-pathf require-path fname)
 nil)
```

```
269                                     (if (has-sym-value-p globale-tab key) t
 al-f))))
270                           al-f)))))
271
272  ;-------------------------------------------------------------------------------
273  ; Other System functions.
274
275  (set-argone 'trace t)
276  (defun cl-trace(args)                              (declare (ignore args))
277    (cond (al-trace (setq al-trace nil) (cons 'suc al-f))
278          (t        (setq al-trace t)   (cons 'suc t))))
279
280  (set-argone 'time-prompt)
281  (defun cl-time-prompt(args)                        (declare (ignore args))
282    (cond (al-time-prompt (setq al-time-prompt nil) (cons 'suc al-f))
283          (t        (setq al-time-prompt t)         (cons 'suc t))))
284
285  (set-argone 'stats-prompt)
286  (defun cl-stats-prompt(args)                       (declare (ignore args))
287    (cond (al-stats-prompt (setq al-stats-prompt nil) (cons 'suc al-f))
288          (t        (setq al-stats-prompt t)         (cons 'suc t))))
289
290  (set-argone 'warning-exec)
291  (defun cl-warning-exec(args)                       (declare (ignore args))
292    (cond (warning-exec (setq warning-exec nil) (cons 'suc al-f))
293          (t            (setq warning-exec t)   (cons 'suc t))))
294
295  (set-argone 'warning-load)
296  (defun cl-warning-load(args)                       (declare (ignore args))
297    (cond (warning-load (setq warning-load nil) (cons 'suc al-f))
298          (t            (setq warning-load t)   (cons 'suc t))))
299
300  (set-argone 'exit t)
301  (defun cl-exit(args)                              (declare (ignore args))
302    (setq exit-alloy-lisp t)
303    (cons 'suc (setq exit-alloy-loop t)))
304
305  (set-argone 'bye t)
306  (defun cl-bye(args)                              (declare (ignore args))
307    (cl-exit 1))
308
309  (set-argone 'quit t)
310  (defun cl-quit(args)                              (declare (ignore args))
311    (cl-exit 1))
312
313  (set-argone 'quit-al)
314  (defun cl-quit-al(args)                              (declare (ignore args))
315    (cons 'suc (setq exit-alloy-loop t)))
316
317  (set-argone 'scheduler-switch-steps)
318  (defun cl-scheduler-switch-steps(args)
319    (let ((sval nil))
320      (cond ((alval-p (setq args (strip-alval args)))      (cons 'sus args))
321            ((null-p  args)                               (cons 'suc gtq-max))
322            ((atom-p  args)                               (cons 'suc al-f))
```

```
323           ((alval-p (setq sval (strip-alval (car args)))) (cons 'sus sval))
324           ((and (integerp sval) (> sval 0))
325                           (cons 'suc (setq gtq-max sval)))
326           ((cons 'suc al-f)))))
327
328  (set-argone 'block1 t)
329  (defun cl-block1(args)
330    (condsvar ((cons 'suc sval))))
331
332  (set-argone 'blockn t)
333  (defun cl-blockn(args)
334    (cl-last-arg args))
335
336  (set-argone 'block t)
337  (defun cl-block(args) (declare (ignore args))
338    (cons 'suc t))
```

A.5.2 build2.l

Other basic built in functions:

```
;-------------------------------------------------------------------------------
; Other built in predicates.

;-------------------------------------------------------------------------------
; Functions with one argument containing all actual arguments. Do not suspend.

(set-argone 'list t)
(defun cl-list(args)
   (cons 'suc args))

(set-argone 'nl)
(defun cl-nl(args &optional stream)                    (declare (ignore args))
  (terpri stream) (force-output stream)
  (cons 'suc t))

;-------------------------------------------------------------------------------
; Functions with one argument containing all actual arguments. May suspend.

(set-argone 'var-p)
(defun cl-var-p(args)
  (cl-var args))

(set-argone 'var)
(defun cl-var(args)
  (let ((sval nil))
    (cond ((alval-p (setq args (strip-alval args)))       (cons 'sus args))
          ((atom-p  args)                                 (cons 'suc al-f))
          ((alval-p (setq sval (strip-alval (car args)))) (cons 'suc t))
          ((cons 'suc al-f)))))

(set-argone '$a$funcall t)
(defun cl-$a$funcall(args)
  (cond ((alval-p (setq args (strip-alval args)))       (cons 'sus args))
        ((atom-p  args)                                 (cons 'suc nil))
        ((let ((p (make-pprocess (make-slist-t (cons '$a$funcall-vs args) p-serial))))
           (cons 'suc (proc-result p))))))

(set-argone 'read)
(defun cl-read(args &optional stream)                   (declare (ignore args))
  (cons 'suc (let ((e (al-read stream)))
               (if (eq e al-eof) al-f e))))

(set-argone 'print)
(defun cl-print(args &optional stream)
  (cond ((alval-p (setq args (strip-alval args)))      (cons 'sus args))
        ((atom-p  args)                                (cons 'suc al-f))
        ((let ((val (strip-alval-e (car args))))
           (al-print val stream) (force-output stream)
           (cons 'suc val)))))

(set-argone 'printnl)
(defun cl-printnl(args &optional stream)
```

```
53    (cond ((alval-p (setq args (strip-alval args)))      (cons 'sus args))
54          ((atom-p  args)      (terpri stream)           (cons 'suc al-f))
55          ((let ((val (strip-alval-e (car args))))
56             (al-print val stream) (terpri stream) (force-output stream)
57             (cons 'suc val)))))
58
59  (set-argone 'random t)
60  (defun cl-random(args)
61    (condsval ((number-p sval) (cons 'suc (random sval)))
62              (t               (cons 'suc al-f))))
63
64  (set-argone 'car t)
65  (defun cl-car(args)
66    (condsval ((consp   sval) (cons 'suc (car sval)))
67              (t              (cons 'suc al-f))))
68
69  (set-argone 'cdr t)
70  (defun cl-cdr(args)
71    (condsval ((consp   sval) (cons 'suc (cdr sval)))
72              (t              (cons 'suc al-f))))
73
74  (set-argone 'scar t)
75  (defun cl-scar(args)
76    (condsval ((slist-p   sval) (cons 'suc (car (slist-expr sval))))
77              (t                (cons 'suc al-f))))
78
79  (set-argone 'scdr t)
80  (defun cl-scdr(args)
81    (condsval ((slist-p sval) (cons 'suc (let ((c (cdr (slist-expr sval))))
82                                           (if (cons-p c) (make-slist :expr c) c))))
83              (t              (cons 'suc al-f))))
84
85  (set-argone 'data)
86  (defun cl-data(args)
87    (condsval (t               (cons 'suc sval))))
88
89  (set-argone 'null-p t)
90  (defun cl-null-p(args)
91    (condsval ((null sval)    (cons 'suc sval))
92              (t              (cons 'suc al-f))))
93
94  (set-argone 'number-p)
95  (defun cl-number-p(args)
96    (condsval ((number-p sval) (cons 'suc sval))
97              (t               (cons 'suc al-f))))
98
99  (set-argone 'string-p)
100 (defun cl-string-p(args)
101   (condsval ((stringp  sval) (cons 'suc sval))
102             (t               (cons 'suc al-f))))
103
104 (set-argone 'closure-p t)
105 (defun cl-closure-p(args)
106   (condsval ((ext-function-p sval)   (cons 'suc sval))
107             (t                       (cons 'suc al-f))))
```

```
108
109  (set-argone 'atom-p t)
110  (defun cl-atom-p(args)
111    (condsval ((consp sval)   (cons 'suc al-f))
112              ((slist-p sval) (cons 'suc al-f))
113              (t              (cons 'suc sval))))
114
115  (set-argone 'cons-p t)
116  (defun cl-cons-p(args)
117    (condsval ((consp sval)   (cons 'suc sval))
118              (t              (cons 'suc al-f))))
119
120  (set-argone 'scons-p t)
121  (defun cl-scons-p(args)
122    (condsval ((slist-p sval)   (cons 'suc sval))
123              (t                (cons 'suc al-f))))
124
125  (set-argone 'not t)
126  (defun cl-not(args)
127    (condsval ((eq sval al-f) (cons 'suc t))
128              (t              (cons 'suc al-f))))
129
130  ;-------------------------------------------------------------------------------
131  ; Other functions. May need an argument to have a value.
132
133  (defmacro wait-out(e)
134    `(catch 'out (cons 'suc ,e)))
135
136  (defmacro get-out(e)
137    `(throw 'out (cons 'sus ,e)))
138
139  (setq al-cons (set-argone 'cons t))
140  (defun cl-cons(args)
141    (cond ((alval-p (setq args (strip-alval-l args))) (cons 'sus args))
142          ((atom-p   args)                            (cons 'suc nil))
143          ((alval-p  (cdr args))                      (cons 'sus (cdr args)))
144          ((atom-p   (cdr args))                      (cons 'suc (car args)))
145          ((cons 'suc (cons (car args) (cadr args))))))
146
147  (setq al-scons (set-argone 'scons t))
148  (defun cl-scons(args)
149    (cond ((alval-p (setq args (strip-alval-l args))) (cons 'sus args))
150          ((atom-p   args)                            (cons 'suc nil))
151          ((alval-p  (cdr args))                      (cons 'sus (cdr args)))
152          ((atom-p   (cdr args))                      (cons 'suc (car args)))
153          ((cons 'suc (make-slist :expr (cons (car args) (cadr args)))))))
154
155  ;(set-argone 'last-arg t)
156  (defun cl-last-arg(args)
157    (cond ((alval-p (setq args (strip-alval-l args))) (cons 'sus args))
158          ((atom-p  args)                             (cons 'suc al-f))
159          ((wait-out (cl-last-argl args)))))
160
161  (defun cl-last-argl(args)
162    (let ((rest (cdr args)))
```

```
163     (cond ((alval-p rest) (get-out rest))
164           ((atom-p  rest) (car args))
165           ((cl-last-argl (cdr args))))))
166
167 ;-----------------------------------------------------------------------------
168 ; Arithmetic functions.
169
170 (defvar mmng-var nil "Variable containing value of last call to my-member-ng.")
171
172 (defun my-member-ng(l)
173   (setq mmng-var (member-ng l)))
174
175 (defun cl-arith(op x rest)
176   (cond ((alval-p rest)                              (get-out rest))
177         ((atom-p  rest)                              x)
178         ((alval-p (car rest))                        (get-out (car rest)))
179         ((not (numberp (car rest)))                  al-f)
180         ((cl-arith op (funcall op x (car rest)) (cdr rest)))))
181
182 (set-argone 'sum)
183 (defun cl-sum(args)
184   (cond ((alval-p (setq args (strip-alval-l args))) (cons 'sus args))
185         ((wait-out (cl-arith '+ 0 args)))))
186
187 (set-argone 'times)
188 (defun cl-times(args)
189   (cond ((alval-p (setq args (strip-alval-l args))) (cons 'sus args))
190         ((wait-out (cl-arith '* 1 args)))))
191
192 (set-argone 'diff)
193 (defun cl-diff(args)
194   (cond ((alval-p (setq args (strip-alval-l args))) (cons 'sus args))
195         ((atom-p   args)                            (cons 'suc 0))
196         ((alval-p  (car args))                      (cons 'sus (car args)))
197         ((alval-p  (cdr args))                      (cons 'sus (cdr args)))
198         ((not (number-p (car args)))                (cons 'suc al-f))
199         ((atom-p   (cdr args))         (cons 'suc (- 0 (car args))))
200         ((wait-out (cl-arith '- (car args) (cdr args))))))
201
202 (set-argone 'div)
203 (defun cl-div(args)
204   (cond ((alval-p (setq args (strip-alval-l args))) (cons 'sus args))
205         ((atom-p   args)                            (cons 'suc 1))
206         ((alval-p  (car args))                      (cons 'sus (car args)))
207         ((alval-p  (cdr args))                      (cons 'sus (cdr args)))
208         ((not (number-p (car args)))                (cons 'suc al-f))
209         ((atom-p   (cdr args))         (cons 'suc (/ 1 (car args))))
210         ((wait-out (cl-arith '/ (car args) (cdr args))))))
211
212 (defun cl-quotx2(x y) (floor (/ x y)))
213
214 (set-argone 'quotient)
215 (defun cl-quotient(args)
216   (cond ((alval-p (setq args (strip-alval-l args))) (cons 'sus args))
217         ((atom-p   args)                            (cons 'suc 1))
```

```
218         ((alval-p  (car args))                    (cons 'sus (car args)))
219         ((alval-p  (cdr args))                    (cons 'sus (cdr args)))
220         ((not (number-p (car args)))              (cons 'suc al-f))
221         ((atom-p   (cdr args))        (cons 'suc (cl-quotx2 1 (car args))))
222         ((wait-out (cl-arith 'cl-quotx2 (car args) (cdr args))))))
223
224 (set-argone 'remainder)
225 (defun cl-remainder(args)
226   (cond ((alval-p (setq args (strip-alval-l args))) (cons 'sus args))
227         ((atom-p   args)                          (cons 'suc 1))
228         ((alval-p  (car args))                    (cons 'sus (car args)))
229         ((alval-p  (cdr args))                    (cons 'sus (cdr args)))
230         ((not (number-p (car args)))              (cons 'suc al-f))
231         ((atom-p   (cdr args))        (cons 'suc (remainder 1 (car args))))
232         ((wait-out (cl-arith 'remainder (car args) (cdr args))))))
233
234 ;-----------------------------------------------------------------------------
235 ; Comparison functions.
236
237 (defun cl-comp(op x rest)
238   (cond ((alval-p rest)                            (get-out rest))
239         ((atom-p  rest)                            t)
240         ((alval-p (car rest))                      (get-out (car rest)))
241         ((when (funcall op x (car rest)) (cl-comp op (car rest) (cdr rest))))))
242
243 (set-argone 'eql t)
244 (defun cl-eql(args)
245   (cond ((alval-p (setq args (strip-alval-l args))) (cons 'sus args))
246         ((atom-p  args)                            (cons 'suc t))
247         ((alval-p (car args))                      (cons 'sus (car args)))
248         ((wait-out (if (cl-comp 'eql (car args) (cdr args))
249                        (car args) al-f)))))
250
251 ;-----------------------------------------------------------------------------
252 ; Arithmetic Comparison functions.
253
254 (defun cl-arcomp(op x rest)
255   (cond ((alval-p rest)                            (get-out rest))
256         ((atom-p  rest)                            t)
257         ((alval-p (car rest))                      (get-out (car rest)))
258         ((not (numberp (car rest)))                nil)
259         ((when (funcall op x (car rest)) (cl-arcomp op (car rest) (cdr rest))))))
260
261 (set-argone 'lt)
262 (defun cl-lt(args)
263   (cond ((alval-p (setq args (strip-alval-l args))) (cons 'sus args))
264         ((atom-p  args)                            (cons 'suc t))
265         ((alval-p (car args))                      (cons 'sus (car args)))
266         ((not (number-p (car args)))               (cons 'suc al-f))
267         ((wait-out (if (cl-arcomp '< (car args) (cdr args))
268                        (car args) al-f)))))
269
270 (set-argone 'gt)
271 (defun cl-gt(args)
272   (cond ((alval-p (setq args (strip-alval-l args))) (cons 'sus args))
```

```
273         ((atom-p  args)                              (cons 'suc t))
274         ((alval-p (car args))                        (cons 'sus (car args)))
275         ((not (number-p (car args)))                 (cons 'suc al-f))
276         ((wait-out (if (cl-arcomp '> (car args) (cdr args))
277                        (car args) al-f)))))
278
279  (set-argone 'eq)
280  (defun cl-eq(args)
281    (cond ((alval-p (setq args (strip-alval-l args))) (cons 'sus args))
282          ((atom-p  args)                             (cons 'suc t))
283          ((alval-p (car args))                       (cons 'sus (car args)))
284          ((not (number-p (car args)))                (cons 'suc al-f))
285          ((wait-out (if (cl-arcomp '= (car args) (cdr args))
286                         (car args) al-f)))))
287
288  (set-argone 'le)
289  (defun cl-le(args)
290    (cond ((alval-p (setq args (strip-alval-l args))) (cons 'sus args))
291          ((atom-p  args)                             (cons 'suc t))
292          ((alval-p (car args))                       (cons 'sus (car args)))
293          ((not (number-p (car args)))                (cons 'suc al-f))
294          ((wait-out (if (cl-arcomp '<= (car args) (cdr args))
295                         (car args) al-f)))))
296
297  (set-argone 'ge)
298  (defun cl-ge(args)
299    (cond ((alval-p (setq args (strip-alval-l args))) (cons 'sus args))
300          ((atom-p  args)                             (cons 'suc t))
301          ((alval-p (car args))                       (cons 'sus (car args)))
302          ((not (number-p (car args)))                (cons 'suc al-f))
303          ((wait-out (if (cl-arcomp '>= (car args) (cdr args))
304                         (car args) al-f)))))
305
```

A.5.3 obuild.l

Basic built in Classes:

```
;-------------------------------------------------------------------------------
; Library of built-in objects.

;-------------------------------------------------------------------------------
; General Utilities.

(defun make-bclass(name interfaces)
  (let ((cl-names (cons (atom-app 'cl-c- name) (atom-app 'cl-o- name))))
    (make-class :origdef `((name ,@ name) (interfaces ,@interfaces))
                :tab cl-names :ndynnames '**bclass**)))

(defmacro bclass-clo(bc)
  `(cdr (class-tab ,bc)))

(defmacro bclass-clc(bc)
  `(car (class-tab ,bc)))

(defun set-bclass(al-name inter &optional export)
  (cl-provide (list al-name))
  (cl-provide (list (string-downcase (string al-name))))
  (set-in-table (make-bclass al-name inter))
  (set-bc-test (atom-app al-name '-p) al-name export))

(defun set-bc-test(al-name cname &optional export)
  (unless (or (has-sym-value-p tab al-name) (has-sym-value-p globale-tab al-name))
          (set-in-table (make-gbfunction `((e) (return (eql ('object-p e)
                                                            (quote ,cname))))))))

(defmacro bclass-p(c)
  `(eq (class-ndynnames ,c) '**bclass**))

(defmacro bobject-p(o)
  `(bclass-p (object-class ,o)))

(defun send-msg-b(msg obj args obj-self) (declare (ignore obj-self))
  (funcall (bclass-clo (object-class obj)) msg (bobject-value obj) args))

(defun send-m-class-b(msg cls args)
  (funcall (bclass-clc cls) msg cls args))

(defun make-bobject(class value)
  (let* ((is (get-feature 'interfaces (class-origdef class)))
         (int (get-feature 'all-methods is)))
    (make-object :class class :interface int :inienv value)))

(defmacro bobject-value(obj)
  `(object-inienv ,obj))

(defmacro condsvs1(&rest args)
  `(let ((sval nil))
     (cond ((alval-p (setq args (strip-alval args)))       (waitfor-val args))
           ((atom-p  args)                                 (give-value  al-f))
```

```
53            ((alval-p (setq sval (strip-alval (pop args)))) (waitfor-val sval))
54            ,@ args)))
55
56  (defmacro condsvs2(&rest args)
57    `(let ((sval nil)(sval2 nil))
58       (cond ((alval-p (setq args (strip-alval args)))      (waitfor-val args))
59             ((atom-p  args)                                (give-value  al-f))
60             ((alval-p (setq sval (strip-alval (pop args)))) (waitfor-val sval))
61             ((alval-p (setq args (strip-alval args)))      (waitfor-val args))
62             ((atom-p  args)                                (give-value  al-f))
63             ((alval-p (setq sval2 (strip-alval (pop args))))(waitfor-val sval2))
64             ,@ args)))
65
66
67  ;-------------------------------------------------------------------------------
68  ; Library objects. with build in special representation.
69
70  ;-------------------------------------------------------------------------------
71  ; Closure
72
73  (set-bclass 'closure '((all-methods val eval)) t)
74
75  (defconstant dumb-oclosure (make-bobject (get-sym-value globale-tab 'closure) 0))
76
77  (defun cl-c-closure(msg c args)    (declare (ignore args c))
78    (give-value (case msg
79                      (new       al-f))))
80  ;                    (new       (make-gfunction '(() (return)))))))
81
82  (defun cl-o-closure(msg value args)      (declare (ignore value))
83    (case msg
84          (val    (condsvs1
85                   (t (give-value sval))))
86          (eval   (condsvs1
87                   (t (give-value nil))))))
88
89  ;-------------------------------------------------------------------------------
90  ; Number
91
92  (set-bclass 'number '((all-methods val string)) t)
93
94  (defconstant dumb-onumber (make-bobject (get-sym-value globale-tab 'number) 0))
95
96  (defun cl-c-number(msg c args)    (declare (ignore args c))
97    (give-value (case msg
98                      (new       0))))
99
100 (defun cl-o-number(msg value args)      (declare (ignore value))
101   (case msg
102         (val    (condsvs1
103                  (t (give-value sval))))
104         (string (condsvs1
105                  (t (give-value (format nil "~A" sval)))))))
106
107 ;-------------------------------------------------------------------------------
```

```
108  ; String
109
110  (set-bclass 'string '((all-methods val atom number explode append)) t)
111
112  (defconstant dumb-ostring (make-bobject (get-sym-value globale-tab 'string) 0))
113
114  (defun cl-c-string(msg c args)    (declare (ignore args c))
115    (give-value (case msg
116                      (new       ""))))
117
118  (defun cl-o-string(msg value args)      (declare (ignore value))
119    (case msg
120          (val (condsvs1
121                (t (give-value sval))))
122          (atom (condsvs1 (t (give-value (intern sval)))))
123          (number (condsvs1 (t (give-value (let ((n (al-read
124                                                     (make-string-input-stream
125                                                      sval) t)))
126                                             (if (number-p n) n al-f))))))
127          (explode (condsvs1 (t (give-value (explode sval)))))
128          (append (let ((x (someend 'nstringp (setq args (strip-alval-e args)))))
129                    (cond ((null x)    (give-value (implode args)))
130                          ((alval-p x) (waitfor-val x))
131                          ((and (cons-p x) (alval-p (car x))) (waitfor-val (car x)))
132                          ((give-value al-f)))))))
133
134  (defun nstringp(x) (not (stringp x)))
135
136  ;-----------------------------------------------------------------------------
137  ; Cons
138
139  (set-bclass 'cons '((all-methods val car cdr)) t)
140
141  (defconstant al-cons-cl (get-sym-value globale-tab 'cons))
142  (defconstant dumb-ocons (make-bobject al-cons-cl 0))
143
144  (defun round-cons-p (e)   (eql al-cons-cl e))
145
146  (defun cl-c-cons(msg c args)    (declare (ignore c))
147    (case msg
148          (new (condsvs2
149                (t (give-value (cons sval sval2)))))))
150
151  (defun cl-o-cons(msg value args)          (declare (ignore value))
152    (case msg
153          (car (condsvs1
154                (t (give-value (car sval)))))
155          (cdr (condsvs1
156                (t (give-value (cdr sval)))))
157          (val (condsvs1
158                (t (give-value sval))))))
159
160  ;-----------------------------------------------------------------------------
161  ; Scons
162
```

```
163  (set-bclass 'scons '((all-methods val car cdr)) t)
164
165  (defconstant al-scons-cl (get-sym-value globale-tab 'scons))
166  (defconstant dumb-oscons (make-bobject al-scons-cl '() ))
167
168  (defun square-cons-p (e)   (eql al-scons-cl e))
169
170  (defun cl-c-scons(msg c args)    (declare (ignore c))
171    (case msg
172          (new (condsvs2
173                (t (give-value (make-slist :expr (cons sval sval2))))))))
174
175  (defun cl-o-scons(msg value args)         (declare (ignore value))
176    (case msg
177          (car (condsvs1
178                (t (give-value (car (slist-expr sval))))))
179          (cdr (condsvs1
180                (t (give-value (let ((c (cdr (slist-expr sval))))
181                                 (if (cons-p c) (make-slist :expr c) c))))))
182          (val (condsvs1
183                (t (give-value sval))))))
184
185  ;-----------------------------------------------------------------------------
186  ; Symbol
187
188  (set-bclass 'symbol '((all-methods val string)) t)
189
190  (defconstant dumb-osymbol (make-bobject (get-sym-value globale-tab 'symbol) 0))
191
192  (defun cl-c-symbol(msg c args)    (declare (ignore args c))
193    (give-value (case msg
194                      (new       nil))))
195
196  (defun cl-o-symbol(msg value args)      (declare (ignore value))
197    (case msg
198          (val    (condsvs1
199                   (t (give-value sval))))
200          (string (condsvs1
201                   (t (give-value (string sval)))))))
202
203
204  ;-----------------------------------------------------------------------------
205  ; Special-Class
206
207  (set-bclass 'special-class '((all-methods val)) t)
208
209  (defconstant dumb-ospecial-class (make-bobject (get-sym-value globale-tab
 'special-class) 0))
210
211  (defun cl-c-special-class(msg c args)    (declare (ignore args c))
212    (give-value (case msg
213                      (new       al-f))))
214
215  (defun cl-o-special-class(msg value args)       (declare (ignore value))
216    (case msg
```

```
217         (val (condsvs1
218               (t (give-value sval))))))
219
220  ;-----------------------------------------------------------------------------
221  ; Main Body of built in objects driver:
222
223  (defun send-bi-obj(msg obj args orig-obj)
224    (cond ((number-p obj) (send-message msg dumb-onumber (cons obj args) orig-obj))
225          ((symbol-p obj) (send-message msg dumb-osymbol (cons obj args) orig-obj))
226          ((stringp  obj) (send-message msg dumb-ostring (cons obj args) orig-obj))
227          ((cons-p  obj)  (send-message msg dumb-ocons
228                                                (cons obj args) orig-obj))
229          ((slist-p obj)  (send-message msg dumb-oscons
230                                                (cons obj args) orig-obj))
231          ((ext-function-p obj)
232                          (send-message msg dumb-oclosure (cons obj args) orig-obj))
233          (t              (send-message msg dumb-ospecial-class
234                                                (cons obj args) orig-obj))))
235
236  ;-----------------------------------------------------------------------------
237  ; Library objects.
238
239  ;-----------------------------------------------------------------------------
240  ; Multiple assignement variable.
241
242  (set-bclass 'ma-var '((all-methods get put) (reader get) (writer put)))
243
244  (defun cl-c-ma-var(msg c args)
245    (case msg
246          (new  (if (alval-p (setq args (strip-alval args)))
247                    (waitfor-val args)
248                  (give-value (make-bobject
249                               c (make-gval :val (if (cons-p args) (car args)
 al-f))))))))
250
251  (defun cl-o-ma-var(msg value args)
252    (case msg
253          (get (give-value (gval-val value)))
254          (put (if (alval-p (setq args (strip-alval args)))
255                   (waitfor-val args)
256                 (give-value (setf (gval-val value) (if (cons-p args) (car args)
 al-f)))))))
257
258  ;-----------------------------------------------------------------------------
259  ; Single assignement variable.
260
261  (set-bclass 'sa-var '((all-methods get put) (reader get) (writer put)))
262
263  (defun cl-c-sa-var(msg c args)
264    (case msg
265          (new  (if (alval-p (setq args (strip-alval args)))
266                    (waitfor-val args)
267                  (give-value (make-bobject c (if (cons-p args)
268                                                  (cons t (car args))
269                                                (cons nil (make-alval-m)))))))))
```

```
270
271  (defun cl-o-sa-var(msg value args)
272    (case msg
273          (get (give-value (cdr value)))
274          (put (if (alval-p (setq args (strip-alval args)))
275                   (waitfor-val args)
276                 (give-value (if (car value)
277                                 al-f
278                               (prog1 (car args) (setf (car value) t)
279                                 (alset-alval (if (cons-p args) (car args) al-f)
280                                              (cdr value)))))))))
281
282
283  ;-------------------------------------------------------------------------------
284  ; Single Assignment Dictionary.
285
286  (set-bclass 'dictionary '((all-methods get put) (reader  get) (writer put)))
287
288  (defun cl-c-dictionary(msg c args)   (declare (ignore args))
289    (give-value (case msg
290                      (new       (make-bobject c (make-hash-table
291                                                  :size 10 :test 'equal))))))
292
293  (defmacro symnamp(v)
294    `(or (numberp ,v) (symbolp ,v)))
295
296  (defun cl-o-dictionary(msg value args)
297    (case msg
298          (get (condsvs1
299  ;               ((not (symnamp sval)) (give-value al-f))
300                ((give-value (let ((v (has-sym-value value sval)))
301                               (cond ((eq v '**1*noval*1**)
302                                      (set-sym-value value sval (make-alval-m)))
303                                     (v)))))))
304          (put (condsvs2
305   ;             ((not (symnamp sval)) (give-value al-f))
306                ((give-value (let ((v (has-sym-value value sval)))
307                               (cond ((eq v '**1*noval*1**)
308                                      (set-sym-value value sval sval2))
309                                     ((alval-p (setq v (strip-alval v)))
310                                      (alset-alval sval2 v) sval2)
311                                     (al-f)))))))))
312
313  ;-------------------------------------------------------------------------------
314  ; Single Assignment Vector.
315
316  (set-bclass 'vector '((all-methods get put) (reader  get) (writer put)))
317
318  (defun cl-c-vector(msg c args)
319    (case msg
320          (new (condsvs1
321                ((not (numberp sval)) (give-value al-f))
322                ((give-value (let ((v (make-array (list sval)
323                                                  :initial-element '**1*noval*1**)))
324                               (make-bobject c v)))))))))
```

```
325
326  (defun cl-o-vector(msg value args)
327    (case msg
328          (get (condsvs1
329                ((not (symnamp sval))                (give-value al-f))
330                ((not (array-in-bounds-p value sval)) (give-value al-f))
331                ((give-value (let ((v (aref value sval)))
332                               (cond ((eq v '**1*noval*1**)
333                                      (setf (aref value sval) (make-alval-m)))
334                                     (v)))))))
335          (put (condsvs2
336                ((not (symnamp sval)) (give-value al-f))
337                ((not (array-in-bounds-p value sval)) (give-value al-f))
338                ((give-value (let ((v (aref value sval)))
339                               (cond ((eq v '**1*noval*1**)
340                                      (setf (aref value sval) sval2))
341                                     ((alval-p (setq v (strip-alval v)))
342                                      (alset-alval sval2 v) sval2)
343                                     (al-f)))))))))
344
345  ;-------------------------------------------------------------------------------
346  ; Multiple Assignment Vector.
347
348  (set-bclass 'ma-vector '((all-methods get put) (reader  get) (writer put)))
349
350  (defun cl-c-ma-vector(msg c args)
351    (give-value
352     (case msg
353           (new (condsvs1
354                 ((not (numberp sval)) (give-value al-f))
355                 ((let ((v (make-array (list sval)
356                                       :initial-element 'nil)))
357                    (make-bobject c v))))))))
358
359  (defun cl-o-ma-vector(msg value args)
360    (case msg
361          (get (condsvs1
362                ((not (symnamp sval))                (give-value al-f))
363                ((not (array-in-bounds-p value sval)) (give-value al-f))
364                ((give-value (aref value sval)))))
365          (put (condsvs2
366                ((not (symnamp sval)) (give-value al-f))
367                ((not (array-in-bounds-p value sval)) (give-value al-f))
368                ((give-value (setf (aref value sval) sval2)))))))
369
370  ;-------------------------------------------------------------------------------
371  ; Multiple Assignment Dictionary.
372
373  (set-bclass 'ma-dictionary '((all-methods get put del) (reader  get) (writer put
 del)))
374
375  (defun cl-c-ma-dictionary(msg c args)    (declare (ignore args))
376    (give-value (case msg
377                      (new       (make-bobject c (make-hash-table
378                                                  :size 10 :test 'equal))))))
```

```
379
380  (defmacro symnamp(v)
381    `(or (numberp ,v) (symbolp ,v)))
382
383  (defun cl-o-ma-dictionary(msg value args)
384    (case msg
385          (get (condsvs1
386  ;              ((not (symnamp sval)) (give-value al-f))
387                (((give-value (let ((v (has-sym-value value sval)))
388                                (cond ((eq v '**1*noval*1**) al-f)
389                                      (v)))))))
390          (del (condsvs1
391  ;              ((not (symnamp sval)) (give-value al-f))
392                (((give-value (let ((v (has-sym-value value sval)))
393                                (cond ((eq v '**1*noval*1**) al-f)
394                                      (t (remhash sval value) v)))))))
395          (put (condsvs2
396                ((not (symnamp sval)) (give-value al-f))
397                ((give-value (set-sym-value value sval sval2)))))))
398
399  ;-------------------------------------------------------------------------------
400  ; Top Level Caller. Used to make calls charged at the top level.
401  ; Useful to define monitors without suspending the creator.
402
403  (set-bclass 'top-level-caller
404   '((all-methods top-level-funcall) () ()))
405
406  (defun cl-c-top-level-caller(msg c args)    (declare (ignore args))
407    (give-value (case msg
408                      (new       (make-bobject c nil)))))
409
410  (defun cl-o-top-level-caller(msg value args) (declare (ignore value))
411    (case msg
412          (top-level-funcall
413           (let ((proc (top-level-proc (make-slist-t (list '$a$pass1 args) p-serial))))
414              (push proc proc-stack)
415              (give-value (proc-result proc))))))
416
417
418  ;-------------------------------------------------------------------------------
419  ; Input.
420
421  (set-bclass 'input
422              '((all-methods read read-line read-string1
423                             eof empty active close) () ()))
424
425  (defconstant in-con (make-bobject (get-sym-value global-tab 'input) (cons nil nil)))
426  (set-sym-value global-tab 'input-console in-con)
427
428  (defun cl-c-input(msg c args)
429    (case msg
430          (new
431           (if (null args) (give-value (make-bobject c (cons nil nil)))
432               (condsvs1
433                ((not (stringp sval))   (give-value al-f))
```

```
434               ((equal sval "console") (give-value (make-bobject c (cons nil nil))))
435               ((give-value (let ((stream (my-open-read-file sval)))
436                              (if (eq al-f stream) al-f
437                                (make-bobject c (cons nil stream))))))))))
438
439  (defun cl-o-input(msg value args)              (declare (ignore args))
440    (cond ((car value) (give-value al-f))
441          ((let ((stream (cdr value)))
442             (case msg
443                  (read-line (give-value (read-line stream nil al-f)))
444                  (read-string1 (give-value (let ((c (read-char stream nil al-f)))
445                                              (if (eq c al-f) al-f
446                                                (string c)))))
447                  (read      (give-value (let ((e (al-read stream)))
448                                           (if (eq e al-eof) al-f e))))
449                  (eof       (give-value (if (eq al-eof (peek-char nil stream nil
 al-eof))
450                                             t al-f)))
451                  (empty     (give-value (if (peek-char nil stream nil nil) al-f t)))
452                  (close     (setf (car value) t) (when stream (close stream))
453                             (give-value t))
454                  (active    (give-value t)))))))
455
456  ;-------------------------------------------------------------------------------
457  ; Output.
458
459  (set-bclass 'output
460              '((all-methods print printnl nl active close) () ()))
461
462  (defconstant out-con (make-bobject (get-sym-value global-tab 'output) (cons nil nil)))
463  (set-sym-value global-tab 'output-console out-con)
464
465  (defconstant err-con (make-bobject (get-sym-value global-tab 'output)
466                                     (cons nil *error-output*)))
467  (set-sym-value global-tab 'error-console err-con)
468
469  (defun cl-c-output(msg c args)
470    (case msg
471          (new
472           (if (null args) (give-value (make-bobject c (cons nil nil)))
473               (condsvs1
474                ((not (stringp sval))   (give-value al-f))
475                ((equal sval "console") (give-value (make-bobject c (cons nil nil))))
476                ((equal sval "error") (give-value (make-bobject c (cons nil
 *error-output*))))
477                ((give-value (let ((stream (my-open-write-file sval)))
478                               (if (eq al-f stream) al-f
479                                 (make-bobject c (cons nil stream))))))))))
480
481  (defun cl-o-output(msg value args)
482    (cond ((car value) (give-value al-f))
483          ((let ((stream (cdr value)))
484             (case msg
485                  (print     (cl-print   args stream) (give-value args))
486                  (printnl   (cl-printnl args stream) (give-value args))
```

```
487                 (nl        (cl-nl nil stream)       (give-value t))
488                 (close     (setf (car value) t) (when stream (close stream))
489                            (give-value t))
490                 (active    (give-value t)))))))
```

A.6 Library in ALLOY

A.6.1 alloylib.a

Higher level built in functions written in ALLOY:

```
;------------------------------------------------------------------------
; Alloy Library written in Alloy.

; Change these to point to your own directories of ALLOY program utilties.

(add-require-path "/harpo.a/student/mitsolid/alloy/progs/")
(add-require-path "/spunky.a/student/mitsolid/alloy/progs/")


; Do not change the rest lines in this file:

(setfun (xxmake-bi l)
   (when (cons-p l) ($a$make-bi (car l)) (xxmake-bi (cdr l))))

(load "lists.a")
(load "bool.a")
(load "loop.a")
(load "misc.a")

(load "higherf.a")
(load "fp.a")
```

A.6.2 lists.a

Utilities dealing with lists:

```
;-------------------------------------------------------------------------
; List primitives

(setfun (append l1 l2)
       (if (cons-p l1) (lreturn (cons (car l1) (append (cdr l1) l2)))
                       (lreturn l2)))

(setfun length(l)
   (return (if (cons-p l) (sum 1 (length (cdr l))) 0)))

(setfun mapcar(f l)
   (return (if (cons-p l) (cons (f (car l)) (mapcar f (cdr l)))
                          ())))

(setfun nth(n l)
   (return (if (gt n 1) (nth (diff n 1) (cdr l)) (car l))))

(setfun cdrn(n l)
   (return (if (ge n 1) (cdrn (diff n 1) (cdr l)) l)))

(setfun carn(n l)
   (return (if (ge n 1) (carn (diff n 1) (car l)) l)))

(setfun first-n-list(n l)
   (return (if (ge n 1) (cons (car l) (first-n-list (diff n 1) (cdr l))) nil)))

(setfun sublist(x y l)
   (return (first-n-list (diff y x -1) (cdrn (diff x 1) l))))

(setfun reverse(l)
   (return (reverse2 l nil)))

(setfun reverse2(l r)
   (return (if (cons-p l) (reverse2 (cdr l) (cons (car l) r))
                          r)))

(xxmake-bi '(append length mapcar nth carn cdrn first-n-list sublist
                    reverse reverse2))

```

A.6.3 bool.a

Utilities dealing with boolean expressions:

```
 1  ; Boolean expressions
 2
 3  (setfun (and . l)
 4     (if (cons-p l)
 5        [block (block  (unless (car l)         (fail))
 6                       (unless (and . [cdr l]) (fail)))
 7               (return (car l))]
 8        (return)))
 9
10  (setfun (and-call . l)
11     (if (cons-p l)
12        [lets (xv) (block [unless (set xv ((car l)))   (fail)]
13                          [unless (and-call . [cdr l]) (fail)])
14                  (return xv)]
15        (return)))
16
17  (setfun (or . l)
18     (when (cons-p l) (lets (rv xv)
19                         (when (set xv (car l))        (return xv))
20                         (when (set rv (or . [cdr l])) (return rv)))))
21
22  (setfun (or-call . l)
23     (when (cons-p l) (lets (rv xv)
24                         [when (set xv ((car l)))           (return xv)]
25                         [when (set rv (or-call . [cdr l])) (return rv)])))
26
27  (setfun equal(x y)
28     (if (eql x y)
29        (return x)
30        (if (cons-p x)
31           (if (and (cons-p y) (equal (car x) (car y)) (equal (cdr x) (cdr y)))
32              (return x) (return %f))
33           (if (scons-p x)
34              (if (and (scons-p y) (equal (scar x) (scar y)) (equal (scdr x) (scdr y)))
35                 (return x) (return %f))))))
36
37  (setfun ground(x)
38     (if (cons-p x) (when (and (ground (car x)) (ground (cdr x))) (return x))
39                    (if (scons-p x) (when (and (ground (scar x)) (ground (scdr x)))
(return x))
40                                    [return (data x)])))
41
42  (xxmake-bi '(ground and or and-call or-call equal))
```

A.6.4 loop.a

Utilities dealing with loops:

```
1  ;-------------------------------------------------------------------------
2  ; Loops
3
4  (setfun repeat(n code)
5     (when (gt n 0) (code) (repeat (diff n 1) code)))
6
7  (setfun srepeat(n code)
8     [when (gt n 0) (code) (srepeat (diff n 1) code)])
9
10 (setfun while(cond code)
11    [when (cond) (code) (while cond code)])
12
13 (setfun until(cond code)
14    [unless (cond) (code) (until cond code)])
15
16 (xxmake-bi '(repeat srepeat while until))
17
```

A.6.5 misc.a

Miscellaneous utilities:

```
1  ;-------------------------------------------------------------------------
2  ; Miscelaneous.
3
4  (setfun gvalue()
5     (return (let ((v ('new sa-var)))
6                 (list ('writer v) ('get v)))))
7
8  (xxmake-bi '(gvalue))
```

A.6.6 hifherf.a

Higher order functions:

```
;------------
; Support functions for higher level functional programming.
;-------------------------------------------------------------------------

(provide "higherf")

;---------------------
; Drivers of functions

(setfun reduce(f init l)
   (return (if (cons-p l) (f (car l) (reduce f init (cdr l)))
                          init)))

(setfun filter(pred l)
   (return (if (cons-p l)
               (if (pred (car l)) (cons (car l) (filter pred (cdr l)))
                                  (filter pred (cdr l)))
               nil)))

(setfun mapcar2(f2 l1 l2)
   (return (if (cons-p l1)
               (cons (f2 (car l1) (car l2)) (mapcar2 f2 (cdr l1) (cdr l2)))
               ())))

(setfun mapcarn(fn . ll) (return (mapcarn-fix fn ll)))

(setfun mapcarn-fix(fn ll)
   (return (if (cons-p (car ll))
               (cons (let ((cars (mapcar car ll))) (fn . cars))
                     (mapcarn-fix fn (mapcar cdr ll)))
               ())))

;-------------------------
; These Produce functions

(setfun h-bu(f2 a1)
   (return (mu(a2) (lreturn (f2 a1 a2)))))

(setfun h-rev2(f2)
   (return (mu(x y) (lreturn (f2 y x)))))

(setfun h-rev(f)
   (return (mu l (lreturn (f . [reverse l])))))

(setfun h-red(f2 a)
   (return (mu(l) (lreturn (reduce f2 a l)))))

(setfun h-dup(f2)
   (return (mu(x) (lreturn (f2 x x)))))

(setfun h-comp(f g)
   (return (mu x (lreturn (f (g . x))))))
```

```
53
54  (setfun h-const(a)
55     (return (mu x (lreturn a))))
56
57  (setfun h-map(f1)
58     (return (mu(l) (lreturn (mapcar f1 l)))))
59
60  (setfun h-map2(f2)
61     (return (mu(l1 l2) (lreturn (mapcar2 f2 l1 l2)))))
62
63  (setfun h-mapn(fn)
64     (return (mu ll (lreturn (mapcarn fn . ll)))))
65
66  (setfun h-construct lfs
67     (return (mu(arg) (lreturn ((h-map (mu(f) (lreturn (f arg)))) lfs)))))
68
69  (setfun h-compn lfs
70     (return (mu (arg) (lreturn (ccallse lfs arg)))))
71
72  (setfun ccallse (lfs e)
73     (return (if (cons-p lfs) ((car lfs) (ccallse (cdr lfs) e)) e)))
```

A.6.7 fp.a

Utilities for programming in FP style:

```
1  ;------------
2  ; FP like support function for higher level functional programming.
3  ;-------------------------------------------------------------------------------
4
5  (provide "fp")
6  (require "higherf")
7
8  (setfun fp-map2(f2) (return (fp-fun (h-map2 f2))))
9
10 (setfun fp-rev2(f2)
11    (return (mu((x y)) (lreturn (f2 (list y x))))))
12
13 (setfun fp-rev(f)
14    (return (mu(l) (lreturn (f (reverse l))))))
15
16 (setfun fp-trans(ll)
17    (return (mapcarn list . ll)))
18
19 (setfun fp-distl((a l))
20    (return (mapcar (mu(x) (lreturn (list a x))) l)))
21
22 (setfun fp-distr((a l))
23    (return (mapcar (mu(x) (lreturn (list x a))) l)))
24
25 (setfun fp-fun(fn)
26    (return (mu(l) (lreturn (fn . l)))))
```

A.7 Creating ALLOY

This sections lists the files useful to create and maintain ALLOY in a system with UNIX.

A.7.1 Makefile

The following file is the file which is used by UNIX command `make` to build ALLOY:

```
 1  # Make the ALLOY interpreter.
 2  # Tested with AKCL Common Lisp interpreter.
 3  # If another version of Common Lisp is used correct file alakcl.l
 4  # If lisp system does not require a CR to give line for processing
 5  #    (e.g. Golden CL) check function interactive in file load.l
 6  # You must change the path in file alloylib.a to point to you library dir.
 7  #    This is the library of ALLOY sources (in progs.tar.Z)
 8  # In this Makefile, replace all occurances of ".o"
 9  #    with the suffix of your lisp's compiled files.
10
11
12  # Or if alloy is not available use these:
13  #LISP=cl
14  #PREXEC='(load "alloy.l")'
15
16  # If an old version of ALLOY is available use these:
17  LISP=alloy
18  PREXEC="(quit-al)"
19
20
21  ########################
22  # Main part of Makefile
23  ########################
24
25  .SUFFIXES: .l .o .a
26  SHELL=/bin/sh
27
28  OBJS=fast.o util.o alutil.o prep.o gprep.o oprep.o load.o values.o io.o\
29       inter.o fcall.o gcall.o rcall.o ocall.o build.o build2.o obuild.o
30
31  LSPS=fast.l util.l alutil.l prep.l gprep.l oprep.l load.l values.l io.l\
32       inter.l fcall.l gcall.l rcall.l ocall.l build.l build2.l obuild.l
33
34  ALIB=alloylib.a lists.a loop.a bool.a misc.a higherf.a fp.a
35
36  akclalloy:  sysal
37  @echo DONE
38
39  sysal:  ${OBJS} ${ALIB}
40  makesysal
41
42  .l.o:
43  execfirst ${LISP} ${PREXEC} "(compile-file " \"$*.l\" " )"
44
45  # Compile all files after puting them into one file. SLOW.
46
47  onesysal:  allal.o ${ALIB}
```

```
48  makesall
49  mv -f sysal onesysal
50
51  allal.o:  allal.l
52  execfirst ${LISP} ${PREXEC} "(compile-file " \"$*.l\" " )"
53
54  allal.l:  ${LSPS}
55  rm -f allal.l
56  cat ${LSPS} > allal.l
57
58  clean:
59  rm -fr *.o *~ allal.l RCS SCCS
60
```

A.7.2 execfirst

The following executable file is used by `make` and allows automated execution of lisp commands from UNIX:

```
1  #! /bin/csh -f
2  # Call first arg and give as input rest args.
3
4  set EXEC=$1
5  shift
6
7  ${EXEC} << ENDEND
8  $*
9  ENDEND
```

Appendix B

Benchmarks

This appendix lists a number of programs used for testing and benchmarking the ALLOY interpreter. These programs are loaded and executed automatically by calling script `bench`:

```
#! /bin/csh -f
# Run all benchmarks

(time ; a ; time) < bench.a > bench.out
```

which marks the system time before and after loading file `bench.a`:

```
(alloy)
(time-prompt)
(stats-prompt)
(load "bench1.a")
(bench1a)
(bench1b)
(bench1c)
(load "bench2.a")
(bench2a)
(bench2b)
(bench2c)
(load "bench3.a")
(bench3)
(load "bench4.a")
(bench4a)
(bench4b)
(load "bench5.a")
(bench5a)
(load "bench6.a")
(bench6a)
(bench6b)
(load "bench7.a")
(bench7a)
(load "bench8.a")
(bench8)
(load "bench9.a")
(bench9)
(load "bench10.a")
```

```
(bench10a)
(bench10b)
(load "bench11.a")
(bench11a)
(bench11b)
(bench11c)
(load "bench12.a")
(bench12)
(bench12b)
(load "bench13.a")
(bench13a)
(bench13b)
(bench13c)
(bench13d)

(exit)
```

This way ALLOY is instructed to print various statistics on the execution of each command. As it can be seen many more benchmarks are performed than those presented in a table of chapter 8. This is what each benchmark of that table is called here: `app3` is `bench1a`, `nrev30` is `bench1c`, `length100` is `bench3`, `whut80` is `bench9`, `qsort11` is `bench5a`, `ssort11` is `bench7a`, `lprimes5` is `bench4a`, `lprimes25` is `bench4b`, `inters33` is `bench12b`, `perms3` is `bench6a`, `perms4` is `bench6b`, `queens4` is `bench10a`, `queens4a` is `bench10b`, `prapp2` is `bench11a`, `prapp2or` is `bench11b`, `pinters33or` is `bench11c`, `dine3-4` is `bench8`.

The rest of this appendix lists the alloy programs forming the full suit of benchmarks.

B.1 Simple Functional

B.1.1 Append and naive reverse

```
;-----------------------------------------------------------------------
; Benchmark set no 1.
; Append and Naive reverse.

(setfun app(l1 l2)
   (return (if (cons-p l1)
               (cons (car l1) (app (cdr l1) l2))
               l2)))

(setfun nrev(l)
   (return (if (cons-p l)
               (app (nrev (cdr l)) (list (car l)))
               nil)))

(setfun bench1a()
  (return (app '(1 2 3) '(4 5 6)) ))

(setfun bench1b()
  (return (nrev '(1 2 3)) ))
```

```
(setfun bench1c()
  (return (nrev '(1 2 3 4 5 6 7 8 9 10
                            11 12 13 14 15 16 17 18 19 20
                            21 22 23 24 25 26 27 28 29 30)) ))
```

B.1.2 Make and find length of list

```
;-----------------------------------------------------------------------
; Benchmark set no 3.
; Make a list and count its length.

(setfun make-listl(n)
   (return (if (lt n 1) nil (cons n (make-listl (diff n 1))))))

(setfun get-length(l)
   (return (if (cons-p l) (sum 1 (get-length (cdr l))) 0)))

(setfun bench3()
   (return (get-length (make-listl 100))))
```

B.1.3 While and until

```
;-----------------------------------------------------------------------
; Benchmark set no 9.
; While and until statements.

(setfun bench9()
   [lets ((_x 0))
      (while ^(lt _x 20) ^[lets() [set _x (sum  _x 1)] [print _x]]) (nl)
      (until ^(lt _x 1)  ^[lets() [set _x (diff _x 1)] [print _x]]) (nl)
      (while ^(lt _x 20) ^[lets() [set _x (sum  _x 1)] [print _x]]) (nl)
      (until ^(lt _x 1)  ^[lets() [set _x (diff _x 1)] [print _x]]) (nl)])
```

B.2 Sorting

B.2.1 Quick sort using lists

```
;-----------------------------------------------------------------------
; Benchmark set no 5.
; Bench mark 5. Quicksort.

(load "qsort.a")

(setfun bench5a()
   (return (qsort '(5 2 6 3 7 4 5 2 4 7))))
```

and file `qsort.a`:

```
1  ;========== ALLOY version 0.1 ============= 10/10/89 ==========================
2  ; qsort.a
```

```
3  ; Fast Quick sort. Input and utput are lists.
4  ;-------------------------------------------------------------------------
5  ; This program demonstrates how fine grain parallelism is possible
6  ;  in many cases without any need for synchronization.
7  ; Assuming an adequate number of processors  is available,
8  ;  this program has a worst case execution time of O(n).
9  ;-------------------------------------------------------------------------
10
11 (setfun qsort(l)
12    (return (qsortrest l nil)))
13
14 (setfun qsortrest(l rest)
15    (if (null-p l) (return rest)
16      (return (let (((lsmall lgreat) (partition (car l) (cdr l))))
17                (qsortrest lsmall (cons (car l) (qsortrest lgreat rest)))))))
18
19 (setfun partition(x l)
20    (if (null-p l) (return '(nil nil))
21      (return (let (((lsmall lgreat) (partition x (cdr l))))
22                (if (lt x (car l)) (list lsmall (cons (car l) lgreat))
23                  (list (cons (car l) lsmall) lgreat))))))
24
25
```

B.2.2 Systolic sort

```
;-------------------------------------------------------------------------
; Benchmark set no 7.
; Systolic sort.

(load "ssort.a")

(setfun bench7a()
   (return (ssort '(5 2 6 3 7 4 5 2 4 7))))
```

and file `ssort.a`:

```
1  ============ ALLOY version 0.1 ============= 10/10/89 ==========================
2  ; ssort.a
3  ; Fast Systolic Sort. Input and utput are lists.
4  ; Executes in O(N) time for input of size N. Needs N processors.
5  ;-------------------------------------------------------------------------
6  ; Shows how one can use the type of streams most convenient.
7  ;-------------------------------------------------------------------------
8
9  (load "putlist.a")
10
11 (setfun ssort(l)
12    (return (if (null-p l)
13                nil
14                (let ((rest ('new putlist)))
15                   (cons (ssortx (car l) (cdr l) rest)
16                         (ssort ('get-orig-head rest)))))))
17
18 (setfun ssortx(x l rest)
```

```
19     [when (null-p l) ('end rest) (return x)]
20     (when (cons-p l)
21        [when (gt (car l) x) ('put rest (car l)) (return (ssortx x       (cdr l) rest))]
22        [when (le (car l) x) ('put rest x)       (return (ssortx (car l) (cdr l)
rest))]))
```

B.3 Lazy evaluation

B.3.1 Prime numbers

```
;-------------------------------------------------------------------------
; Benchmark set no 4.
; Prime numbers, lazily.

(load "sieved.a")
(load "printn.a")

(setfun bench4a()
   (return (printn 5 (lprimes))))

(setfun bench4b()
   (return (printn 25 (lprimes))))
```

file `sieved.a`:

```
1  ;========= ALLOY version 0.1 ============= 10/10/89 ==========================
2  ; sieve.a
3  ; Evaluates eagerly or lazily all prime numbers.
4  ;-------------------------------------------------------------------------------
5  ; This program demonstrates how a program that evaluates eagerly
6  ;  can be made to evaluate lazyly.
7  ;-------------------------------------------------------------------------------
8
9  (setfun lprimes()
10   (return (lazy (primes))))
11
12 (setfun primes()
13   (return (cons 1 (sieve (ints 2)))))
14
15 (setfun sieve(l)
16   (return (cons (car l) (sieve (filter (car l) (cdr l))))))
17
18 (setfun filter(x l)
19   (if (equal (remainder (car l) x) 0) (return (filter x (cdr l)))
20     (return (cons (car l) (filter x (cdr l))))))
21
22 (setfun ints(n)
23   (return (cons n (ints (sum n 1)))))
```

file `printn.a`:

```
1  ; Prints n elements from list
2
3  (setfun printn(i l)
4          [when (gt i 0) [print (data (car l))] (nl) (printn (diff i 1) (cdr l))])
```

B.4 Generators

B.4.1 Intersection

```
;-----------------------------------------------------------------------
; Benchmark set no 1.
; NUll replicator in list intersection.

(load "member.a")

(setfun bench12()
   (return (list #*(intersection '(2 4 6 1) '(3 5 2 6 7 1)))))

(setfun bench12b()
   (return (list #*(intersection '(5 1 2) '(2 4 5)))))
```

file `member.a`:

```
1  ;========== ALLOY version 0.1 ============= 10/10/89 ==========================
2  ; member.a
3  ; Membership and intersection.
4  ;-------------------------------------------------------------------------------
5  ; This program demonstrates the use of multiple generators in a way that
6  ;  can replace prolog's backtracking.
7  ;-------------------------------------------------------------------------------
8
9  (setfun getmember(l)
10   (when (cons-p l) (return (car l)) #(return *(getmember (cdr l)))))
11
12 (setfun intersection(s1 s2)
13   #(lets ((x *(getmember s1)) (y *(getmember s2)))
14       (when (eql x y) (return x))))
15
16 (setfun ismember(x l)
17   #(if (eql x *(getmember l)) (return)))
18
```

B.4.2 Permutations

```
;-----------------------------------------------------------------------
; Benchmark set no 1.
; All permutations.

(load "perms.a")

(setfun bench6a()
   (return (list #*(permutations '(1 2 3)))))

(setfun bench6b()
   (return (list #*(permutations '(1 2 3 4)))))
```

file `perms.a`:

```
1  ;========== ALLOY version 0.1 ============= 10/10/89 ==========================
2  ; perms.a  ??
3  ; Return one or all permutations of a list.
4  ; If called normally will return one of the possible permutations of the list.
```

```
 5  ; If called appropriatelly it will return all permutations of the list.
 6  ; ------------------------------------------------------------------------
 7  ; This program demonstrates how a non-deterministic function can be made
 8  ;  to return all possible solutions.
 9  ; ------------------------------------------------------------------------
10
11  (provide "perms")
12
13  (setfun permutations(l)
14    (if (cons-p l)
15        (list #(lets (((x rest) *(delete-one l)))
16                  #(return (cons x *(permutations rest)))))
17        (return nil)))
18
19  (setfun lpermutations(l)
20    (if (cons-p l)
21        (list #(lets (((x rest) *(delete-one l)))
22                  #(return (cons x *(lpermutations rest)))))
23        (return nil)))
24
25  (setfun delete-one(l)
26    (when (cons-p l)
27      (return (list (car l) (cdr l)))
28     #(return (let (((x rest) *(delete-one (cdr l))))
29                (list x (cons (car l) rest))))))
```

B.4.3 Queens

```
;------------------------------------------------------------------------
; Benchmark set no 10.
; 4 queens. One solution and all solutions.

(load "queens.a")

(setfun bench10a()
   (return (queens 4)))

(setfun bench10b()
   (return (list #*(queens 4))))
```

file `queens.a`:

```
 1  ;========== ALLOY version 0.1 ============= 10/10/89 ========================
 2  ; queens.a
 3  ; Solves the n queens problem.
 4  ; Evaluates and returns one or all possible places, as requested.
 5  ;------------------------------------------------------------------------
 6  ; This program demonstrates the ability to find multiple solutions
 7  ;  while at the same time maximum parallelism is used.
 8  ; Here, Checking of a configuration proceeds in parallel with the
 9  ;  creation of the permutation that describes the configuration.
10  ;------------------------------------------------------------------------
11
12  (require "perms")
13
14  (setfun queens(n)
```

```
15   (lets ((lst (make-list n)))
16     #(lets ((poslist (board lst *(permutations lst))))
17        (when (not (unsafeall poslist)) (printnl poslist) (return poslist)))))
18
19  (setfun unsafeall(board)
20    (when (cons-p board)
21      (when (unsafe (car board) (cdr board)) (return))
22      (when (unsafeall (cdr board))          (return))))
23
24  (setfun unsafe(q board)
25    (when (cons-p board)
26       (when (unsafe1 q (car board)) (return))
27       (when (unsafe  q (cdr board)) (return))))
28
29  (setfun abs(x)
30    (return (if (lt x 0) (diff x) x)))
31
32  (setfun unsafe1(q p)
33    (when (eq (abs (diff (car q) (car p)))
34              (abs (diff (cdr q) (cdr p)))) (return)))
35
36  (setfun board(lst places)
37    (return (if (null-p lst) ()
38              (cons (cons (car lst) (car places))
39                    (board (cdr lst) (cdr places))))))
40
41  (setfun make-list(n)
42    (if (ge n 1)
43       (return (cons n (make-list (diff n 1))))
44       (return ())))
```

B.5 Prolog

B.5.1 Append, Or Parallel Append, and Or Parallel Intersection

```
;------------------------------------------------------------------------
; Benchmark set no 11.
; Prolog append. Or parallel Prolog append. Or parallel Prolog Interesction.

(load "p-app.a")
(load "p-app-or.a")
(load "p-inters.a")

(setfun bench11a()
   (return (test-app)))

(setfun bench11b()
   (return (test-app-or)))

(setfun bench11c()
   (return (intersectionp '(5 1 2) '(2 4 5))))
```

file p-app.a:

```
1  ;========== ALLOY version 0.1 ============= 10/10/89 ========================
```

```
 2  ; prolapp.a
 3  ; prolog style append functions.
 4  ; When called as a generator returns all solutions.
 5  ; Usually called with logical variables as arguments.
 6  ;-------------------------------------------------------------------------
 7  ; Shows how prolog style functions can be written in ALLOY.
 8  ; To add a cut anywhere in the program, insert "*(unless *(cut) (fail)"
 9  ; Where function cut is defined as "(setfun cut() (return))"
10  ;-------------------------------------------------------------------------
11
12  ; Prolog Program.
13  ; append([],     L, L     ).
14  ; append([H| T], L, [H| R]):- append(T, L, R).
15
16  ; Simplified Prolog Program.
17  ; append(P1, P2, P3):- unify(P1,[]), unify(P2, L), unify(P3, L).
18  ; append(P1, P2, P3):- unify(P1, [H| T]), unify(P3, [H| R]), append(T, P2, R).
19
20  ; The result of a mechanical transformation. No optimizations.
21
22  (require "unify")
23
24  [setfun p-app(p1 p2 p3)
25     [lets ((l ('new logvar)))
26        #[s *(unify p1 ()) #[s *(unify p2 l) #[s *(unify p3 l) (return)]]]]
27     [lets ((l ('new logvar)) (t ('new logvar)) (r ('new logvar)) (h ('new logvar)))
28        #[s *(unify p1 (cons h t)) #[s *(unify p3 (cons h r)) #[s *(p-app t p2 r)
(return)]]]]]
29
30  (setfun (s x))
31
32  (setfun test-app()
33     (return [let (((_v1 _v2) [list ('new logvar) ('new logvar)]))
34                [list #[blockn *(p-app _v1 _v2 '(1 2)) [rstrip (list _v1 _v2)]]]]))
```

file p-app-or.a:

```
 1  ============ ALLOY version 0.1 ============ 10/10/89 ========================
 2  ; prolapp.a
 3  ; OR-parallel prolog style append functions.
 4  ; When called as a generator returns all solutions.
 5  ; Usually called with logical variables as arguments.
 6  ;-------------------------------------------------------------------------
 7  ; Shows how or-parallel prolog style functions can be written in ALLOY.
 8  ; To add a cut anywhere in the program, insert "*(unless *(cut) (fail)"
 9  ; Where function cut is defined as "(setfun cut() (return))"
10  ;-------------------------------------------------------------------------
11
12  ; Prolog Program.
13  ; append([],     L, L     ).
14  ; append([H| T], L, [H| R]):- append(T, L, R).
15
16  ; Simplified Prolog Program.
17  ; append(P1, P2, P3):- unify(P1,[]), unify(P2, L), unify(P3, L).
18  ; append(P1, P2, P3):- unify(P1, [H| T]), unify(P3, [H| R]), append(T, P2, R).
19
20  ; Mechanical transformation. No optimizations.
```

```
21  ; Cut would mean that all branches of the predicate failed.
22
23  (require "prolog")
24
25  (setfun p-app-or(lp1 lp2 lp3)
26     (let [((p1 p2 p3) [copyloge (list lp1 lp2 lp3)])
27           ((l) (lvs 1))]
28        (snest ^!*(unify p1 ()) ^!*(unify p2 l) ^!*(unify p3 l)
29               ^(return (list p1 p2 p3))))
30     (let [((p1 p2 p3) [copyloge (list lp1 lp2 lp3)])
31           ((l t r h) (lvs 4))]
32        (snest ^!*(unify p1 (cons h t)) ^!*(unify p3 (cons h r))
33               ^![lets [((T2 L2 R2) *(p-app-or t p2 r))]
34                       (return (list (cons h T2) l2 (cons h R2)))])))
35
36  (setfun test-app-or()
37     (return [let (((_v1 _v2) [list ('new logvar) ('new logvar)]))
38                [list #[printnl [rstrip *(p-app-or _v1 _v2 '(1 2))]]]]))
```

file p-inters.a:

```
1   ;========== ALLOY version 0.1 ============= 10/10/89 ==========================
2   ; proinsp.a
3   ; OR-parallel prolog style intersection with member function.
4   ; Usually called with logical variables as arguments.
5   ;-----------------------------------------------------------------------------
6   ; Shows how or-parallel prolog style functions can be written in ALLOY.
7   ; Demonstrates double use of the member predicate.
8   ; To add a cut anywhere in the program, insert "*(unless *(cut) (fail)"
9   ; Where function cut is defined as "(setfun cut() (return))"
10  ;-----------------------------------------------------------------------------
11
12  ; Prolog Program.
13  ; memberp(X, [X| _]).
14  ; memberp(X, [_| R]):- memberp(X, R).
15  ; common(X, L1, L2):- memberp(X, L1), memberp(X, L2).
16  ; intersection(L, L1, L2):- bagof(X, common(X, L1, L2), L).
17
18  ; Simplified Prolog Program.
19  ; memberp(X, P2):- unify(P2, [X| _]).
20  ; memberp(X, P2):- unify(P2, [_| R]), memberp(X, R).
21  ; commonp(X, L1, L2):- memberp(X, L1), memberp(X, L2).
22  ; intersectionp(L, L1, L2):- bagof(X, commonp(X, L1, L2), L).
23
24  ; Mechanical transformation. Minimal optimizations (nested lets).
25  ; Cut would mean that all branches of the predicate failed.
26
27  (require "prolog")
28
29  [setfun memberp(lp1 lp2)
30     [lets [((p1 p2) (copyloge (list lp1 lp2)))]
31        (snest ^!*(unify (cons p1 ('new logvar)) p2)
32               ^(return (list p1 p2)))]
33     (lets [((x p2) (copyloge (list lp1 lp2)))
34            ((r h) (lvs 2))]
35        (snest ^!*(unify p2 (cons h r))
36               ^![lets [((X2 R2) *(memberp x r))]
```

```
37                      (return (list X2 (cons h r2)))]))]
38
39  (setfun commonp(lp1 lp2 lp3)
40      [lets [((x l1 l2) (copyloge (list lp1 lp2 lp3)))]
41         #[lets [((X2 l1_2) *(memberp x l1))]
42                #[lets [((x3 l2_2) *(memberp x2 l2))]
43                       (return (list x3 l1_2 l2_2))]]])
44
45  (setfun intersectionp(lp2 lp3)
46     (return (let [((x) (lvs 1))]
47                [list #[rstrip *(commonp x lp2 lp3)]])))
```

B.6 Miscellaneous

B.6.1 Dining Philosophers

```
;-------------------------------------------------------------------------
; Benchmark set no 8.
; Philosophers

(load "aphil.a")

(setfun bench8()
   (dine 3 4))
```

file `phil.a`:

```
 1  ;-------------------------------------------------------------------
 2  ; N Philosophers.
 3
 4  (class-start philosopher
 5               (static id fork1 fork2)
 6               (import print nl gt diff printnl sum)
 7               (common faa semaphore)
 8               (methods live))
 9
10  (setfun new(f1 f2)
11     (set id (sum ('faa c-faa 1) 1)) (set fork1 f1) (set fork2 f2)
12     (if (gt id 1) ('v c-semaphore)))
13
14  (setfun live(n)
15     (loop n))
16
17  [setfun loop(n)
18     ('p c-semaphore) [printnl (list "Philosopher" id "entered room.")]
19     ('p fork1)       [printnl (list "Philosopher" id "                took left
fork")]
20     ('p fork2)       [printnl (list "Philosopher" id "                took right
fork")]
21     [printnl (list "Philosopher" id "started eating")]
22     (eat 2)
23     [printnl (list "Philosopher" id "stoped eating")]
24     [printnl (list "Philosopher" id "                put down left  fork")] ('v fork1)
25     [printnl (list "Philosopher" id "                put down right fork")] ('v fork2)
26     ('v c-semaphore) [printnl (list "Philosopher" id "exited  room.")]
```

```
27     (if (gt n 0) (loop (diff n 1)))]
28
29  (setfun eat(n)
30     (if (gt n 0) (eat (diff n 1))))
31
32  (class-end philosopher)
33
34  ------------------------------------
35  ; Dinner for p philosophers n times.
36
37  [setfun dine(p n)
38     [printnl "The big feasting starts..."]
39     (dinen2 (if p p 5) (if n n 3))]
40
41  (setfun dinen2(p n)
42     (let ((fork1 ('new semaphore 1)) create)
43        (set create (mu (previous i)
44                        (if (gt i 1)
45                           (let ((f ('new semaphore 1)))
46                              ('live ('new philosopher previous f) n)
47                              (create f (diff i 1)))
48                           ('live ('new philosopher previous fork1) n))))
49        (create fork1 p)))
```

Bibliography

[Agh86a] Gul Agha. An overview of actor languages. *SIGPLAN Notices*, 21(10):58–67, October 1986.

[Agh86b] Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. Series in Artificial Intelligence. The MIT Press, 1986.

[Agh89] Gul Agha. Foundational issues in concurrent computing. *ACM SIGPLAN Notices*, 24(4):60–65, April 1989.

[AH87] Gul Agha and Carl Hewitt. Concurrent programming using actors. In Akinori Yonezawa and Marion Tokoro, editors, *Object-Oriented Concurrent Programming*, Series in Computer Science, pages 36–53. The MIT Press, 1987.

[Bac78] John Backus. Can programming be liberated from the von newmann style? a functional style and its algebra of programs. *Communications of ACM*, 21(8):613–641, August 1978.

[BBB+57] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibit, H. L. Herrick, R. A. Hughes, and R. Nutt. The FORTRAN automatic coding system. In *Proceedings of Western Jt. Computer Conference*. AIEE (now IEEE), 1957. Los Angeles.

[BDG+88] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common lisp object system specification X3JI3 document 88-002R. *SIGPLAN Notices*, 23, September 1988. Special Issue.

[BG89] Reem Bahgat and Steve Gregory. Pandora: Non-deterministic parallel logic programming. In 6^{th} *International Conference on Logic Programming*, pages

471–486, Lisbon, Portugal, 1989.

[BKK+86] Daniel G. Bobrow, Kenneth Kahn, Gregory Kiczales, Larry Masinter, Mark Stefic, and Frank Zdybel. Commonloops merging lisp and object-oriented programming. *SIGPLAN OOPSLA Proceedings*, 21:17–29, September 1986.

[Can82] H. I. Cannon. Flavors: A non-hierarchical approach to object-oriented programming, 1982.

[CG81] K. L. Clark and S. Gregory. A relational language for parallel programming. In *Proceedings ACM Conference on Functional Languages and Computer Architecture*, pages 171–178, 1981. Also Chapter 1 in "Shapiro E. (ed.) *Concurrent Prolog*".

[CG83] K. L. Clark and S. Gregory. Parlog: A parallel logic programming language. Research Report DOC 83/5, Department of Computing, Imperial College of Science and Technology, University of London, London, UK, May 1983.

[CG84] K. L. Clark and Steve Gregory. Notes on the implementation of PARLOG. *Proceedings of International Conference on Fifth Generation Computer Systems*, pages 299–306, 1984. Tokyo.

[CG85] Keith Clark and Steve Gregory. Notes on the implementation of parlog. *The Journal of Logic Programming*, 2(1):17–42, April 1985.

[CG86] Keith Clark and Steve Gregory. Parlog: Parallel programming in logic. *ACM TOPLAS*, 8(1):1–49, January 1986. First version at 1983.

[CG88] N. Carriero and D. Gelernter. Applications exeperience with linda. *Proceedings of the ACM/SIGPLAN Symposioum on Parallel Programming*, pages 173–187, July 1988.

[CG89] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[Chu41] Alonzo Church. *The Calculi of Lambda-Conversion.* Princeton University Press, New Jersey, 1941.

[CM79] K. L. Clark and F. G. McCabe. The control facilities of IC-PROLOG. In Donald Michie, editor, *Expert Systems in the Micro Electronic Age*, chapter Knowledge-handling Formalisms, pages 123–149. Edinburgh University Press, 1979.

[Col72] A. Colmerauer. Un systeme de communication home-machine en francais. Rapport preliminaire, groupe de res. en intelligence artificielle, U. d' Aix-Marseille, 1972.

[Cox87] Brand J. Cox. *Object Oriented Programming, An Evolutionary Approach.* Addison-Wesley, 1987.

[CS87] M Codish and E. Shapiro. Compiling Or-Parallelism into And-Parallelism. *New Generation Computing*, 5(1):45–61, 1987. Also chapter 6 in "Shapiro E. (ed.), *Concurrent Prolog*".

[Dij75] E. W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *ACM Communications*, 19(8):453–457, August 1975.

[Dim88] Isaac Dimitrovsky. *Zlisp - A Portable Parallel Lisp Environment.* PhD thesis, New York University, Ultracomputer Group, 10th Floor, 715 Broadway, New York, NY 10003, May 1988.

[DN66] O. Dahl and K. Nygaard. Simula-an algol-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[DW83] Martin D. Davis and Elaine J. Weyuker. *Computability, Complexity, and Languages.* Series of Monographs and Textbooks. Academic Press, Inc., 1983.

[ELS88] J. Edler, J. Lipkis, and E. Schoenberg. Process management for highly parallel unix systems. In *Usenix workshop on UNIX and Supercomputers*, 10^{th} floor, 719 Broadway, New York, NY 10003, September 1988. USENIX.

[FG91] Eric Freudenthal and Allan Gottlieb. Process coordination with fetch-and-increment. *Architectural Support for Programming Languages and Operating Systems, ACM*, pages 260–268, 1991.

[GG83] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language.* Software Series. Prentice-Hall, 1983.

[GGK+83] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Lawrence Rudolph, and Marc Snir. The NYU ultracomputer–designing an MIMD shared memory parallel computer. *IEEE Transactions on Computing*, pages 175–189, February 1983.

[GLR83] Allan Gottlieb, B. D. Lubachevsky, and Larry Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM TOPLAS*, 5:164–189, April 1983.

[GM84] Richard P. Gabriel and John McCarthy. Queue-based multi-processing lisp. *SIGPLAN Lisp Conference*, pages 25–43, 1984.

[GN88] Michael R. Genesereth and Nils J. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann, 1988.

[Got86] Allan Gottlieb. An overview of the NYU ultra computer project. Ultracomputer Note 100, New York University, Ultra Computer Laboratory, July 1986.

[GR83] Adele Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[GR89] Adele Goldberg and David Robson. *SMALLTALK-80 The Language.* Series in Computer Science. Addison Wesley, 1989. Describes Version 2 of Smalltalk-80.

[Gre87] Steve Gregory. *Parallel Logic Programming in Parlog, The Language and its Implementation.* International Series in Logic Programming. Addison Wesley, Wokingham, UK, 1987.

[Hal84] Robert H. Halstead Jr. Implementation of multilisp: Lisp on a multiprocessor. *SIGPLAN Conference on Lisp programming*, pages 9–16, 1984.

[Hal86] Robert H. Halstead Jr. An assessment of multilisp: Lessons from experience. *International Journal of Parallel Programming*, 15(6):459–501, December 1986.

[Har86] Robert Harper. Introduction to standard ml. Computer Science Department, University of Edinburgh, Edinburgh EH9 3JZ, February 1986.

[HB88] Seif Haridi and Per Brand. Andorra prolog–an integration of prolog and commited choise languages. In *International Conference on Fifth Generation Computing.* Addison Wesley, 1988.

[Hen80] Peter Henderson. *Functional programming: Application and Implementation.* Series in Computer Series. Prentice/Hall International, 1980.

[Hew77] Carl Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8:323–364, 1977.

[HF82] Christofer T. Haynes and Daniel P. Friedman. Engines build process abstractions. *ACM Symposium on Lisp and Functional Programming*, pages 18–24, 1982.

[HFW84] Christofer T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. *ACM Symposium on Lisp and Functional Programming*, pages 293–299, 1984.

[HM87] Malcolm C. Harrison and Thanasis Mitsolides. Upl: Ultra prolog in lisp. New York University, Courant Institute, December 1987.

[HMP87] Malcolm C. Harrison, Nick Markantonatos, and George Papadopoulos. Ultra prolog. Ultra Computer Note 116, New York University, Ultra Computer Laboratory, January 1987.

[Hud89] Paul Hudak. Conception, evolution, and application of functional programming languages. In Salvatore T. March, editor, *ACM Computing Surveys: Special Issue on Programming Language Paradigms*, pages 359–411. ACM, ACM Press, September 1989. Vol 21, Number 3.

[Ing78] D. H. Ingals. The smalltalk-76 programming system design and implementation. *Proceedings of the 5th ACM Principles of Programming Languages Conference*, pages 9–16, 1978.

[KHM89] David A. Kranz, Robert H. Halstead Jr., and Eric Mohr. Mul-t: A high performance parallel lisp. *SIGPLAN Conference on Lisp programming*, pages 81–90, 1989.

[Kow74] R. A. Kowalski. Predicate logic as programming language. In *Proc. IFIP Congress*. North-Holland, Stockholm, 1974.

[Kow88] Robert A. Kowalski. The early years of logic programming. *Communications of the ACM*, 31(1):38–43, January 1988.

[KR78] Brian W. Kerninghan and Dennis M. Ritchie. *The C Programming Language*. Software Series. Prentice-Hall, 1978.

[KTMB86] Kenneth Khan, Eric Dean Tribble, Mark S. Miller, and Daniel G. Bobrow. Objects in concurrent logic programming languages. *SIGPLAN OOPSLA Proceedings*, 21:242–257, September 1986.

[KTMB86] K. Khan, E. D. Tribble, M. S. Miller, and D. G. Bobrow. Vulcan: Logical concurrent objects. In B. Shiver and P. Wegner, editors, *Research Directions in Object Oriented programming*. *, 86. Also in Shapiro, E. (ed.), *Concurrent Prolog*.

[LBO+88] E. Lusk, R. Butler, R. Olson, R. Overbeek, R. Stevens, D. H. D. Warren, A. Calderwood, P. Szeredi, S. Haridi, P. Brand, M. Carlsson, A. Ciepielewski, and B. Hausman. The aurora or-parallel prolog system. In *International Conference on Fifth Generation Computer Systems 1988*. ICOT, Addison Wesley, 1988.

[Lie86] Henry Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. *ACM OOPSLA Proceedings*, pages 214–223, September 1986.

[LS79] B. Liskov and A. Snyder. Exception handling in CLU. *IEEE Transactions on Software Engineering*, 1979.

[LS87] Jacob Levy and Ehud Shapiro. Cfl — a concurrent functional programming language embedded in a concurrent logic programming environment. In Ehud

Shapiro, editor, *Concurrent Prolog Vol 2*, chapter 35, pages 442–469. MIT Press, 1987.

[LSAS77] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[MAE+65] J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. Levin. *Lisp 1.5 Programmers Manual.* MIT Press, Cambridge, Massachusetts, 1965.

[Mar88] Nickolaos Markantonatos. The evolution of parallel logic programming languages. Technical Report 377, New York University, 251 Mercer Street, New York, NY 10012, June 1988.

[McC60] John McCarthy. Recursive functions symbolic expressions and their computation by machine. *Communications of the ACM*, 3(4):184–195, 1960.

[MH90] Thanasis Mitsolides and Malcolm Harrison. Generators and the replicator control structure in the parallel environment of ALLOY. In *ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 189–196, White Plains, New York, June 1990.

[Mil85] Robin Milner. The standard ml core language. Technical report (revised), Edinburg University, June 1985.

[Mit] Thanasis I. Mitsolides. *The Design and Implementation of ALLOY, a Parallel Higher Level Programming Language.* PhD thesis, New York University, 251 Mercer Street, New York, NY 10012, **. In preparation.

[Mit87] Thanasis Mitsolides. Highly parallel functions for zlisp. Ultra Computer Documentation Note 12, New York University, 251 Mercer Street, New York NY 10012, June 1987.

[Mit88a] Thanasis Mitsolides. An implementation of ultra prolog. Ultra Computer Documentation Note 10, New York University, 251 Mercer Street. New York, NY 10012, December 1988.

[Mit88b] Thanasis Mitsolides. Supporting or-parallelism in ultra prolog. Internal Report, New York University, May 1988.

[Mit89a] Thanasis Mitsolides. Blocking synchronization primitives on the ultra computer. Ultra Computer Documentation Note 11, New York University, 251 Mercer Street, New York, NY 10012, January 1989.

[Mit89b] Thanasis I. Mitsolides. Features of serial and parallel functional, object oriented and logic programming languages. Survey paper, New York University, 251 Mercer Street, New York, NY 10012, May 1989.

[Mit90a] Thanasis Mitsolides. Logic programming in the parallel environment of ALLOY. Technical report, New York University, 251 Mercer Street, New York, NY 10012, April 1990. Currently Been Written.

[Mit90b] Thanasis Mitsolides. Mu calculous and functional programming in the parallel environment of ALLOY. Technical report, New York University, 251 Mercer Street, New York, NY 10012, March 1990. Currently Been Written.

[Moo86] David A. Moon. Object-oriented programming with flavors. *SIGPLAN OOPSLA Proceedings*, 21:1–8, September 1986.

[MS80] Donals P. McKay and Stuart C. Shapiro. MULTI - a lisp based multiprocessing system. *SIGPLAN Lisp Conference*, 1980.

[Nai86] Lee Naish. Negation and control in prolog. *Lecture Notes in Computer Science*, 238, 1986.

[Nai88] Lee Naish. Parallelizing nu-prolog. In Kenneth A Bowen and Robert A. Kowalsky, editors, *Proceeding of the Fifth International Conference/Symposioum on Logic Programming*, pages 1546–1564, 1988.

[NTU84] H. Nakashima, S. Tomura, and K. Ueda. What is a variable in prolog? *Proceedings of International Conference on Fifth Generation Computer Systems*, pages 327–332, 1984. Tokyo.

[OTO+84] Hiroshi G. Okuno, Ikuo Takeuchi, Nobuyasu Osato, Yasushi Hibino, and Kazufumi Watanabe. Tao: A fast interpreter-centered system on lisp machine elis. *ACM Symposium on functional programming*, pages 140–149, 1984.

[RG86] Jonathan Rees and William Glinger. Revised report on the algorithmic language scheme. *SIGPLAN Notices*, 21(12):37–79, December 1986.

[Rin88] G. A. Ringwood. Parlog86 and the dining logicians. *Communications of the ACM*, 31(1):10–25, January 1988.

[Rob65] J. A. Robinson. A machine oriented logic based on the resoilution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[Rou75] P. Roussel. Prolog: Manual de reference et d' utilisation. Technical documentation, Groupe d' Intelligence Artificielle, Marseille-Luminy, 1975.

[Sak88] Markku Sakkinen. On the darker side of c++. *ECOOP 88, LNCS 322*, pages 162–176, 1988.

[Sar85] Vijay A. Saraswat. Problems with concurrent prolog. Technical Report 86-100, Carnegie-Mellon University, May 1985.

[SDDS86] J. T. Schwartz, R. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets An Introduction to SETL*. Springer-Verlag, 1986.

[Sha83a] Ehud Shapiro. Object oriented programming in concurrent prolog. *New Generation Computing*, 1(1):25–49, 1983. Also in Shapiro, E. (ed.), *Concurrent Prolog*.

[Sha83b] Ehud Shapiro. A subset of concurrent prolog and its interpreter. ICOT Technical Report TR-003, Institute for New Generation Computer Technology, Tokyo, 1983. Also in Shapiro, E. (ed.), *Concurrent Prolog*.

[Sha86a] Ehud Shapiro. Concurrent prolog: A progress report. *IEEE Computer*, 19(8):45–58, August 1986. Also in Shapiro, E. (ed.), *Concurrent Prolog*.

[Sha86b] Ehud Shapiro. Systems programming in concurrent prolog. In M. Canegham and D. H. D. Warren, editors, *Logic Programming and its Applications*, pages 50–74. Ablex Publishing Co., 1986. Also in Shapiro, E. (ed.), *Concurrent Prolog*.

[Sha87] Ehud Shapiro. Or prolog in flat concurrent prolog. In J. L. Lassez, editor, *Proceedings of 4th International Conference of Logic Programming*, pages 311–337. MIT Press, 1987. Also in Shapiro, E. (ed.), *Concurrent Prolog*.

[Sha89] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, September 1989.

[SHHS87] William Silverman, Michael Hirsch, Avshalom Houri, and Ehud Shapiro. The logix system user manual, version 1.21. In Ehud Shapiro, editor, *Concurrent Prolog vol 2*, chapter 21, pages 46–77. MIT Press, 1987.

[SM84] Ehud Shapiro and Colin Mierowsky. Fair, biased and self-balancing merge operators. *New Generation Computing*, 2(3):221–240, 1984. Also in Shapiro, E. (ed.), *Concurrent Prolog*.

[Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. *SIGPLAN OOPSLA Proceedings*, 21:38–45, September 1986.

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. Series in Logic Programming. MIT Press, Cambridge Massachusetts, July 1986.

[Ste87] Lynn Andrea Stein. Delegation is inheritance. *ACM OOPSLA Proceedings*, pages 138–146, October 1987.

[Str86] Bjarne Stroustrup. An overview of c++. *SIGPLAN Notices*, 21(10):7–18, October 1986.

[Str87] Bjarne Stroustup. What is "object-oriented programming". *Lecture Notes in Computer Science*, 276:51–78, june 1987.

[TF86] Akikazu Takeuchi and Koichi Furukawa. Parallel logic programming languages. In Ehud Shapiro, editor, *Proceedings* 3^{rd} *International Conference on Logic Programming, LNCS 225*, pages 242–255. Springer Verlag, 1986. Also chapter 6 in "Shapiro E. (ed.), *Concurrent Prolog*".

[TOO83] Ikuo Takeuchi, Hiroshi Okuno, and Nobuyasu Ohsato. Tao — a harmonic mean of lisp, prolog and smalltalk. *ACM SIGPLAN Notices*, 18(7):65–74, July 1983.

[Tur86] David Turner. An overview of miranda. *SIGPLAN Notices*, 21(12):158–166, December 1986.

[Ued86] Kazunori Ueda. Guarded horn clauses. *Logic Programming, LNCS*, 221:168–179, 1986. Also in Shapiro, E. (ed.), *Concurrent Prolog.*

[Ued87a] Kazunori Ueda. Making exhaustive search programs deterministic. *New Generation Computing*, 5:29–44, 1987. ICOT Research Center.

[Ued87b] Kazunori Ueda. Making exhaustive search programs deterministic, part ii. In *Proceedings of the Fourth International Conference on Logic Programming.* MIT Press, May 1987. Melbourne, Australia.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. *ACM OOPSLA Proceedings*, pages 227–242, October 1987.

[Wag90] Peter Wagner. Concepts and paradigms of object-oriented programming. *OOPS Messenger SIGPLAN*, 1(1):7–87, August 1990.

[Wan80] Mitchell Wand. Continuation-based multiprocessing. *SIGPLAN Lisp Conference*, pages 19–28, 1980.

[War77] D. H. D. Warren. Implementing prolog — compiling predicate logic programs. Technical Report DAI 39/40, University of Edinburg, 1977.

[War80] D. H. D. Warren. Logic programming and compiler writing. *Software-Practice and Experience*, 10(2), 1980.

[War87] David H. D. Warren. Or-parallel execution models of prolog. In *TAPSOFT '87*, pages 243–259. Joint Conference on Theory and Practice of Software Development, March 1987. University of Manchester, Manchester M13 9PL.

[Weg76] Peter Wegner. Programming languages the first 25 years. *IEEE Transactions on Computers*, pages 1207–1225, December 1976.

[Weg87] Peter Wegner. Dimensions of object-based language design. *ACM OOPSLA Proceedings*, pages 168–182, October 1987.

[Weg89] Peter Wegner. Learning the language. *BYTE*, 14(3):245–253, March 1989.

[Wil88] J. M. Wilson. *Operating System Data Structures for Shared-Memory MIMD Machines with fetch-and-ADD.* PhD thesis, Nyw York University, 251 Mercer Street, New York, NY 10012, June 1988.

[Woo89] D. Wood. Parallel queues and pools and their performance. Master's thesis, New York University, 10^{th} floor, 719 Broadway, New York, NY 10003, january 1989.

[WZ88] Peter Wegner and Stanley B. Zdonik. Inheritance as an incremental modification mechanism. *Notes In Computer Science: ECOOP*, LNCS 322, 1988.

[YA86] Rong Yang and Hideo Aiso. P-prolog: A parallel logic language based on exclusive relation. *3rd International Logic Programming Conference, LNCS*, 225:255–269, July 1986. Imperial College, London, UK.