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 gratitude 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
**CONTENTS**

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
4 Functions, Closures, and their Evaluation Modes

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 let 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
## CONTENTS

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 Inefficient 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
<table>
<thead>
<tr>
<th>A The Interpreter of ALLOY in Common Lisp</th>
<th>141</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1 Top level</td>
<td>141</td>
</tr>
<tr>
<td>A.1.1 alloy.l</td>
<td>141</td>
</tr>
<tr>
<td>A.1.2 load.l</td>
<td>143</td>
</tr>
<tr>
<td>A.1.3 inter.l</td>
<td>145</td>
</tr>
<tr>
<td>A.2 Main Interpreter</td>
<td>151</td>
</tr>
<tr>
<td>A.2.1 fcall.l</td>
<td>151</td>
</tr>
<tr>
<td>A.2.2 ocall.l</td>
<td>161</td>
</tr>
<tr>
<td>A.2.3 rcall.l</td>
<td>168</td>
</tr>
<tr>
<td>A.2.4 gcall.l</td>
<td>175</td>
</tr>
<tr>
<td>A.3 Pre-processor</td>
<td>181</td>
</tr>
<tr>
<td>A.3.1 prep.l</td>
<td>181</td>
</tr>
<tr>
<td>A.3.2 gprep.l</td>
<td>187</td>
</tr>
<tr>
<td>A.3.3 oprep.l</td>
<td>191</td>
</tr>
<tr>
<td>A.4 Utilities</td>
<td>193</td>
</tr>
<tr>
<td>A.4.1 io.l</td>
<td>193</td>
</tr>
<tr>
<td>A.4.2 values.l</td>
<td>197</td>
</tr>
<tr>
<td>A.4.3 alutil.l</td>
<td>200</td>
</tr>
<tr>
<td>A.4.4 util.l</td>
<td>204</td>
</tr>
<tr>
<td>A.5 Built in functions and objects</td>
<td>207</td>
</tr>
<tr>
<td>A.5.1 build.l</td>
<td>207</td>
</tr>
<tr>
<td>A.5.2 build2l</td>
<td>214</td>
</tr>
<tr>
<td>A.5.3 obuild.l</td>
<td>220</td>
</tr>
<tr>
<td>A.6 Library in ALLOY</td>
<td>230</td>
</tr>
<tr>
<td>A.6.1 alloylib.a</td>
<td>230</td>
</tr>
<tr>
<td>A.6.2 lists.a</td>
<td>231</td>
</tr>
<tr>
<td>A.6.3 bool.a</td>
<td>232</td>
</tr>
<tr>
<td>A.6.4 loop.a</td>
<td>233</td>
</tr>
<tr>
<td>A.6.5 misc.a</td>
<td>234</td>
</tr>
<tr>
<td>A.6.6 hiherf.a</td>
<td>235</td>
</tr>
<tr>
<td>A.6.7 fp.a</td>
<td>237</td>
</tr>
<tr>
<td>A.7 Creating ALLOY</td>
<td>238</td>
</tr>
</tbody>
</table>
B Benchmarks

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 naïve 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 $\mu$ 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, caching, 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 maintenance 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 successful.

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 than, 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) (1return (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)))
```
2.1 Basic Functional

Test runs:

```
ALLOY > (fact 25)
==> 1551121004333098598400000
```

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)
==> 1551121004333098598400000
```

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 (((issmall lgreat) (partition (car l) (cdr l))))
      (psortrest issmall (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:
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))))
(isetrees '(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)
2.3 Lazy Evaluation

An object process \texttt{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:

\begin{verbatim}
ALLOY > (set p2 ('new proca (mu()) (1return (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)
\end{verbatim}

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:

\begin{verbatim}
(setfun ints(n)
  (return (cons n (ints (sum n 1))))
\end{verbatim}
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:

```lisp
(setfun fibonacci()
  (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 (fibonacci))
ALLOY > (mapcar data (sublist 6 9 f))
==> (13 21 34 55)
ALLOY > (mapcar data (sublist 50 55 f))
==> (20365011074 32951280099 53316291173 86267571272 139583862445 228851433717)
```

### 2.3.3 Prime Numbers

Function `primes` returns lazily the list of all prime numbers in order:
(setfun 1primes()
    (return (lazy (primes))))

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

(setfun sieve(1)
    (return (cons (car 1) (sieve (filter (car 1) (cdr 1)))))

(setfun filter(x l)
    (if (equal (remainder (car 1) x) 0) (return (filter x (cdr 1)))
        (return (cons (car 1) (filter x (cdr 1)))))

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

Test runs:

ALLOY > (set p (1primes))
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 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89)
ALLOY > (data (nth 15 p))
=> 43
ALLOY > p
=> (1 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89)

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 1)
    (return (let ((rest (bb-1 length 1)))
        (lazy (bb-2 1 rest))))))
(setfun bb-1(n l)
  (if (lt n 1) (return 1)
      [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 1) (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 9th element of fb the original stream f got evaluated up to its 14th 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.
2.4 Networks

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

\begin{verbatim}
(retrieve "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 1 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:
\end{verbatim}

\begin{verbatim}
ALLOY > (ssort '(5 2 7 3 5 1 4 2 3))
=> (1 2 2 3 3 4 5 5 7)
\end{verbatim}

2.4.2 Merge Sort

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

\begin{verbatim}
(retrieve "merge")
(retrieve "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)))))
\end{verbatim}
The definition of **amerge** follows:

```
(provide "merge")

(setfun amerge(11 12)
 (when (null-p 11) (return 12))
 (when (null-p 12) (return 11))
 (when (le (car 11) (car 12))
   (return (cons (car 11) (amerge (cdr 11) 12))))
 (when (ge (car 11) (car 12))
   (return (cons (car 12) (amerge (cdr 12) 11))))
```

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 $2^x \ast 3^y \ast f^z \forall x, y, z \geq 0$:

```
(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 \texttt{intsxy} returns one or each number in a range (depending on how it has been called).

\begin{verbatim}
(setfun intsxy(x y)
  (when (leq x y)
   (return x)
   #!-(return *(intsxy (sum x 1) y))))
\end{verbatim}

Test runs:

\begin{verbatim}
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 ?))
=> ((1 5) (1 6) (1 7) (2 5) (2 6) (2 7))
ALLOY > #*(intsxy 1 *(intsxy 1 5))
=> (1 2 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)
)
\end{verbatim}

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 \texttt{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 \texttt{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 \texttt{intersection}. Also, all functions can be used either normally or as generators.

\begin{verbatim}
(setfun getmember(l)
  (when (cons-p l) (return (car l)) #!-(return *(getmember (cdr l)))))
\end{verbatim}

\begin{verbatim}
(setfun intersection(s1 s2)
  #(let (x *(getmember s1)) (y *(getmember s2))
    (when (eqv x y) (return x)))
\end{verbatim}
(setfun ismember(x 1)
    (if (eql x *(getmember 1)) (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 leaves 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))
2.5 Backtracking and Searching

```
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(1)
  (if (cons-p 1)
      (list #(let ((x rest) *(delete-one 1))
            #(return (cons x *(permutations rest))))
     (return nil)))

(setfun delete-one(1)
  (when (cons-p 1)
    (return (list (car 1) (cdr 1)))
    #((return (let (((x rest) *(delete-one (cdr 1))))
     (list x (cons (car 1) 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)
  (let ((list (make-list n))
    #(let ((poslist (board list *(permutations list))))
      (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 course, 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 (1st ()
    (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 enqueue enqueue)
  (interface dequeue dequeue))

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

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

(setq 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 been processed.

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

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

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

(setq p(n)
  (ps n) (return %t)]

(setq v(n)
  (vs n) (return %t)]

(setq ps(n)
  (repeat (if n n 1) ~(data ('dequeue q))))

(setq vs(n)
  (repeat (if n n 1) ~(('enqueue q 't))))

(class-end semaphore)
2.6 Parallel Objects

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 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 println)
  (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 diner()
(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
3.2 Essential expressions and primitives

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.

- \([\ldots]\)
  Similar to (\ldots) 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

- \((\mathtt{mu\ args-pattern\ e_1 e_2 \cdots e_n})\)
  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
referred 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 1return
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
c 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.

- \texttt{(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

- \texttt{# 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.

- \texttt{! 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.

- \texttt{* expr}
  
  If argument is a function call the function behaves as a generator. It also returns values on demand. See \texttt{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

- \texttt{(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 \texttt{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

- (**eq! e1 e2 ... en**)  
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.

- \(\text{\texttt{(lreturn e)}}\)
  The local return. Affects the local-most function.

- \(\text{\texttt{(lfail)}}\)
  The local failure. Affects the local-most function.

- \(\text{\texttt{(var-p variable-name)}}\)
  Succeeds if variable is not set. It can be implemented in terms of an enhanced \texttt{sa-var} class.

- \(\text{\texttt{\_other-name}}\)
  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 \texttt{ma-var} class.

Macros

The following expressions are simple macros:

- \(\text{\texttt{(setfun name vars-pattern e_1 e_2 \ldots e_n)}}\)
  The same with \(\text{\texttt{(set name \{mu vars-pattern e_1 e_2 \ldots e_n\})}}\) 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.
3.3 Non-essential special forms and primitives

- \textbf{setfun (name vars-pattern) } e_1e_2\ldots e_n \\
  The same as above.

- \textbf{let ((vars-pattern expression) \ldots) be}_1 \ be_2 \ldots \ 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.

- \textbf{lets ((vars-pattern expression) \ldots) be}_1 \ be_2 \ldots \ be_n \\
  Like the let expression but returns a failure value by default. An return expression can be used to return something else.

- \textbf{letrec ((vars-pattern expression) \ldots) be}_1 \ be_2 \ldots \ be_n \\
  Like the let expression but all expression can access the variables in vars-pattern. Useful for the definition of recursive local functions.

- \textbf{letrecs ((vars-pattern expression) \ldots) be}_1 \ be_2 \ldots \ be_n \\
  Like the letrec expression but does not return anything by default. An return expression can be used to return something.

- \textbf{when } e_1 \ e_2 \ldots 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 e1 e2 e3)`
  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 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:
\begin{itemize}
\item (\textit{name-p object})
\end{itemize}
Succeeds only when the \textit{object} is an instance of the class with name \textit{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 \texttt{\`new} message to its class etc.

\textbf{List management}

Since lists (objects of class \texttt{cons}) are very useful objects, the following functions are pre-defined to ease their use:

\begin{itemize}
\item (\texttt{car e})
\end{itemize}
The \texttt{car} (first element) of a pair (an object).

\begin{itemize}
\item (\texttt{cdr e})
\end{itemize}
The \texttt{cdr} (second element) of a pair.

\begin{itemize}
\item (\texttt{cons e_1 e_2 \ldots e_n})
\end{itemize}
Groups the expressions in a list of pairs.

\begin{itemize}
\item (\texttt{list e_1 e_2 \ldots e_n})
\end{itemize}
Like \texttt{cons} but the last element of the last pair is the special atom \texttt{nil}.

\begin{itemize}
\item (\texttt{null-p e})
\end{itemize}
Same as \texttt{(eq e nil)}.

\begin{itemize}
\item (\texttt{nth n e})
\end{itemize}
Will return the \(n^{th}\) element of list \texttt{e}.

\begin{itemize}
\item (\texttt{carn n e})
\end{itemize}
Will return the \(n^{th}\) \texttt{car} of list \texttt{e}.

\begin{itemize}
\item (\texttt{cdrn n e})
\end{itemize}
Will return the \(n^{th}\) \texttt{cdr} of list \texttt{e}.

\begin{itemize}
\item (\texttt{sublist i j e})
\end{itemize}
Will return the part of list \texttt{e} from position \texttt{i} to position \texttt{j}. 

• \((\text{append } l_1 \ l_2 \ \cdots \ l_n)\)
  Returns the concatenation of the lists given to it as arguments.

• \((\text{length } l)\)
  Will return the length of list \(l\).

• \((\text{mapcar } \text{function\_message } l)\)
  Will return the list of the results of the application of \text{function\_message} to each of
  the elements of list \(l\).

\textbf{Square List management}

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

• \((\text{sca}r \ e)\)
  The car (first element) of a square pair (an object).

• \((\text{sccdr} \ e)\)
  The cdr (second element) of a square pair.

• \((\text{scons} \ e_1 \ e_2 \ \cdots \ e_n)\)
  Groups the expressions in a list of pairs.

\textbf{Arithmetic Functions}

Here are some messages accepted by numbers:

• \((\text{'}\text{string } \text{number})\)
  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:

• \((\text{sum } \text{number} \ \cdots)\)
  Will return the sum of its arguments. Like the functions described bellow, the
  operation treats its arguments in a left associative way.

• \((\text{diff } \text{number} \ \cdots)\)
  Will return the difference of its arguments.
3.3 Non-essential special forms and primitives

- \(\text{times \ number \ ...}\)
  Will return the product of its arguments.

- \(\text{div \ number \ ...}\)
  Will return the division of its arguments.

- \(\text{quotient \ number \ ...}\)
  Will return the quotient of the division of its arguments.

- \(\text{remainder \ number \ ...}\)
  Will return the remainder of the division of its arguments.

- \(\text{eq \ number \ ...}\)
  Returns first argument if all arguments are equal numbers, otherwise fails.

- \(\text{lt \ number \ ...}\)
  Returns first argument if all arguments are numbers and of each pair of successive ones the first is less than the second, otherwise fails.

- \(\text{gt \ number \ ...}\)
  Returns first argument if all arguments are numbers and of each pair of successive ones the first is greater than the second, otherwise fails.

- \(\text{le \ number \ ...}\)
  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.

- \(\text{ge \ number \ ...}\)
  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:

- \(\text{'string \ atom}\)
  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 s₁ s₂ ... sₙ)
  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 l₁ ... (function lₙ value)).

• (filter predicate list)
  Equivalent to (function l₁ ... (function lₙ value)).

• (mapcar2 function list₁ vlist2)
  Applies function to pairs of elements from the two lists with the same position.
  Returns the list of results.

• (mapcarn function l₁ l₂ ... lₙ)
  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.

- \( (h\text{-}bu \text{ function value}) \)
  Returns a unary function which when applied to an argument \( a \) returns \( (\text{function value } a) \).

- \( (h\text{-}rev \text{ function}) \)
  Returned function expects the arguments of \( \text{function} \) in the reverse order. Special case is function \( h\text{-}rev2 \) for binary functions.

- \( (h\text{-}red \text{ function value}) \)
  Returned function does reduction using \( \text{function} \) and an initial value of \( \text{value} \).

- \( (h\text{-}dup \text{ function}) \)
  Returns a unary function which when applied to an argument \( a \) returns \( (\text{function } a \ a) \).

- \( (h\text{-}comp } f_1 f_2 \)
  Returned function is a composition of argument functions.

- \( (h\text{-}compn } f_1 f_2 \ldots f_n \)
  Returned function is a composition of argument functions.

- \( (h\text{-}const } value \)
  Returned function always returns \( \text{value} \).

- \( (h\text{-}construct } f_1 f_2 \ldots f_n \)
  Returned function takes \( n \) arguments and passes one to each unary function \( f_i \).

- \( (h\text{-}mapn } f_1 f_2 \ldots f_n \)
  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 \( h\text{-}map \) and \( h\text{-}map2 \).

FP support

A number of predicates facilitating programming in FP style is provided in library file \( fp \). It can be loaded using the \textbf{require} expression. All of these functions (except \textbf{fp-fun})
expect one physical argument which is a list of the actual arguments.

- \textbf{(fp-fun function)}
  
  Returned function expects all the arguments of the original function in one list.

- \textbf{(fp-rev function)}
  
  Returned function expects the argument list in the reverse order.

- \textbf{(fp-trans (list\_1 list\_2 \ldots 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.

- \textbf{(fp-distl (value list))}
  
  Returns a list whose elements are lists of two elements the first of which is \textit{value} and the second the corresponding element from list.

- \textbf{(fp-distr (value list))}
  
  Distributes \textit{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 cannot be defined in ALLOY itself.

**Input**

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

- \texttt{('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 "\texttt{console}".

- \texttt{('read input)}
  
  Will read an expression from object \texttt{input}. 
3.4 Recommended Libraries

- (`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`.

- (`println 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.
3.4 Recommended Libraries

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 \ldots e_n)`
  Returns the conjunction of the arguments values as soon as possible.

• `(or e_1 e_2 \ldots e_n)`
  Returns the disjunction of the arguments values as soon as possible.
3.4 Recommended Libraries

- **(not e)**
  Returns the negation of the argument’s value.

- **(and-call c₁ c₂ ⋯ cₙ)**
  Returns the conjunction of the results returned by calling the arguments, as soon as possible. Kills incomplete calls as soon as possible.

- **(or-call c₁ c₂ ⋯ cₙ)**
  Returns the disjunction of the results returned by calling the arguments, as soon as possible. Kills incomplete calls as soon as possible.

- **(equal e₁ e₂ ⋯ eₙ)**
  Like eq1 but works for list as well.

Blocks

- **(block₁ e₁ e₂ ⋯ eₙ)**
  Returns the value of the first parameter.

- **(blockn e₁ e₂ ⋯ eₙ)**
  Returns the value of the last parameter.

- **(block e₁ e₂ ⋯ eₙ)**
  Returns t.

Loops

- **(while cₖ cₑ)**
  Calls cₑ if cₖ when called succeeds. Repeats as soon as evaluation of cₑ has completed.

- **(until cₖ cₑ)**
  Calls cₑ unless cₖ when called succeeds. Repeats as soon as evaluation of cₑ has completed.

- **(repeat n cₑ)**
  Calls cₑ n times in parallel.

- **(srepeat n cₑ)**
  Calls cₑ n times serially.
System Functions

- **(include filename)**
  Includes the contents of the file as if it were typed from the terminal and read at the top level.

- **(load filename)**
  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
3.4  Recommended Libraries

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
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

---

1Environments 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 call1 call2 ... calln)`. 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 $\mu$ is the letter after $\lambda$ 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` (`return` 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 `returning` 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.
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 forms 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 `return` 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:

\[
\text{\texttt{(\texttt{\textasciitilde lets} (\texttt{(\texttt{\textasciitilde p1 e1}) \cdots (\texttt{\textasciitilde pk ek}) \cdots (\texttt{\textasciitilde pn en}))}) b_1 \cdots b_i \cdots b_n)^\texttt{\textasciitilde}}} 
\]

is expanded to:

\[
\text{\texttt{(\texttt{\textasciitilde (\texttt{\textasciitilde mu(p1 \cdots pk \cdots pn) \ b1 \cdots b_i \cdots b_n})}) b_1 \cdots b_i \cdots b_n)^\texttt{\textasciitilde}}} \]

\texttt{e_1 \cdots e_i \cdots e_n)^\texttt{\textasciitilde}}
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 \texttt{let}s 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 \texttt{set} expression in the 
body.

\subsection*{4.4.2 The \texttt{let} expression}

The second form, \texttt{let}, has semantics closer to the ones of \texttt{let} in LISP, but its expansion 
is even more complex than the expansion of \texttt{let}s because of the need to behave as a 
generator\footnote{Since a generator can be controlled, serial and parallel evaluation mode must be preserved.} when its body is a generator.

\begin{equation}
\begin{aligned}
\{^i\text{let} & \ (^l(p_1 \ e_1) \cdots (p_k \ e_k) \cdots (p_n \ e_n))^j \\
& b_1 \cdots b_k \cdots b_n)^6 \}
\end{aligned}
\end{equation}

is expanded to:

\begin{equation}
\begin{aligned}
\{^i\text{let} & \ (^l(\text{return} \ #(^l\text{mu dummy} \ #(\text{return} \ b_n))^6 \\
& b_1 \cdots b_k \cdots b_{n-1})^6)) \\
& e_1 \cdots e_k \cdots e_n)^i \}
\end{aligned}
\end{equation}

\subsection*{4.5 Comparisons}

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

\subsubsection*{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.
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:

```lisp
(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)

(defun par:sys:or:n (all)
  (declare (zLISPopts no!edefine 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))))
  (declare (zLISPopts no!edefine 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))))))
```

4.5 Comparisons

63
Its is clearly a complex and involved program.

Here is the implementation in ALLOY:

\[
\begin{align*}
\text{(setfun \text{or-call .} 1)} \\
\text{(when \text{cons-p} l) \text{(lets \{rv xv\)}})} \\
\quad \begin{align*}
\text{(when \text{set xv} ((car l)))} & \quad \text{(return xv)} \\
\text{(when \text{set rv} \text{(or-call . \}cdr l\})}) & \quad \text{(return rv)}}
\end{align*}
\end{align*}
\]

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 [KH89,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 \textit{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.

\[
\begin{align*}
\text{(defmacro \text{par-or}(Expr1 Expr2)}} \\
\quad \text{'}(\text{let ((x nil) e1 e2)}} \\
\quad \text{(setq x \text{(future \text{(or \{future \text{(if \text{setq e1 ,Expr1 \}\text{determine x e1}})})}}))} \\
\quad \text{(future \text{(if \text{setq e2 ,Expr2 \}\text{determine x e2}})}))} \\
\text{(touch x))}
\end{align*}
\]

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

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

\[
\begin{align*}
\text{(setfun \text{or-call e1 e2)}} \\
\text{(lets \{(r1 (e1))\} \text{(when r1 (return r1))}} \\
\text{(lets \{(r2 (e2))\} \text{(when r2 (return r2))}})
\end{align*}
\]

Since ALLOY does not have macros, the arguments must be prefixed with operator \texttt{^} 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
4.5 Comparisons

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))))))
(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]:

```lisp
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:

```lisp
(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)
```

...
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 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. 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 \texttt{bubble-up} function which surfaces the $n^{th}$ element of a list at its head. Auxiliary function \texttt{delete-nth} deletes the $n^{th}$ element of the list, returns the resulting list and back-communicates the deleted value through the \texttt{gvalue} object:

\begin{verbatim}
(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 1) v))))))
\end{verbatim}

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

\subsection{Communication to Many}

In the above situations communication to one process is called \textit{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.

\subsection{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 \textit{static} or \textit{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.

```scheme
(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.

```scheme
(provide "estack")

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

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

(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 \texttt{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 \texttt{empty} and \texttt{full}.

A parallel index is used to access the next available slot in the array. A message arriving first increments that index using \texttt{fetch-and-add}. That index modulo the size of the array points at the current slot. The message waits on the \texttt{empty} flag of that slot. Then it creates and places in that slot a set of dynamic variables. After that, setting the \texttt{full} flag it declares that the slot is filled. Finally, it waits on the \texttt{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 \texttt{next} dynamic variables it created itself and placed in the current slot, and starts execution. The previous slot is then freed by setting its \texttt{empty} flag and re-setting its \texttt{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).

\footnote{\texttt{fetch-and-add} is an atomic operation which allows a process 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:

```
(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)))))))
```

Function hamming returns, sorted, all numbers of the form $2^i3^j5^k$, $i, j, k > 0$. 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.

(resecure "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 enqueue-queue enqueue)
 (interface dequeue-queue 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))
5.4 Examples

(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(1)
    (return (if (cons-p l) (cons (cons ('id (car l)) (car l))
                                 (idsemalist (cdr l)) nil)))

(setfun all-key(1)
    (if (cons-p l)
        (if (and (eql ('object-p (car l)) 'key) (all-key (cdr l)))
            (return ())
            (return %t))
        (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:

```
(requires "key")
(requires "lock")

class-start philosopher
  (static id forks)
  (import gt diff println 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)
    [println (list "Philosopher" id "took both forks")]
    [println (list "Philosopher" id "started eating")]
    (eat 2)
    [println (list "Philosopher" id "stopped eating")]
    [println (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)
  
  [println "The big feasting starts..."
   (dinen2 (if p p 5) (if n n))]

(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:

(let (delete-nthx)
  (setfun bubble-up (n 1)
    (lreturn (let (((gval value) (gvalue)))
      (cons value (delete-nthx n 1 gval)))))
  (setfun delete-nthx (n 1 v)
    (when (cons-p 1)
      (lreturn (if (le n 1)
        (block1 (cdr 1) ('put v (car 1)))
        (cons (car 1) (delete-nthx (diff n 1) (cdr 1) 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:

```lisp
(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:

```lisp
(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)
  ==> %

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:

```
(defun (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)))))))))

(defun (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:

```
(defun 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
5.7 Comparison with Other Programming Languages

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

Actors have 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 imple-
mentation 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]:

```prolog
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:

```prolog
?- 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 enqueuing. 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 pro-
gramming, 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]:

```c
void phil(int i)
{ while(1) { think();
    in("room ticket"); in("chopstick", i);
    int "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.
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:

```alloy
(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
5.8 Summary

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:

```lisp
(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.

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 [Di75,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 coroutines.

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 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 nondeterministically. 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 \texttt{factory} object\(^2\). This is a library object (implemented in ALLOY) which in effect creates co-expressions. When an instance of a \texttt{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 \texttt{factory} class can be defined in terms of classes \texttt{coex} (chapter 5) and \texttt{vector} (chapter 3).

Just as \(e\) is equivalent to (quote e) in lisp, \(*e\) is equivalent to (mu #(1return 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 \texttt{factory} object uses replicators and generators itself. This object should be implemented at a lower level.

\(^3\)Note that \texttt{mu} is the function creation function like \texttt{lambda} is for Lisp.
("\text{nth f 3}\)

The above expression will return the 3rd 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 \textit{replicable} expression is replicated in such a way that its enclosed generators are replaced by their values. The program fragment:

\[(^c... \#(\ldots *e_1 \ldots *e_i \ldots *e_n \ldots)^r \ldots)^c\]

is transformed to:

\[
\begin{align*}
\text{\texttt{let}} & \quad (f_1 ('\text{new factory} \ ^{*}e_1)) \\
& \quad (f_i ('\text{new factory} \ ^{*}e_i)) \\
& \quad \cdot \quad \cdot \\
& \quad (f_n ('\text{new factory} \ ^{*}e_n)) \\
& \quad \text{\texttt{rep}} (\text{mu} (p_1 \ldots p_k \ldots p_n)) \\
& \quad \quad \quad \text{\texttt{return}} ('\ldots p_1 \ldots p_k \ldots p_n \ldots)^r)) \\
& \quad (^c \ldots ('\text{rep} ('\text{nth f}_1 \ 1) \ \ldots ('\text{nth f}_i \ 1) \ \ldots ('\text{nth f}_n \ 1)) ^r) \\
& \quad ('\text{rep} ('\text{nth f}_1 \ 1) \ \ldots ('\text{nth f}_i \ 1) \ \ldots ('\text{nth f}_n \ 2)) ^r) \\
& \quad ('\text{rep} ('\text{nth f}_1 \ 1) \ \ldots ('\text{nth f}_i \ 1) \ \ldots ('\text{nth f}_n \ m_n)) ^r \\
& \quad \cdot \quad \cdot \quad \cdot \\
& \quad ('\text{rep} ('\text{nth f}_1 \ 1) \ \ldots ('\text{nth f}_i \ 2) \ \ldots ('\text{nth f}_n \ 1)) ^r) \\
& \quad ('\text{rep} ('\text{nth f}_1 \ 1) \ \ldots ('\text{nth f}_i \ 2) \ \ldots ('\text{nth f}_n \ 2)) ^r) \\
& \quad ('\text{rep} ('\text{nth f}_1 \ 1) \ \ldots ('\text{nth f}_i \ 2) \ \ldots ('\text{nth f}_n \ m_n)) ^r \\
& \quad \cdot \quad \cdot \quad \cdot \\
& \quad ('\text{rep} ('\text{nth f}_1 \ 1) \ \ldots ('\text{nth f}_i \ m_i) \ \ldots ('\text{nth f}_n \ m_n)) ^r \\
& \quad \cdot \quad \cdot \quad \cdot \\
& \quad ('\text{rep} ('\text{nth f}_1 \ m_1) \ \ldots ('\text{nth f}_i \ m_i) \ \ldots ('\text{nth f}_n \ m_n)) ^r \\
& \quad \ldots )^c
\end{align*}
\]

where generators \(e_1 \ldots e_n\) 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 \texttt{rep} the generators must be grouped\footnote{One way to do this is to place generators as arguments of a call to function \texttt{list}. That call should then be concurrent or serial as needed.} accordingly. Of course, function \texttt{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\footnote{This helps to define functions simulating conditional expressions such as \texttt{when}. However, these functions would then have to take closures as arguments.}. 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 \texttt{#} in place of the regular replicator \texttt{#}. 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:

\[(c \ldots \ast (r \ldots \ast e_1 \ldots \ast e_i \ldots \ast e_n \ldots r \ldots c)\]

is transformed to:

\[
\ast (\text{\texttt{\textcolor{red}{let}}} \begin{array}{l}
(f_i \ \text{\texttt{\textcolor{red}{('new factory \ ^*e_i')}}}) \\
(f_i \ \text{\texttt{\textcolor{red}{('new factory \ ^*e_i')}}}) \\
\vdots \\
\end{array})
\]
\begin{verbatim}
(fn ('new factory ^*e_n))
(replmu (p_1 ... p_i ... p_n))
   #(1return *("rep ('n nth f_1 1) ... ('n nth f_n 1)")^)
   #(1return *("rep ('n nth f_1 1) ... ('n nth f_n 2)")^)
   #(1return *("rep ('n nth f_1 1) ... ('n nth f_n m_n)")^)
   ... ...
   #(1return *("rep ('n nth f_i 2) ... ('n nth f_n 1)")^)
   #(1return *("rep ('n nth f_i 2) ... ('n nth f_n 2)")^)
   #(1return *("rep ('n nth f_i 2) ... ('n nth f_n m_n)")^)
   ... ...
   #(1return *("rep ('n nth f_i m_1) ... ('n nth f_n m_n)")^)
\end{verbatim}

where the internal generators are those in the scope of the external one and the calls to repl (as well as repl 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

(\texttt{(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) #(1return (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.

\[(\text{print w (list #*(ints 1 5))})\]

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

\[(\text{print w (+ #*(ints 1 5))})\]

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:

\[\text{[let()] #(print w *(ints-inc 1 5))} \]

\[\text{[let()] (print w 1) (print w 2) (print w 3) (print w 4) (print w 5)}\]

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:

\[\text{[let()] #(print w (cons *(ints 1 4) *(ints 5 8)))}\]

\[\text{[let()] #[let ((v1 *(ints 1 4)) (v2 *(ints 5 8))) (print w (cons v1 v2))]}\]

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:
6.3 Examples

(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 \((\sum_{i=1}^{5} i) \times (\sum_{i=6}^{8} i - 5) = 15 \times 6 = 90\) pairs of numbers. These will be some
permutation of the sequence: \((1 . 6) \ldots (1 . 8) (1 . 6) \ldots (1 . 8) (2 . 6) \ldots (2 . 8) (1 . 6) \ldots (3 . 8) \ldots (5 . 8)\).

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 (de-
pending 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 ([[list (make-list n)]
     #(lets ([[bd (board lst * (permutation lst))]]
       (unless (unsafeall bd) (return bd)))))))
(setfun unsafe-all(board)
  (when (cons-p board)
    (when (unsafe (car board) (cdr board)) (return))
    (when (unsafe-all (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(1)
  #(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))
6.3 Examples

\[
\#(\text{return } *(\text{all-ints } (\text{sum middle 1 } y))))
\]

(setfun delete-nth(n l)
   (return (let (((bc val) ('new become))
      (list val (delete-nth x n 1 bc))))))

(setfun delete-nthx(n 1 bc)
   (when (cons-p 1
      [when (le n 1)
         ('make bc (car l)) (return (cdr l))]
      (when (gt n 1)
         (return (cons (car 1)
            (delete-nthx (diff n 1) (cdr 1)
               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 \text{permutation} \) in \(O(\max(\frac{n!}{\text{cpus}}, n \log n))\) time. The number of available processing elements is \text{cpus}.

Even though the last program achieves higher processor utilization it also has considerably greater overhead. This is because function \text{delete-one} in the last program requires quadratic processing power, while in the first program only linear processing power.

Note that function \text{delete-nth} makes use of the “value to become” object \text{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.
(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. 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

\[\text{\footnote{This is true when the parameters of a predicate are used both for input and output. In such a case logical variables/unification (objects) will be needed.}}\]
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 \texttt{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:

\[
\text{append}([\text{H} | \text{T}], \text{L}, [\text{H} | \text{T}2]):- \text{append}(\text{T}, \text{L}, \text{T2}).
\]

\[
\text{append}([], \text{L}, \text{L}).
\]

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)
7.2 Logic Programming and Styles

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 uninstatiated, 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:

\[
\begin{align*}
\text{fibonacci}(X, Y, Y). \\
\text{fibonacci}(X, Y, V) & : - Z \text{ is } X + Y, \text{fibonacci}(Y, Z, V).
\end{align*}
\]

Finally predicate length returns the length of a list. For similar reasons it is written in a functional style:

\[
\begin{align*}
\text{length}([], L) & : - L \text{ is } 0. \\
\text{length}([\_ | T], L) & : - \text{length}(T, L2), L \text{ is } L2 + 1.
\end{align*}
\]

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:

\[
\begin{align*}
\text{intersection}(L1, L2, L) & : - \text{setof}(X, \text{common}(L1, L2, X), L). \\
\text{common}(L1, L2, L) & : - \text{member}(X, L1), \text{member}(X, L2). \\
\text{member}(X, [\_ | T]) & : - \text{member}(X, T).
\end{align*}
\]

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:

```prolog
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:

```prolog
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:

```prolog
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 (((1) (lvs 1)))
      (snest "!*((unify p1 ()) "!*((unify p2 l) "!*((unify p3 l) "*(return))])
      [lets (((1 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(11 12)
    (return (if (cons-p 11) (cons (car 11) (append (cdr 11 12))) 12)))
```

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:
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 11 12] (copyloge (list lp1 lp2 lp3))]]
    [#(lets [[[x2 11_2] *(memberp x 11)]
              [[x3 12_2] *(memberp x 12)]
              (when (unify x2 x3) (return (list x3 11_2 12_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 11 12] (copyloge (list lp1 lp2 lp3))]]
    #[lets [[[x2 11_2] *(memberp x 11)]
              #[lets [[[x3 12_2] *(memberp x2 12)]
                        (return (list x3 11_2 12_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)
7.3 Comparisons

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

(setfun getmember(1)
  (when (cons-p 1) (return (car 1))
    #!(return *(getmember (cdr 1)))))

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(11 12)
  (when (null-p 11) (return 12))
  (when (null-p 12) (return 11))
  (when (cons-p 11) (return (cons (car 11) (merge 12 (cdr 11)))))
  (when (cons-p 12) (return (cons (car 12) (merge (cdr 12) 11))))

The same definition can be called lazily. If recursive calls were made replicative this func-
tion 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 isotreex in ALLOY:

(setfun isotreex(t1 t2)
  (if (and (cons-p t1) (cons-p t2))
    (if (or-call ~(and-call ~(isotreex (car t1) (car t2)))
      ~(isotreex (cdr t1) (cdr t2))))
      ~(isotreex (car t1) (cdr t2)))
~(isotreex (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:

\[
\begin{align*}
\text{setfun squares}() &= \text{return (lazy (squares1 (ints 0)))))} \\
\text{setfun squares1(1)} &= \text{return (cons (times (car 1) (car 1)) (squares1 (cdr 1))))} \\
\text{setfun ints(x)} &= \text{return (cons x (ints (sum x 1))))}
\end{align*}
\]

This is the same as the eager version with the exception of the \textit{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 \texttt{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.

```lisp
(provide "prolog")
(require "unify")

(setfun lvs(i)
   (if (> i 0) (return (cons (new logvar) (lvs (diff i 1))))
    (return nil)))

(setfun (nest . 1) (nestl 1))

(setfun nestl(1)
   [when (cons-p 1) #*[s *((car 1)) (nestl (cdr 1))]])

(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.

```lisp
(provide "unify")
(require "logvar")

;-----------------------------
; Simple, non backtrackable unification.
(setfun s-unify(11 12)
   (return (simpl-unify 11 12)))

;-----------------------------
; Simple, non backtrackable left unification.
(setfun s-unifyl(11 12)
   (return (simpl-unifyl 11 12)))
```
(setfun simpl-unify(11 12)
  (when (logvar-p 11)
    (return (if ('put 11 12) 11 (simpl-unify ('get 11 12)))))
  (when (and (cons-p 11) (cons-p 12))
    (return (and-call "(simpl-unify (car 11) (car 12))
                       "(simpl-unify (cdr 11) (cdr 12))))
  (when (and (scons-p 11) (scons-p 12))
    (return (and-call "(simpl-unify (scar 11) (scar 12))
                       "(simpl-unify (scdr 11) (scdr 12))))
  (when (atom-p 11) (return (eql 11 12))))

(setfun simpl-unify(11 12)
  (when (and (cons-p 11) (cons-p 12))
    (return (and-call "(simpl-unify (car 11) (car 12))
                       "(simpl-unify (cdr 11) (cdr 12))))
  (when (and (scons-p 11) (scons-p 12))
    (return (and-call "(simpl-unify (scar 11) (scar 12))
                       "(simpl-unify (scdr 11) (scdr 12))))
  (if (logvar-p 11)
    (return (if ('put 11 12) 11 (simpl-unify ('get 11 12))))
  (if (logvar-p 12)
    (return (if ('put 12 11) 12 (simpl-unify ('get 12 11))))
  (when (atom-p 11) (return (eql 11 12))))

; Backtrackable unification.
(setfun unify(11 12)
  [when (and (cons-p 11) (cons-p 12))
    #[(return (cons *(unify (car 11) (car 12))  
                   *(unify (cdr 11) (cdr 12))))]
  [when (and (scons-p 11) (scons-p 12))
    #[(return (scons *(unify (scar 11) (scar 12))  
                  *(unify (scdr 11) (scdr 12))))]
  (if (logvar-p 11)
    [block #[(return *(unifyput 11 12))]
    (if (logvar-p 12)
      [block #[(return *(unifyput 12 11))]
      (when (atom-p 11) (when (eql 11 12) (return 11)))]))]

(setfun unifyput(11 12)
  [lets [(_ok %t)]
    #[(return (block1 *(put 11 12) (set _ok %t))))
    [unless _ok #[(return *(unify ('get 11 12))))]]]

(setfun copyloge(e)
  (return (let ((d ('new dictionary)))
            (copy-loged e d))))

(setfun copy-loged(e d)
  (when (cons-p e)
    (return (let [[(elef copy-loged (car e) d)]
                   (eright copy-loged (cdr e) d)]
      (if (and (eql (car e) elef) (eql (cdr e) eright))
        e
        (cons elef 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 (1)
      (unify p1 ()) (unify p2 1) (unify p3 1)   (p-cut)]
   [p-clause (1 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 powerful utilities 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 \texttt{al-read} with optional argument an input stream (default is the console). The function for printing an ALLOY expression is \texttt{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 \texttt{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 \texttt{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 \texttt{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 \texttt{oprep} which is also called by the main pre-processor.

\textbf{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).

\textbf{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 \texttt{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 \texttt{cl-c} and \texttt{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 \texttt{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 \texttt{avail} and is defined in subsection A.1.3. The various fields of it are:

- \texttt{init} is true if the variable has been set to some value (which could be another variable). Otherwise it is \texttt{nil}.
- \texttt{value} contains the value of the synchronizing variable if it has one.
- \texttt{suspend} contains pointers to processes which cannot proceed until this variable has been set.
- \texttt{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 \texttt{ext-function} and is defined in subsection A.1.3. It has the following fields:
8.2 The interpreter in Lisp

- 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.
- **pnu** 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.
8.2 The interpreter in Lisp

- **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.

- **ndyrmames** 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 \texttt{fgrec} is defined in subsection A.2.3 and has the following fields:

- \texttt{result} is a list of the single result of the fake generator.
- \texttt{expr} is the fake generator itself.
- \texttt{lazy} is the mode in which the generator must evaluate.
- \texttt{act} is the action that must be taken when a value is being requested by the generator.

It can be:

- \texttt{first} if a value has not been requested yet.
- \texttt{wait} if a value has been requested already.
- \texttt{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 $\mu$ closure. The data structure used to represent it is called \texttt{genstate} and is defined in subsection A.2.4. Its fields are:

- \texttt{result} is the list of results is generated up to that moment. The list may end in a synchronizing variable.

- \texttt{finished} is the queue of \texttt{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.

- \texttt{resend} is the tail of the \texttt{results} list. It is either a list or a synchronizing variable. That is where the next generated element will be placed.

- \texttt{next} is a tail of the \texttt{results} list. The car is the next element to be given.

- \texttt{rest} holds the list of values which must follow after the generated ones. Used for testing. Normally, it is always nil.

- \texttt{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 \texttt{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 \texttt{genstate} and is defined in subsection A.2.4 of chapter A. Its fields are:

- \texttt{results} is the list of all results of the expression with the multiple generators.

- \texttt{resend} is the tail of the \texttt{results} list. Useful to trigger generation of the next element.

- \texttt{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:

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>msecs</th>
<th>Cycles</th>
<th>Calls</th>
<th>Proc</th>
<th>Ents</th>
<th>Vars</th>
<th>PPS</th>
<th>EPS</th>
<th>PPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>app3</td>
<td>50</td>
<td>56</td>
<td>35</td>
<td>52</td>
<td>10</td>
<td>22</td>
<td>1040</td>
<td>200</td>
<td>5.2</td>
</tr>
<tr>
<td>nrev30</td>
<td>5500</td>
<td>6374</td>
<td>3953</td>
<td>5878</td>
<td>994</td>
<td>2421</td>
<td>1068</td>
<td>180</td>
<td>5.9</td>
</tr>
<tr>
<td>length100</td>
<td>2067</td>
<td>2314</td>
<td>1505</td>
<td>2112</td>
<td>406</td>
<td>808</td>
<td>1021</td>
<td>196</td>
<td>5.2</td>
</tr>
<tr>
<td>whut80</td>
<td>2439</td>
<td>2725</td>
<td>1797</td>
<td>2420</td>
<td>530</td>
<td>1026</td>
<td>924</td>
<td>217</td>
<td>4.5</td>
</tr>
<tr>
<td>qsort11</td>
<td>927</td>
<td>1247</td>
<td>805</td>
<td>1119</td>
<td>163</td>
<td>1463</td>
<td>828</td>
<td>175</td>
<td>6.8</td>
</tr>
<tr>
<td>ssort11</td>
<td>2433</td>
<td>2865</td>
<td>1834</td>
<td>2699</td>
<td>514</td>
<td>1277</td>
<td>1104</td>
<td>211</td>
<td>5.3</td>
</tr>
<tr>
<td>dine3-4</td>
<td>4274</td>
<td>4336</td>
<td>3134</td>
<td>3899</td>
<td>1290</td>
<td>1847</td>
<td>912</td>
<td>301</td>
<td>3.0</td>
</tr>
</tbody>
</table>

Lazy evaluation and generators:
### 8.3 Practical Considerations

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>msecs</th>
<th>Cycles</th>
<th>Calls</th>
<th>Procs</th>
<th>Envs</th>
<th>Vars</th>
<th>PPS</th>
<th>EPS</th>
<th>PPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>lprimes5</td>
<td>491</td>
<td>558</td>
<td>291</td>
<td>467</td>
<td>56</td>
<td>170</td>
<td>949</td>
<td>113</td>
<td>8.3</td>
</tr>
<tr>
<td>lprimes25</td>
<td>15426</td>
<td>15602</td>
<td>8601</td>
<td>12907</td>
<td>1366</td>
<td>4430</td>
<td>836</td>
<td>87</td>
<td>9.4</td>
</tr>
<tr>
<td>inters33</td>
<td>350</td>
<td>623</td>
<td>458</td>
<td>537</td>
<td>32</td>
<td>300</td>
<td>1534</td>
<td>91</td>
<td>16.8</td>
</tr>
<tr>
<td>perms3</td>
<td>1927</td>
<td>2859</td>
<td>2047</td>
<td>2388</td>
<td>127</td>
<td>1212</td>
<td>1256</td>
<td>66</td>
<td>18.8</td>
</tr>
<tr>
<td>perms4</td>
<td>14117</td>
<td>12724</td>
<td>9157</td>
<td>10518</td>
<td>582</td>
<td>5340</td>
<td>745</td>
<td>41</td>
<td>18.0</td>
</tr>
<tr>
<td>queens4</td>
<td>13150</td>
<td>10057</td>
<td>6868</td>
<td>9222</td>
<td>728</td>
<td>4221</td>
<td>701</td>
<td>55</td>
<td>12.6</td>
</tr>
<tr>
<td>queens4a</td>
<td>28034</td>
<td>24603</td>
<td>16813</td>
<td>22509</td>
<td>1782</td>
<td>10219</td>
<td>802</td>
<td>63</td>
<td>12.6</td>
</tr>
<tr>
<td>prapp2</td>
<td>9517</td>
<td>8489</td>
<td>5856</td>
<td>7890</td>
<td>1017</td>
<td>3738</td>
<td>829</td>
<td>106</td>
<td>7.7</td>
</tr>
<tr>
<td>prapp2or</td>
<td>12217</td>
<td>14780</td>
<td>10138</td>
<td>13837</td>
<td>2001</td>
<td>6430</td>
<td>1132</td>
<td>163</td>
<td>6.9</td>
</tr>
<tr>
<td>pinters33or</td>
<td>55183</td>
<td>44219</td>
<td>30015</td>
<td>41724</td>
<td>5490</td>
<td>19228</td>
<td>756</td>
<td>99</td>
<td>7.6</td>
</tr>
</tbody>
</table>

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 **mrev30** 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 Inefficient 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 Implementations on Parallel Architectures

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:

```lisp
1 ;-----------------------------------------------------------------------------------------------
2 ; ALLOY interpreter.
3 ; By Thanasis Mitsolides (mitsolid@cs.nyu.edu)
4 ; Written in Common Lisp
5 ; Tested in Austin Kyoto Common Lisp and Alegro Common Lisp.
6 ;-----------------------------------------------------------------------------------------------
7 ; Features:
8 ; - Serial/parallel execution.
9 ; - Eager/Lazy evaluation.
10 ; - Flexible commitment mechanism.
11 ; - Static scoping. Nested procedures.
12 ; - Higher order functions.
13 ; - Generators. Replicators.
14 ; - Object Oriented Programming.
15 ; - Parallel ADTs.
16 ; - Delegation.
17 ; - SIMPLICITY!
18 ;-----------------------------------------------------------------------------------------------
19 ; Speed
20 ; - On a Sparcstation 1 (12 VAX MIPS) using Austin-Kyoto Common Lisp
21 ; - An average of 700 function calls per second.
22 ; - A minimum of 500 fcps.
23 ; - A maximum of 1200 fcps.
```
The Interpreter of ALLOY in Common Lisp

;---------------------------------------------------------------------
; Tested in Austin Kyoto Common Lisp and Allegro Common Lisp.
;---------------------------------------------------------------------

; Main file. Loads everything else.
;---------------------------------------------------------------------

(proclaim '(optimize (speed 3) (safety 0) (space 0)))

(load "util.l")
(load "alutil.l")
(load "prep.l")
(load "gprep.l")
(load "oprep.l")
(load "load.l")
(load "values.l")
(load "io.l")
(load "inter.l")
(load "fcall.l")
(load "gcall.l")
(load "rcall.l")
(load "ocall.l")
(load "build.l")
(load "build2.l")
(load "obuild.l")

(compile '(load "alloylib.a"))

; the next command may need to be given again from the terminal.
(alloy)
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 execution 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 "--v--") (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*))
```
(unless (streamp stream) (al-error "Stream expected in: " stream))
(terpri stream))

(defun pure-list(e)
  (if (slist-p e) (slist-expr e) e))

;-------------------------------------------------------------
; Loader.

(defun al-load(fname out)
  (setq fname (sformat "A" fname))
  (let ((f (my-open-read-file fname)))
    (if (not f)
      (progn al-f (p-error "Cannot open file for reading: A" fname))
      (progn t (al-load-file f out)))))

(defun al-load-file(f out)
  (progn e)
  (force-output)
  (setq e (al-read f)) (when (eq e al-eof) (go exit))
  (setq e (al-prep e)) (when (eq e al-f) (go start))
  (when al-trace (princ "--L-> ") (al-print e) (terpri) (write e) (terpri))
  (feval (if out e (list '$a$no-result e)) al-trace)
  (go start)
  (exit)
  (close f))
A.1.3 inter.l

The main interpreter loop, fundamental structures and initialization:

```lisp
(defvar *alval* (make-alval))
(defvar *alproc* (make-proc))
(defvar *alinfo* (make-muinfo))
(defvar *alenv* (dumb-muinfo))
(defvar *alcall* (dumb-gmu))
(defvar *alcommit* (nil))
(defvar *alresult* nil)
(defvar *aledie* nil)
(defvar *alcommit* nil)
(defvar *alresult* (dumb-alval))
(defvar *alresult* (dumb-alval))
```

; Synchronizing variables are represented as:

```lisp
(defvar *alenv* (dumb-muinfo))
(defvar *alcall* (dumb-gmu))
(defvar *alcommit* (nil))
(defvar *alresult* nil)
(defvar *aledie* nil)
(defvar *alcommit* nil)
(defvar *alresult* (dumb-alval))
(defvar *alresult* (dumb-alval))
```

; The environment for a process is represented as:

```lisp
(defvar *alenv* (dumb-muinfo))
(defvar *alcall* (dumb-gmu))
(defvar *alcommit* (nil))
(defvar *alresult* nil)
(defvar *aledie* nil)
(defvar *alcommit* nil)
(defvar *alresult* (dumb-alval))
(defvar *alresult* (dumb-alval))
```
(lazy nil)
(top nil) ; At top level.
(object (muinfo-object p-lmu))) ; Copy object from last mu

; Declarating a process to be the child of a parent environment.
(defun make-child(ch mu)
  (setf (muinfo-pmu ch) mu)
  (unless (muinfo-top mu) (push ch (muinfo-chmus mu))))

; Holds global values. They can be changed by anyone.
(defstruct (gval)
  (val nil))

; fast structures.
(proclaim '(inline make-alval))
(proclaim '(inline make-proc))
(proclaim '(inline make-muinfo))

; Declarations. Top level variables.
(defvar al-cons nil "Used by fcall")
(defvar al-scons nil "Used by fcall")
(defvar proc-stack nil "The stack of processes.")
(defvar proc-queue nil "The queue of processes.")
(defvar proc nil "Current process.")
(defvar tab nil "Current hashing table. Current global environment.")
(defvar global-tab nil "Holds the global table during class definitions.")
(defvar globale-tab nil "Holds the global exported table.")
(defvar spec-table nil "Table of special expressions and their code.")
(defvar dumb-proc nil "Dumb process.")
(defvar dumb-alval nil "Dumb alloy variable.")
(defvar dumb-gmu nil "Dumb global environment.")
(defvar dumb-muinfo nil "Dumb local environment.")
(defvar global-class nil "Current global class.")
(defvar global-object nil "Current global object.")

(defvar gtq-max-arg 1000
  "Original value.")
(defvar gtq-max gtq-max-org
  "Move a process from the stack to the queue every steps.")
(defvar gtq-cycle gtq-max "Do the move when counter becomes negative.")

(proclaim '(type (or t nil) p-same))
(defvar p-same nil "Execute the same proc structure again.")

(proclaim '(type fixnum p-waitfor))
(proclaim '(type muinfo p-lmu))
(proclaim '(type (or t nil) p-serial))
(proclaim '(type alval  p-result))
(proclaim '(type cons  p-call))
(proclaim '(type proc  dumb-proc))
(proclaim '(type munfo dumb-muninfo))
(proclaim '(type alval  dumb-alval))
(proclaim '(type munfo dumb-gmu))
(proclaim '(type class  global-class))
(proclaim '(type object  global-object))

(defvar p-serial  nil  "Serial-parallel  part of current process")
(defvar p-call   nil  "Function Call  part of current process")
(defvar p-result nil  "Result variable  part of current process")
(defvar p-lmu    nil  "Local mu closure  part of current process")
(defvar p-waitfor nil  "Waiting for process  part of current process")
(defvar p-suspended nil  "Suspended process  part of current process")
(defvar p-lazy   nil  "Lazy not eager  part of current process")

; Main loop

(defun feval0($rest args)
  (init-structs)
  (apply 'al-eval args))

(defun feval($rest args)
  (reset-stats)
  (apply 'al-eval args))

(defun al-eval(start-expr &optional trace)
  (let ((al-eval3 start-expr
          (if class-define (nth 2 class-define) dumb-gmu)))
    (push proc proc-stack)
    (al-eval-loop trace)
    t))

(defun top-level-proc(start-expr)
  (al-eval4 start-expr dumb-gmu))

(defun al-eval3(start-expr gmu)
  (al-eval4 (make-alist :expr (list "$a$finish-eval start-expr)) gmu))

(defun al-eval4(expr gmu)
  (let* ((d-mu gmu)
          (d-proc (make-proc-m lazy nil :waitfor 100))
          (proc (make-proc-m :call expr
           :result (make-alval-m)
           :lmu d-mu
           :suspended d-proc
           :lazy nil
           :waitfor 0)))
    proc))

(defvar cycles-nosx 0)
(defvar proc-nosx 0)
(defvar alval-nosx 0)
(defvar muinfo-nosx 0)
(defvar funcall-nosx 0)
(defvar specall-nosx 0)

(defun al-eval-loop(trace)
  (prog ()
    (start
      (incf cycles-nosx)
      (when (minusp (decf gtq-cycle))
        (setq gtq-cycle gtq-max)
        (when proc-stack (queue-put (pop proc-stack) proc-queue)))
      (cond (p-same (setq p-same nil))
        (proc-stack (setq proc (pop proc-stack))
          (expand-proc))
        ((queue-nonempty proc-queue) (setq proc (queue-get proc-queue))
          (expand-proc))
        (t (go exit)))
      (when trace (pstatus))
      (when (muinfo-die p-lmu) (go start))
      (when (lazy-delay-p p-lazy)
        (setq p-lazy lazy1) (delay-me-expanded) (go start))
      (when (lazy-delay-p p-lazy)
        (setq p-lazy lazy1) (delay-me-expanded) (go start))
      (unless (spec-eval)
        (if p-serial (do-sc all) (do-pcall))
        (go start)
        (exit))
    (defun expand-proc()
      (let ((c (proc-call proc))
        (cond ((alist-p c) (setq p-serial t) (setq p-call (slist-expr c)))
          (t (setq p-serial nil) (setq p-call c)))
        (setq p-result (proc-result proc))
        (setq p-lmu (proc-lmu proc))
        (setq p-waitfor (proc-waitfor proc))
        (setq p-lazy (proc-lazy proc))
        (setq p-suspended (proc-suspended proc))
        (setq tab (class-tab (object-class (muinfo-object p-lmu)))))
    (defun construct-proc()
      (setq proc (let ((c (if p-serial (make-alist :expr p-call) p-call)))
        (make-proc-m :call c :result p-result :lm u p-lmu :waitfor p-waitfor
          :suspended p-suspended :lazy p-lazy)))))
    (defun construct-procc()
(check-p-serial)
(construct-proc)
(defun check-p-serial()
 (cond ((slit-p p-call) (setq p-serial t) (setq p-call (slit-expr p-call)))
 (t (setq p-serial nil)))
(defun p-ser-to-p-call()
 (when p-serial (setq p-serial nil) (setq p-call (make-slist-t p-call t)))))

; Initialization functions.
; Set up environment.
(defun init()
 (init-structs)
 (setq *print-circle* t)
 (setq *print-length* 14)
 (setq *print-level* 6)
 (setq spec-table (make-sym-table))
 (setq tab (setq global-tab (make-sym-table)))
 (setq globale-tab (make-sym-table))
 (setq dumb-alval (make-alval))

(setq global-class
 (make-class :origdef '(#(name . global) (static)(dynamic)(common)
 (inherit)(import)(methods))
 :tab global-tab :ndynames '(0 () ()))))
 (setq global-object (make-object :class global-class :createvar t))

(setq dumb-gmu
 (def-dumb-gmu))
 (setf (muinto-gmu dumb-gmu) dumb-gmu)
 (setf (muinto-gmu dumb-gmu) dumb-gmu)

(setq dumb-proc (make-proc-# :lazy nil :waitfor 9999999
 :result dumb-alval :suspended dumb-proc
 :call '(quote totop) ) :lmu dumb-gmu))

(setq p-lmu dumb-gmu)
 (setq dumb-muinto dumb-gmu))

(defun def-dumb-gmu()
 (make-muinto-# :commit t :suspended dumb-proc :top t
 :chmus nil :result dumb-alval :die nil :object global-object
 :lenv '((#(original-msg alloy-loop global-object)
 (here ,@ global-object) (self ,@global-object))))))

(defun init-structs()
 (reset-stats)
 (setq gtq-cycle gtq-max)
 (setq proc-stack nil)
 (setq proc-queue (make-queue))
 (setq proc nil))
(defun reset-stats()
  (setq cycles-nosx 0)
  (setq muinfo-nosx 0)
  (setq alval-nosx 0)
  (setq proc-nosx 0)
  (setq funcall-nosx 0)
  (setq specall-nosx 0)
)

; Tracing Functions.

(defun pstats()
  (pprint (list (cons "Cycles : " cycles-nosx)
                 (cons "Proc : " proc-nosx)
                 (cons "Muinfo : " muinfo-nosx)
                 (cons "Alval : " alval-nosx)
                 (cons "Funcall : " funcall-nosx)
                 (cons "Specallo: " (- specall-nosx funcall-nosx))
                 (cons "FScallt : " specall-nosx)))))

(defun pstatus()
  (print-reg p-call)

(defun print-reg (reg)
  '(let()
     (princ (quote ,reg)) (princ ";")
     (princ "=")
     (terpri))

(defun write ,reg :length 8 :level 4 :pretty t :circle t) (terpri))
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.

(defun do-call()
  (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-process(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) (list '$a$passi e)))

(defvar do-scall()
  (let ((g-fc (make-naddproc nil p-result p-lmu 1 p-suspended)))

(g-calls nil)
(setq p-call (nreverse (rm-dot-list p-call)))
(setq g-calls (do ((stk nil) expr (prev g-fc))
  ((null p-call) (setf (proc-waitfor prev) 0)
   (push prev proc-stack) stk)
  (setq expr (pop p-call))
  (push (valprocs) stk)))
(nput-dot-list-nc g-calls)
(setf (proc-call g-fc) (make-slist :expr (cons 'nfuncall-v5 g-calls))))

(deffunc valprocs()
  '(cond ((alcall-p expr) (let ((varres (make-alval-nil)))
      (setq prev (make-nadproc expr varres p-lmu lmu prev)
        varres))
      ((get-alval-nil expr)))))
(deffunc make-nadproc(expr result lmu waitfor suspended)
  ' (make-proc-m :call ,expr :result ,result ,lmu ,lmu
        :waitfor ,waitfor :suspended ,suspended) )

; Apply a function after its arguments have been computed.
(deffvar alfvs-f-nil "Function name of current external function.")
(deffvar alfvs-mu-nil "When t mu are not spawned but given to alfvs-mu.")
(defun al-funcall-vs()
  (setq alfvs-mu-nil)
  (al-funcall-vs-main))

;----
(defun al-funcall-nomu()
  (setq alfvs-mu-p-result)
  (setq p-result (cadr p-call))
  (when (slist-p (cddr p-call)) (setq p-serial t))
  (setq p-call (cons (car p-call) (rm-slist (cddr p-call))))
  (al-funcall-vs-main)
  (when (alval-p alfvs-mu) (alset-alval 'nomu alfvs-mu)))
(defun al-funcall-vs-main()
  (let* ((args (cdr p-call))
    (f (strip-alval (pop args)))
    (setq args (strip-alval args))
    (cond ((alval-p f) (waitfor-val f))
      ((eq f al-f) (arggone 'cl-failfunc nil))
      ((ext-function-p f)
        (incf funcall-noex)
        (setq alfvs-f-nil)
        (funcall (ext-function-type f) (ext-function-expr f) args)
        (%(and (eql f 'eval) (alval-p args)) (waitfor-val args)))
      (t (funcall f args)))
    (setq args)
    (setq p-call (cadr p-call)))
  (setq p-result (cadr p-call))
  (when (slist-p (cddr p-call)) (setq p-serial t))
  (setq p-call (cons (car p-call) (rm-slist (cddr p-call))))
  (al-funcall-vs-main)
  (when (alval-p alfvs-mu) (alset-alval 'nomu alfvs-mu)))
(defun al-funcall-nomu()
  (setq alfvs-mu-p-result)
  (setq p-result (cadr p-call))
  (when (slist-p (cddr p-call)) (setq p-serial t))
  (setq p-call (cons (car p-call) (rm-slist (cddr p-call))))
  (al-funcall-vs-main)
  (when (alval-p alfvs-mu) (alset-alval 'nomu alfvs-mu)))
(defun al-funcall-vs-main()
  (let* ((args (cdr p-call))
    (f (strip-alval (pop args)))
    (setq args (strip-alval args))
    (cond ((alval-p f) (waitfor-val f))
      ((eq f al-f) (arggone 'cl-failfunc nil))
      ((ext-function-p f)
        (incf funcall-noex)
        (setq alfvs-f-nil)
        (funcall (ext-function-type f) (ext-function-expr f) args)
        (%(and (eql f 'eval) (alval-p args)) (waitfor-val args)))
      (t (funcall f args)))
    (setq args)
    (setq p-call (cadr p-call)))
  (setq p-result (cadr p-call))
  (when (slist-p (cddr p-call)) (setq p-serial t))
  (setq p-call (cons (car p-call) (rm-slist (cddr p-call))))
  (al-funcall-vs-main)
  (when (alval-p alfvs-mu) (alset-alval 'nomu alfvs-mu)))
(defun al-funcall-nomu()
  (setq alfvs-mu-p-result)
  (setq p-result (cadr p-call))
  (when (slist-p (cddr p-call)) (setq p-serial t))
  (setq p-call (cons (car p-call) (rm-slist (cddr p-call))))
  (al-funcall-vs-main)
  (when (alval-p alfvs-mu) (alset-alval 'nomu alfvs-mu)))
(defun al-funcall-vs-main()
  (let* ((args (cdr p-call))
    (f (strip-alval (pop args)))
    (setq args (strip-alval args))
    (cond ((alval-p f) (waitfor-val f))
      ((eq f al-f) (arggone 'cl-failfunc nil))
      ((ext-function-p f)
        (incf funcall-noex)
        (setq alfvs-f-nil)
        (funcall (ext-function-type f) (ext-function-expr f) args)
        (%(and (eql f 'eval) (alval-p args)) (waitfor-val args)))
      (t (funcall f args)))
    (setq args)
    (setq p-call (cadr p-call)))
  (setq p-result (cadr p-call))
  (when (slist-p (cddr p-call)) (setq p-serial t))
  (setq p-call (cons (car p-call) (rm-slist (cddr p-call))))
  (al-funcall-vs-main)
  (when (alval-p alfvs-mu) (alset-alval 'nomu alfvs-mu)))
((and (or (eql f 'eval) (eql f 'new)) (cons-p args))
 (setq args (cons (setf (car args) (strip-alval (car args)))
 (cdr args))))
(alval-p (car args)))
(waitfor-val (car args)))
((and (eql f 'eval) (cons-p args) (ext-function-p (car args))))
(setq p-call (cons (car p-call) args))
(al-funcall-vs-main))
((and (eql f 'new) (cons-p args) (round-cons-p (car args))))
(setq p-call (cons (car p-call) (cons al-cons (cdr args))))
(al-funcall-vs-main))
((and (eql f 'new) (cons-p args) (square-cons-p (car args))))
(setq p-call (cons (car p-call) (cons al-scons (cdr args))))
(al-funcall-vs-main))
((symbol-p f) (message-other f args))
((class-p f)
 (setq p-call (cons (car p-call) (cons 'new (cdr p-call))))
 (al-funcall-vs-main))
(t)
(setq p-call (cons (car p-call) (cons 'eval (cdr p-call))))
(al-funcall-vs-main))
(t (when warning-exec
 (p-warning "Function or message expected in: "A" f))
 (argone 'cl-failfunc nil))))
(defun cl-failfunc(#rest dumb) (declare (ignore dumb))
 (cons 'suc al-f))
(defun pffunction(cl-fun)
 (funcall cl-fun))
(defun gfunction(args-body vals)
 (let ((current-gmu (select-lmu alfvs-f)))
 (lfunction-lmu args-body vals current-gmu t)))
(defun gbfunction(args-body vals)
 (let ((current-gmu (select-lmu alfvs-f)))
 (lfunction-lmu args-body vals current-gmu t)))
(defun select-lmu(f)
 (let ((lmu (ext-function-lmu f)))
 (if (eq (object-class (muinfo-object p-lmu))
 (object-class (muinfo-object lmu)))
 p-lmu lmu)))
(defun lfunction(args-body vals)
 (lfunction-lmu args-body vals (ext-function-lmu alfvs-f) nil))
(defun lfunction-lmu(args-body vals lmu globalp)
 (let ((body (cond ((cons args-body) (body-clist nil (cdr args-body)))
 ((slist-p args-body) (body-clist t (cdr (slist-expr args-body)))))
 ;-----
 ; Creates a process from the application of a closure.
 (defun lfunction-lmu(args-body vals lmu globalp)
 (let ((body (cond ((cons args-body) (body-clist nil (cdr args-body)))
 ((slist-p args-body) (body-clist t (cdr (slist-expr args-body)))))
 ;-----
 ; Creates a process from the application of a closure.
((al-error "Unexpected function definition: "#" args-body")))

(args (xcar args-body)))

(let* ((call-env (new-env args vals (muinfo-env lmu)))
   (mu (make-muinfo :gmu (muinfo-gmu lmu))
      :lenv (cdr call-env)
      :result p-result :lazy p-lazy
      :suspended p-suspended
      :object (muinfo-object lmu)))

(final (make-proc :call (list '>$a$alfun-fail)
      :lmu mu :waitfor 1
      :lazy nil))

(proc2 (make-proc :call body
      :result (make-alval-m)
      :lmu mu :suspended final :lazy nil
      :waitfor 0)))))

(make-child mu p-lmu)
(when globalp (setf (muinfo-gmu mu) mu))
(when (car call-env) (setq proc2 (lfl-carg (car call-env) proc2)))
(if alfvs-mu (progn (alset-alval proc2 alfvs-mu) (setq alfvs-mu nil))
  (push proc2 proc-stack))))

(defun lfl-carg(call proc2)

(let ((proga (make-proc :call call
      :result dumb-alval :lazy nil
      :waitfor 0 :suspended dumb-proc))
      (push proga proc-stack)
      (when p-serial
        (if alfvs-mu
          (let* ((vproc (make-alval))
          (prest (make-proc :call '($a$pass-v ,proc2)
             :result vproc
             :lazy nil
             :waitfor 1 :suspended dumb-proc))
          (setq (proc-suspended proga) prest) (setq proc2 vproc)
          (progn (setq (proc-suspended proga) proc2) (incf (proc-waitfor proc2) 1)
             (pop proc-stack) (setq proc2 proc-stack))))
      proc2))

(defun body-clist(serp body)

(if (and (cons-p body) (null (cdr body)) (alcall-p (car body)) (car body)
    (make-slist-t (cons 'list body) serp)))

;-----

; Build in function call which accepts all args in a list.

(defun argone(cl-func args)

(let ((v (funcall cl-func args))
  (cond ((eq (car v) 'suc) (give-value (cdr v)))
    ((eq (car v) 'sus) (waitfor-val (cdr v)))
  ((al-error "argone: Succeed or suspend cons expected: "#" v))))

;-----
; Build in function call which accepts all args separately.
(defun noninst (cl-func args)
  (let ((v (apply cl-func args)))
    (cond ((eq (car v) 'suc) (give-value (cdr v)))
          ((eq (car v) 'sus) (waitfor-val (cdr v)))
          ((al-error "noninst: Succeed or suspend cons expected: " v))))
)

; Operations on ALLOY values variables.

;-----
(defun get-sym-value (sym)
  (cond ((null sym) sym)
        ((symbolp sym)
         (strip-alval (prog2 (check-nvars)
                              (get-val sym (muinfo-lenv p-1mu) tab globale-tab)
                              (check-cvars)))))
  (sym)))

;-----
(defun waitfor-val (alval)
  (wake-up-delayed alval)
  (construct-proc)
  (push proc (alval-suspended alval)))

(defun wake-up-delayed (alval)
  (when (alval-delayed alval)
    (wakesps (alval-delayed alval)) (setf (alval-delayed alval) nil)))

;-----
(defun give-value (val)
  (alset-alval val p-result)
  (check-suspended))

;-----
(defun alset-alval (val alval) ; value variable
  (unless (or (not (alval-p alval)) (alval-init alval))
    (setq val (strip-alval val))
    (setf (alval-init alval) t)
    (setf (alval-value alval) val)
    (let ((susps1 (alval-suspended alval)))
      (setf (alval-suspended alval) nil)
      (if (alval-p val)
(when susp1 (push susp1 (alval-suspended val)) (wake-up-delayed val))
(wakeups susp1))

(defun wakesps(l)
  (cond ((null l))
    ((consp l) (wakesps (car l)) (wakesps (cdr l)))
    ((push l proc-stack)))

;-----
; Inform waiting process (continuation) that current process has finished.
; If waiting process is ready to execute make it ready.

(defun check-suspended()
  (check-suspended1 p-suspended))

(defun check-suspended1 (proc2)
  (when (zerop (decf (proc-waitfor proc2)))
    (push proc2 proc-stack)))

;---------------------------------------------------------------
; Evaluation of Special Expressions.
; Treated as macros.

(defun spec-eval()
  (let ((fun-name (get-spec-value (car p-call))))
    (when fun-name
      (incf specall-nosx)
      (funcall (car fun-name))
      t)))

(defun spec-quote()
  (let ((qarg (cdr p-call)))
    (cond ((consp qarg) (give-value (car qarg)))
      (null qarg) (give-value qarg))
    (t (p-error "Quote has dotted argument."))))

(defun spec-$aspass-v()
  (let ((arg (cadr p-call)))
    (give-value arg))

(defun spec-lazy()
  (setq p-lazy lazyd)
  (setq p-call (cadr p-call))
  (check-proc-call)
  (next-the-same))

(defun spec-delay()
  (setq p-call (cadr p-call))
  (check-proc-call)
  (delay-me))

(defun spec-eager()
  (setq p-call (cadr p-call))
  (setq p-lazy nil)
  (check-proc-call)
(next-the-same))

(defun delay-me()
  (if (alval-suspended p-result)
      (next-the-same)
      (push (construct-procc) (alval-delayed p-result)))))

(defun delay-me-expanded()
  (p-ser-to-p-call)
  (delay-me))

(defun spec-mu()
  (let ((mu-call (make-slist-t (cdr p-call) p-serial)))
    (give-value (if (muinfo-top p-lmu) ; Global level.
       (make-gfunction mu-call)
       (make-lfunction mu-call))))

(defun spec-function()
  (setq p-call (list () (cons 'return (cdr p-call))))
  (spec-mu)

(defun spec-%a$%set-vi()
  (let ((var-val (cdr p-call)))
    (setq p-call (caddr p-call))
    (unless (alset-alval p-result var-val)
      (alset-alval al-f p-result))
    (check-proc-call)
    (next-the-same))

(defun spec-set()
  (spec-set-forse nil))

(defun spec-when()
  (let ((condition (cadr p-call)))
    (cond ((alcall-p condition)
      (spec-xcond-wait `$a$%such-when-v condition))
      ((let ((c (get-alsym-value condition)))
        (setq p-call (cons `$a$%such-when-v (cons c (caddr p-call))))
        (spec-`$a$%such-when-v))))

(defun spec-xcond-wait(code condition)
  (if p-serial
    (let ((alval (make-alval-m)))
      (setq p-call (cons code (cons alval (caddr p-call))))
      (incf p-waitfor)
      (construct-proc)
      (make-addproc condition alval p-lmu 0 proc))
    (let ((alval (proc-result (make-pprocess condition))))
      (setq p-call (cons code (cons alval (caddr p-call))))
      (waitfor-val alval))))

(defun spec-if-wait(alval)
  (setq p-call (cons `$a$%if-v (cons alval (caddr p-call))))
  (waitfor-val alval))


(defun spec-$a$when-v()
  (let* ((args (strip-alval (cdr p-call)))
         (c (strip-alval (pop args))))
    (cond ((alval-p c) (waitfor-val c))
          ((eq c al-f) (argone 'cl-failfunc nil))
          ((null args) (give-value c))
          (t
           (setq p-call
                 (make-slist-t
                  (cons 'blockn (if (rlist-p args)
                                   (cons '($a$pass-v ,c) args)
                                   args))
                  (p-serial))
                  (next-the-same))))))
  (defun spec-if()
    (let ((condition (cadr p-call)))
      (cond ((alcall-p condition)
             (spec-xcond-wait 'spec-if-v condition))
            ((let ((c (get-alsym-value condition))
                   (spec-p-call (cons 'spec-if-v (cons c (cadr p-call))))
                   (spec-$a$if-v))))))
    (defun spec-$a$if-v()
      (let* ((args (cdr p-call))
             (c (strip-alval (pop args))))
        (cond ((alval-p c) (waitfor-val c))
              ((and (eq c al-f) (null (cdr args)))
               (give-value c))
              ((eq c al-f)
               (setq p-call (cadr args))
               (check-proc-call)
               (next-the-same))
              (t
               (setq p-call (car args))
               (check-proc-call)
               (next-the-same))))))
  ;------------------------------------------------------------------
  ; Commitment mechanisms. Return and fail. Local and global.
  (defun spec-return()
    (spec-lreturn-lmu (muinfo-gmu p-lmu) t))
  (defun spec-lreturn()
    (spec-lreturn-lmu p-lmu nil))
  (defun spec-lreturn-lmu(lmu globalp)
    (let ((r (muinfo-result lmu)))
      (spec-lr-lmus (if (muinfo-p r) r lmu) lmu globalp)))
  (defun spec-lr-lmus(muinfo-p-lmu globalp)
    (cond ((muinfo-commit muinfo-p-lmu)
           (if globalp
(p-error "return statement executed after commit."
(p-error "lreturn statement executed after commit."
(give-value al-f))
((check-ser-return))
((let ((expr (cadr p-call)))
  (gen-or-commit comgen-lmu))
  (cond ((alcall-p expr)
    (let ((com-mu (make-muinfo-g :mu (muinfo-gmu lmu)
      (muinfo-env p-lmu)
      :commit t
      :result dumb-alval
      :suspended dumb-proc))
        (make-child com-mu (muinfo-gmu lmu))
      (when globalp (setf (muinfo-gmu com-mu) com-mu))
      (setq p-lmu com-mu)
      (setq p-call expr)
      (setq p-lazy (muinfo-lazy lmu))
      (next-the-same)))
    (t (give-value (get-alsym-value expr))))))))
(defun gen-or-commit(lmu)
  (if (genstate-p (muinfo-result lmu))
    (gen-lreturn-lmu lmu)
    (progn (terminate-fun lmu)
      (alset-alval p-result (muinfo-result lmu)))))
(defun terminate-fun(lmu)
  (setq p-suspended (muinfo-suspended lmu))
  (kill-mus lmu))
(defun check-ser-return()
  'when p-serial
    (let* ((v (make-alval))
      (prest (make-proc-m :call (list (if globalp 'return 'lreturn) v)
        :result p-result :lazy nil
        :lmu p-lmu :waitfor 1 :suspended p-suspended))
      (setq p-suspended prest)
      (setq p-result v)
      (setq p-call (cdr p-call))
      (check-proc-call)
      (next-the-same))
    t))
(defun alcall-p(e)
  (or (comp e) (alist-p e)))
(defun kill-mus(mu)
  (declare (type muinfo mu))
  (setq (muinfo-commit mu) t)
  (setq (muinfo-die mu) t)
  (let ((chmus (muinfo-chmus mu)))
    (while chmus (kill-mus (pop chmus)))
    (setq (muinfo-chmus mu) nil)))
(defun spec-$a$-result()
  (alset-alval '[$a$no-result p-result]
  (setq p-result (make-alval))
  (setq p-call (cdr p-call))
  (make-eager-self)
  (check-proc-call)
  (next-the-same))

(defun spec-alfun-fail()
  (alset-alval al-f (muinfo-result p-lmu))
  (check-suspended (muinfo-suspended p-lmu)))

(defun spec-fail()
  (spec-fail-lmu (muinfo-gmu p-lmu) t))

(defun spec-ifail()
  (spec-fail-lmu p-lmu nil))

(defun spec-fail-lmu (lmu globalp)
  (let ((r (muinfo-result lmu)))
    (spec-f-lmus (if (muinfo-p r) r lmu) lmu globalp)))

(defun spec-f-lmus (comgen-lmu lmu globalp) (declare (ignore lmu))
  (cond ((muinfo-commit comgen-lmu)
    (if globalp
      (p-error "fail statement executed after commitment.")
      (p-error "lfail statement executed after commitment.")
    (t
      (terminate-fun comgen-lmu)
      (setp p-lmu comgen-lmu)
      (if (genstate-p (muinfo-result comgen-lmu))
        (spec-alfun-gfail)
        (spec-alfun-fail)))
    (give-value al-f))
)

(defun make-eager-self()
  (setq p-lazy nil))

(defun make-eager-self()
  (setq p-lazy nil))

(defun next-the-same()
  (check-p-serial)
  (setq p-same t))

(defun check-proc-call()
  (unless (alcall-p p-call)
    (setq p-call (list '[$a$pass1 p-call))))

(defun next-the-same()
A.2.2 ocall.lib

Run time execution of objects and classes:

```lisp
(defun spec-$a$setfun1()
  (let (((v (get-alval-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)
                   '(function 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))))))
```
(alset-alval al-f p-result)
(set-sym-value tab var-name p-result)
(alset-alval al-f p-result)))

(check-cvars)
(check-proc-call)
(next-the-same))

; Class definition.

(defstruct (class)
  (origdef nil) ; name and assoc list of definition in start
  (tab nil) ; Symbol table of set/common inherited vars
  (ndynnames nil) ; 1 #. 2 name. 3 next-names.
)

(defun class-name(class)
  (get-feature 'name (class-origdef class)))

(defun object-name(object)
  (get-feature 'name (class-origdef (object-class object))))

(defstruct (object)
  (class nil) ; The class
  (interface nil) ; List of current methods.
  (inenv nil) ; Multi alist list of static vars
  (nextdyn (make-gval :val nil)) ; List of next dynamic variables.
  (createvar nil) ; Can create static variables.
)

(defun spec-$a$class-start()
  (let* ((defs (cdr p-call))
     (cname (car defs))
     (c (make-class)))
    (setq defs (cadr defs))
    (if (not (check-set-cdef cname)) (give-value al-f)
      (let()
       (let ((obj-gmu (make-muinfo-m :top t :commit t)))
         (setq muinfo-gmu obj-gmu)
         (make-child obj-gmu p-lmu)
         (setq muinfo-object obj-gmu) (make-object :class c :createvar t))
         (setq class-define (list cname c obj-gmu)))
         (setq class-origdef c) (cons (cons 'name cname) defs))
         (setq class-tab c) (make-sym-table))
        (let ((dyns (get-feature 'dynamic defs)))
          (setq class-ndynnames c)
          (list (length dyns) dyns (ren-list dyns 'next-)))
          (setq-sym-value tab cname c)
          (setq-sym-value tab (atom-app cname 'p)
            (make-gbfunction '((e) (return (eql 'object-p e)
              (quote ,cname)))))))))
    (setq tab (class-tab c))
    (set-sym-value tab 'new
      (make-gbfunction '() (return)))
    (give-value c))))
(defun check-set-cdef (cname)
  (let ((v (get-alarm-value cname))) ; Striped
    (cond (class-define
      (p-error "Nested class definition ignored: "A" cname) nil)
      (alval-p v)
      (when warning-load (p-msg "Defining class : "A" cname)) t)
      (class-p v)
      (when warning-load (p-msg "Redefining class: "A" cname)) t)
      (t
      (p-error "Can not define class. Symbol is not a user class: "A" cname)
    nil))))

(defun ren-list (names prefix)
  (cond ((null names) nil)
    ((cons (atom-app prefix (car names)) (ren-list (cdr names) prefix)))))

(defun create-objs (mytab names prefix)
  (cond ((null names) (t)
    (let* ((name (car names))
      (args nil)
      (oname (atom-app prefix name)) (v (make-alval))
      (set-sys-value mytab oname v)
      (make-pprocessv '(new ,(get-sys-value global-tab name) ,@args) v)
      (create-objs mytab (cdr names) prefix))))))

(defun spawn-obj-list (names prefix)
  (when names
    (let* ((name (car names))
      (args nil)
      (oname (atom-app prefix name)) (v (make-alval))
      (make-pprocessv '(new ,(get-sys-value global-tab name) ,@args) v)
      (cons (cons oname v) (spawn-obj-list (cdr names) prefix)))))

(defun spec-$class-end()
  (let ((cname (cadr p-call)))
    (if (not (eq (car class-define) cname))
      (if (car class-define)
        (p-msg "ERROR: end of class definition expected: "A" (car class-define))
        (p-msg "ERROR: unexpected class conclusion: "A" cname))
      (let* ((c (cadr class-define)) (defs (class-origdef c))
        (check-methods (get-feature 'methods defs) tab)
        (when warning-load (p-msg "Defined class: "A" cname))
        (setq class-define nil)
        (create-objs tab (get-feature 'common defs) 'c-)))))
    (give-value cname)))))

(defun get-feature (name alist)
  (cdr (assoc name alist)))

(defun check-methods (mlist tab)
  (cond ((null mlist) nil)
    (t
      (let* ((m (car mlist)) (def (get-sys-value tab m)))
        (unless (ext-function-p (strip-alval def)))))

A.2 Main Interpreter
(p-warning "Undefined method: "A" m)))
(check-methods (cdr mlist) tab)))
(defun link-vars (names to td)
  (cond ((null names))
    (t
     (let ((name (car names)))
      (set-sym-value td name (get-sym-value to name)))
      (link-vars (cdr names) to td))))

; Actual processing of messages (sent to class or object).
(defun message-other (f args)
  (setq args (strip-alval args))
  (let ((obj nil))
    (cond ((null args)
      (p-error "incorrect message passing in: "A" (cons f nil))
      (argone 'cl-failfunc nil))
      ((alval-p args) (waitfor-val args))
      ((not (listp args))
      (p-error "incorrect message passing in: "A" (cons f args))
      (argone 'cl-failfunc nil))
      ((alval-p (setq obj (strip-alval (pop args)))) (waitfor-val obj))
      ((message-other (f obj args obj)))

(funcall message-other (msg obj args orig-obj))
  (cond ((eq msg 'delegate) (delegate-message obj args))
     ((eq msg 'resend) (resend-message obj args))
     ((object-p obj) (send-message msg obj args orig-obj))
     ((class-p obj) (send-m-class msg obj args orig-obj))
     (t (send-bi-obj msg obj args orig-obj)))

(defun send-m-class (msg c args orig-obj)
  (case msg
    (class-p (give-value (class-name c)))
    (object-p (give-value al-f))
    (class (give-value al-f))
    (name (give-value (class-name c)))
    (new (create-instance c args orig-obj))
    (otherwise
     (when warning-exec
      (p-warning "Message "A unknown to class "A" msg (class-name c))
      (argone 'cl-failfunc nil)))))

(defun send-message (msg obj args orig-obj)
  (case msg
    (object-p (give-value (object-name obj)))
    (class-p (give-value al-f))
    (class (give-value (object-class obj)))
    (interface (get-interface obj))
    (otherwise
     (cond ((member msg (object-interface obj)))
        (send-message-x msg obj args orig-obj)))))
((-member msg (current-interface-names obj)) (get-interface obj (list msg)))

(t (when warning-exec
  (p-warning "Message `A unknown to object of class `A"
    msg (object-name obj))
  (p-warningc "Current interface: `A" (object-interface obj)))
  (argone `cl-failfunc nil))))

(defun send-message-x(msg obj args obj-self)
  (if (bobjext-p obj)
      (send-msg-b msg obj args obj-self)
      (send-message-x2 msg obj args obj-self))

(defun send-message-x2(msg obj args obj-self)
  (let* ((tabo (class-tab (object-class obj)))
         (ndn (class-nadynames (object-class obj)))
         (curdny (gval-val (object-nextdyn obj)))
         (nextdny (make-alvals-l (caddr ndn)))
         (gmu (make-muinfor :object obj
            :lenv '(((original-msg ,msg ,obj-self ,@ args)
                   (here ,@ obj) (self ,@ obj-self))
                 ,(pairis (cadr ndn) curdny)
                 ,(pairis (caddr ndn) nextdny)
                 ,@ (object-inienv obj))
            :commit t)))
    (make-child gmu p-lmu)
    (setf (gval-val (object-nextdyn obj)) nextdny)
    (setf (muinfor-gmu gmu) gmu)
    (setq p-lmu gmu)
    (setq p-call `($a$funcall-nomu ,p-result ,(get-sym-value tabo msg) ,@args))
    (setq p-result alvse-mu)
    (setq alvse-mu nil)
    (setq tab tabo)
    (next-the-same))

(defun delegate-message(obj args) (declare (ignore args))
  (let ((last-msg (get-val 'original-msg (muinfor-lenv p-lmu) tab)))
    (message-other4 (car last-msg) obj (caddr last-msg) (cadr last-msg)):)))

(defun resend-message(obj args) (declare (ignore args))
  (let ((last-msg (get-val 'original-msg (muinfor-lenv p-lmu) tab)))
    (message-other4 (car last-msg) obj (caddr last-msg) obj)))

(defun get-interface(obj args)
  (cond ((null args)
         (p-error "Name expected for interface.")
         (argone `cl-failfunc nil))
         ((alval-p args) (waitfor-val args))
         ((not (listp args))
          (p-error "Name expected for interface.")
          (argone `cl-failfunc nil))
         ((alval-p (setq args (strip-alval (car args)))) (waitfor-val args))
         ((let ((int (get-interface-list obj args)))
            (cond ((null int)
                   (p-error "Unknown interface: `A of a: `A object")
The Interpreter of ALLOY in Common Lisp

(defun get-interface-list(obj name)
  (let ((is (get-feature 'interfaces (class-origdef (object-class obj)))))
    (assoc name is)))

(defun current-interface-names(obj)
  (let ((is (get-feature 'interfaces (class-origdef (object-class obj)))))
    (mapcar 'car is)))

;-----
; Process message new send to class.

(defun create-instance2(cls args obj-self)
  (let* ((def (class-origdef cls))
         (tabo (class-tab cls))
         (ndn (class-ndynames cls))
         (curdyn (make-alvals-1 (cadr ndn)))
         (nextdyn (make-alvals-1 (caddr ndn)))
         (snames (get-feature 'static def))
         (static (make-alvals-1 snames))
         (obj (make-object
                :class cls
                :interface (get-feature 'methods def)
                :inienvv (list (pairlis snames static)
                               (spawn-obj-list (get-feature 'part def) 'p-))
                :nextdyn (make-gval :val nextdyn)))
         (gmu (make-munifo-m :object obj
            :lenv '(((original-msg new ,obj-self ,0 args)
                     (here ,0 obj) (self ,0 obj-self))
                     ,(pairlis (cadr ndn) curdyn)
                     ,(pairlis (caddr ndn) nextdyn)
                     ,(0 (object-inienv obj)))
                    :commit t)))))
  (setf (munifo-gmu gmu) gmu)
  (setq p-lmu gmu)
  (setq tab tabo)
  (link-vars (get-feature 'import def) global-tab tab)
(setq p-lazy nil) ; New function is eager.
(let ((prest (make-proc-m :call '$(a$pass-v ,obj) :result p-result
           :lazy nil
           :lmu p-lmu :waitfor 1 :suspended p-suspended)))
  (setq p-suspended prest))
(setq p-result dumb-alval)
(setq p-call '$(a$funcall-vs ,($(get-sym-value tabo 'new) ,@args))
  (next-the-same)))
(defun make-alvals-l(lo)
  (let ((nil))
    (while lo (push (if (sassg-p (car lo)) (make-alval nil) 1) (pop lo))
      (nreverse 1)))
(defun make-assoc-t(names mytab)
  (let ((nil))
    (while names (push (cons (car names) (get-sym-value mytab (car names))) 1)
      (pop names))
    1))
A.2.3 rcall.1

Handling of replicators and nested generators:

(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)(gcq-popcall) (next-the-same))
      (lazy (setq p-lazy lazyd) (gcq-popcall) (next-the-same))
      (eager (setq p-lazy nil) (gcq-popcall) (next-the-same))
      (otherwise
        (cond ((get-spec-value fname)
          (gire-value (make-fgen-call (cdr p-call)
            (caddr p-call) (cdddr p-call))))
        ((setq p-call
          (make-slist-t (cons '$a$gen-call (cdr p-call))
            (cadr p-call))
          (next-the-same))))))

(defun gcq-popcall()
  (let* ((call (caddr 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-$a$gen-call(fnargs)
  (let ((fname (car fnargs)) (args (cdr fnargs)))
    (cond ((alval-p (setq fname (strip-alval fname))
        (cons 'alval fname)))
      ((cons 'alval (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))

The Interpreter of ALLOY in Common Lisp
(setf (fgrec-act s) 'first)
(set s))

(defun cl-$a$fgen-next(fgrec)
  (cond ((alval-p (setq fgrec (strip-alval fgrec))) (cons 'suc fgrec))
    ((not (fgrec-p fgrec))
      (al-error "$a$fgen-next: fgrec expected in: "f" fgrec))
    ((fgrec-act fgrec)
      (case (fgrec-act fgrec)
        (first (fgen-next-act fgrec))
        (wait (cons 'suc (fgrec-result fgrec)))
        (otherwise
          (if (genstate-p (fgrec-act fgrec))
            (cl-$a$gen-next (fgrec-act fgrec))
            (al-error "Internal. Unknown fake generator: "f" fgrec))))
      ((fgen-next fgrec)))))

(defun fgen-next(fgrec)
  (let ((result (strip-alval (fgrec-result fgrec))))
    (cond ((not (alval-p result)) (cons 'suc al-f))
      ((let* ((gval (make-gval))
        (ps (make-proc-m :call '($a$csuspended-g1 , gval)
          :result dumb-alval :lazy nil
          :lmu p-lmu :waitfor 1
          :suspended p-suspended))
        (px (make-pprocess (fgrec-expr fgrec))))))
      (setf (proc-lazy px) (fgrec-lazy fgrec))
      (setf (proc-suspended px) ps)
      (alset-alval (cons (cons gval (proc-result px)) nil) result)
      (cons 'suc (car (strip-alval result))))))

(defun fgen-next-act(fgrec)
  (setf (fgrec-act fgrec) 'wait)
  (let* ((gval (make-gval))
    (res (make-alval))
    (ps (make-proc-m :call '($a$csuspended-g1 , gval)
      :result dumb-alval :lazy nil
      :lmu p-lmu :waitfor 1
      :suspended p-suspended))
    (px (make-pprocess '($a$funcall-nomu , res , 0(fgrec-expr fgrec))))
    (pf (make-pprocess '($a$fcmu-final , (proc-result px) , gval , res , fgrec))))
  (make-eager pf)
  (setf (proc-suspended px) ps)
  (cons 'suc (proc-result pf))))

(defun cl-$a$fcmu-final(typ gval res fgrec)
  (cond ((alval-p (setq typ (strip-alval typ))) (cons 'sus typ))
    ((alval-p (setq fgrec (strip-alval fgrec))) (cons 'sus fgrec))
    ((alval-p (setq gval (strip-alval gval))) (cons 'sus gval))
    ((proc-p typ)
      (when (zerop (defc (proc-waitfor p-suspended)))
        (al-error "$a$fcmu-final: Internal."))
      (let ((g (cdr (apply-gproc typ nil))))
        (setf (fgrec-act fgrec) g))))
The Interpreter of ALLOY in Common Lisp

(alset-alval (genstate-result g) (fgrec-result fgrec))
(cons 'suc (proc-result (make-pprocess '($a$gen-next ,g)))))
(t
(alset-alval (list (cons gval res)) (fgrec-result fgrec))
(setf (fgrec-act fgrec) nil) (cons 'suc (cons gval res)))))

;-----------------------------------------------
; FAST linear-time generators when #! is in use.

(defun cl-$a$directg1 f &rest args)
  (let* ((res (make-alval))
      (px (make-pprocess (make-slist-t '($a$funcall-nomu ,res ,f ,@args)
p-serial)))
      (pf (make-pprocess '($a$dirg-final ,proc-result px) ,gl ,res)))
       (make-eager px) (make-eager pf)
    (cons 'suc (proc-result pf)))))

defun cl-$a$dirg-final(typ gl res)
  (cond ((alval-p (setq typ (strip-alval typ))) (cons 'sus typ))
      ((alval-p (setq gl (strip-alval gl))) (cons 'sus gl))
      ((proc-p typ)
       (let ((lmu (if gl (muinfo-gmu p-lmu) p-lmu)))
         (setf (muinfo-result (proc-lmu typ))
               (if (muinfo-p (muinfo-result lmu)) (muinfo-result lmu) lmu)))
       (push typ proc-stack)
       (cons 'suc 88))
       (t
       (gen-or-commit (if gl (muinfo-gmu p-lmu) p-lmu))
       (cons 'suc res)))))

;-----------------------------------------------
; Run-time replicator.

defun cl-$a$rep-driver(serp clex ggen rest)
  (cond ((alval-p (setq serp (strip-alval serp))) (cons 'sus serp))
      ((alval-p (setq clex (strip-alval clex))) (cons 'sus clex))
      ((alval-p (setq ggen (strip-alval ggen))) (cons 'sus ggen))
      ((alval-p (setq rest (strip-alval rest))) (cons 'sus rest))
      (t
      (make-eager (make-pprocess '($a$ggen-next ,ggen)))
      (let ((pl (make-pprocess
                   (list '$a$rd-loop (ggen-results ggen)
                    ggen serp clex
                    (if serp rest
                     (proc-result (make-pprocess rest)))))))
       (cons 'suc (proc-result pl)))))

(defun make-eager(lproc)
  (setf (proc-lazy lproc) nil))
(defun reset-lazy-self()
  (when (eq lazyt p-lazy) (setq p-lazy lazyd))
(defun spec-$a$rd-loop()
  (let* ((args (cdr p-call))
        (list '$a$rd-loop (ggen-results ggen)
             ggen serp clex
             (if serp rest
              (proc-result (make-pprocess rest)))))))
A.2 Main Interpreter

(defun rd-loop-cont (results ggen serp clex rest)
  (cond ((null results)
            (give-value (if serp (proc-result (make-pprocedure rest)) rest))
        ((let ((newend (make-alval-m))
               (res (make-alval-m)))
            (alset-alval (cons res newend) p-result)
            (reset-lazy-self)
            (setq p-result newend)
            (setq p-call (cons (car p-call) (cons (cdr results) (caddr p-call))))
            (if serp
             (let ((pexec nil))
              (incf p-waitfor)
              (construct-procc)
              (setq pexec (make-proc-m :call '($a$rd-exec ,(car results)
                        .ggen .serp .clex .p-lazy)
                        :result res .lmu p-lmu :waitfor 0
                        :suspended proc))
              (make-eager pexec)
              (push pexec proc-stack))
             (let ()
              (make-eager (make-pprocedure '($a$rd-exec ,(car results)
                        .ggen .serp .clex .p-lazy)
                        res))
             (next-the-same))))))

(defun spec-$a$rd-exec()
  (let* ((args (cdr p-call))
          (gval-res (pop args)) (ggen (pop args))
          (serp (pop args)) (clex (pop args)) (lazy (pop args)))
    (cond ((not serp)
             (make-eager (make-pprocedure '($a$ggen-next .ggen))
             (setq p-call (funcall clex (cdr gval-res)))
             (setq p-lazy lazy) (reset-lazy-self)
             (check-proc-call)
             (next-the-same))
        ((if-gval-susp-proc (car gval-res) proc)
         (incf (proc-waitfor proc))
        ((let ((pn (make-proc-m :call '($a$ggen-next .ggen)
                            :result dumb-proc :lazy nil
                            :lmu p-lmu :waitfor 1
                            :suspended p-suspended)))
          (setq p-suspended pn)
          (setq p-call (funcall clex (cdr gval-res)))
          (setq p-lazy lazy) (reset-lazy-self)
          (check-proc-call)
          (next-the-same)))))))

(defstruct (mngrec)
(results nil)
(resend nil)
(serp nil)
(ggen nil)
(clex nil)

; Takes as parameters a general generator ggen and a cl expr producing a
general generator ggeni.
; Produces all possible ggeni, calls ggenis and returns all their values.
(defn cl-$a$gen-driver(serp clex ggen)
  (cond ((alval-p (setq serp (strip-alval serp))) (cons 'sus serp))
        ((alval-p (setq clex (strip-alval clex))) (cons 'sus clex))
        ((alval-p (setq ggen (strip-alval ggen))) (cons 'sus ggen))

(t
  (let* (((v (make-alval-m))
      (mngrec (make-mngrec :results v :resend v
          :ggen ggen :clex clex :serp serp))
      (pl (make-proc-m
           :call (list '$a$gd-loop (ggen-results ggen)
                  mngrec nil)
           :result v :lmu p-lmu :waitfor 0
                  :suspended dumb-proc)))))
    (delay-proc pl) (push pl proc-stack)
  (cons 'suc mngrec))))))

(defun spec-$a$gd-loop() ; global-gen-results mngrec rest
  (let ((mngrec (caddr p-call)))
    (make-eager (make-proc-process ('$a$gen-next , (mngrec--gen mngrec))))))
  (cons p-call (cons '$a$gd-loop-cont (cdr p-call)))
  (next-the-same)))))

(defun spec-$a$gd-loop-cont() ; global-gen-results mngrec rest
  (let* (((args (cdr p-call))
      (results (strip-alval (pop args)))
      (mngrec (pop args)) (rest (pop args)))
      (cond ((alval-p results) (waitfor-val results))
      (null-p results) (give-value rest))
      ((let ((lgen (make-alval-m)) (fend (make-alval-m)) cont g2)
        (setq cont (make-procprocess
                    ('$a$gd-loop ,(cdr results) ,mngrec ,rest) fend))
      (delay-proc cont))
    (setq g2 (funcall (mngrec-clex mngrec) (cdar results)))
    (make-eager
      (make-procprocess
       (make-alist-t
        (cons '$a$gen-call-gq (cons (alist-p g2) (rm-alist g2)))
        (alist-p g2) lgen))
    (setq p-call ('$a$genlist ,lgen ,(caar results) ,fend))
    (cond ((mngrec-serp mngrec)
      (construct-proc)
      (if (if-val-sus-proc (caar results) proc)
        (incf (proc-waitfor proc))
      (next-the-same))))
(defun spec-$a$gennlist() ; coex gval rest
  (let ((coex (strip-alval (cdr p-call))))
    (cond ((alval-p coex) (waitfor-val coex))
      (t (setq p-call (list 'a$gennmake (ggen-results coex)
                          coex (cddr p-call))))
      (delay-me))))

(defun spec-$a$gennmake() ; local-results coex gval-res
  (let ((coex (cadr p-call)) pnext)
    (setq pnext (make-pprocess (list 'a$genn-next coex))
      (make-eager pnext)
    (setq p-call (cons 'a$gennmake (cdr p-call)))
    (delay-me)))

(defun spec-$a$gennmake() ; local-results coex gval-res
  (let ((next (strip-alval (cadr p-call)))
         (gval-rest (car (last p-call))))
    (cond ((alval-p next) (waitfor-val next))
      ((null next) (give-value (cadr gval-rest)))
      ((let ((newend (make-alval-m)))
        (gval (ngen-gval (car gval-rest) (caar next))))
        (alset-alval (cons (cons gval (cadr next)) newend) p-result)
      (setq p-result newend)
    (setq p-call (cons 'a$gennmake (cons (cadr next) (cadr p-call)))
    (delay-me))))

(defun cl-$a$gdgen-next(mngrec)
  (cond ((alval-p (setq mngrec (strip-alval mngrec))) (cons 'sus mngrec))
      (not (mngrec-p mngrec))
    (al-error "$a$gdgen-next: mngrec expected in "$a$" mngrec))
  (mngrec))

(defun mdgen-next(mngrec)
  (let ((resend (strip-alval (mngrec-resend mngrec))))
    (cond ((alval-p resend) (cons 'sus resend))
      ((null-p resend) (cons 'suc al-f))
    (t ; Can now take a result.
      (when (if-gval-sus-proc (caar resend) p-suspended)
        (incf (proc-waitfor p-suspended)))
      (setf (mngrec-resend mngrec) (cdr resend))
      (cons 'suc (car resend))))))

(defun ngen-gval(gv1 gv2)
  (cond ((and (wake-up-gval-p gv1) (wake-up-gval-p gv2)) gv1)
      ((wake-up-gval-p gv1) gv1)
      ((wake-up-gval-p gv2) gv2)
    (let ((make-gval) pwait)
      (setq pwait (make-proc-m :call '($a$csuspended-gl ,gval)
                     :result dumb-alval :mu p-lmu :lazy nil
                     :waitfor 2 :suspended dumb-proc))
      (push pwait (gval-val gv1))
      (push pwait (gval-val gv2))
      ..."
(defun spec-$a$gen-next()
  (let ((gen (cdr p-call)))
    (cond ((alval-p (setq gen (strip-alval gen)))
       (waitfor-val gen)
       ((genstate-p gen)
       (setq p-call (list '$a$gen-next gen) (next-the-same)))
       ((mgrec-p gen)
       (setq p-call (list '$a$gen-next gen) (next-the-same)))
       ((and (fgrec-p gen) (genstate-p (fgrec-act gen)))
       (setq p-call (list '$a$gen-next (fgrec-act gen))) (next-the-same)))
    (fgrec-p gen)
    (setq p-call (list '$a$gen-next gen) (next-the-same)))
    ((mgrec-p gen)
    (setq p-call (list '$a$gen-next gen) (next-the-same)))
    ((al-error "Generator expected in: `A' p-call))))
  (defun ggen-results(gen)
    (cond ((genstate-p gen) (genstate-result gen))
      ((fgrec-p gen) (fgrec-result gen))
      ((mgrec-p gen) (mgrec-results gen))
      ((mgrec-p gen) (mgrec-results gen))
      ((al-error "Generator expected in: `A' gen")))
  ;-----
  ; Create association list for environment.
  (defun make-blist(names values)
    (make-blist2 names (strip-alval values)))
  (defun make-blist2(names values)
    (cond ((null-p names) nil)
      ((symbolp names) (list (cons names values)))
      ((cons-p names) (append (make-blist2 (car names) (car values))
        (make-blist2 (cdr names) (cdr values))))
      ((al-error "make-blist2: Unexpected name structure: `A' names")))
  (defun replace-syms(e blist)
    (cond ((cons-p e) (cons (replace-syms (car e) blist)
      (replace-syms (cdr e) blist)))
      ((slist-p e) (make-slist :expr (replace-syms (slist-expr e) blist)))
      ((symbol-p e) (let ((v (assoc e blist))
        (if v '(quote ,(cdr v) e)))
        (e)))
      (e)))
A.2.4  gcall.l

Handling of basic generators.

```lisp
(defvar (genstate)
  (result nil)
  (finished (make-queue))
  (resend nil)
  (next nil)
  (rest nil)
  (suspended (make-queue)))

(defvar (mgrec)
  (results nil)
  (resend nil)
  (sargens nil))

(defvar (apply-proc(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 '$aalfun-gfail))
    (cons 'suc coex)))

(defvar (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-g1 ,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))))

(defvar spec-alfun-gfail()
  (let ((gstate (muinfo-result p-lmu)))
    (alset-alval (genstate-rest gstate) (genstate-resend gstate))
    (setf (genstate-resend gstate) nil))) ; completed
(defun cl-$a$gen-results(e)
 (cl-$a$gen-results-u e))

(defun cl-$a$gen-results-u(e)
 (let ((sval (strip-alval e)))
  (cond ((alval-p sval) (cons 'sus sval))
        ((not (genstate-p sval)) (cons 'suc al-f))
        (t (cons 'suc (genstate-result sval)))))

(defun cl-$a$gen-next(e)
 (cl-$a$gen-next-u e))

(defun cl-$a$gen-next-u(e)
 (let ((sval (strip-alval e)))
  (cond ((alval-p sval) (cons 'sus sval))
        ((not (genstate-p sval)) (cons 'suc al-f))
        (t (gen-next sval)))))

(defun gen-next(coex)
 (let ((next (strip-alval (genstate-next coex)))
 (if (null q)
  (cond ((null-p next) (cons 'sus next))
        (t (al-error "gen-next: unaccounted generated values in: " next))
        (progn (check-suspended1 (queue-get q))
               (gen-next-coex next coex)))))

(defun cl-$a$gen-nextw(e)
 (cl-$a$gen-nextw-u e))

(defun cl-$a$gen-nextw-u(coex)
 (gen-next-coex (strip-alval (genstate-next coex)) coex))

(defun gen-next-coex(next coex)
 (cond ((alval-p next)
  (setq p-call '($a$gen-nextw ,coex))
  (cons 'sus next))
  (t (cons 'suc al-f))
  ((let ((res (car next)))
    (sigt (queue-get (genstate-finished coex))))
    (setf (genstate-next coex) (cdr next))
    (check-gral-sus sig)
    (cons 'suc res)))))))

(defun if-gval-sus-proc(sig proc)
 (when (listp (gval-val sig))
 (push proc (gval-val sig)))))

(defun check-gral-sus(sig)
(when (if-gval-sus-proc sig p-suspended)
  (setq p-suspended dumb-proc))

(defun spec-$a$&suspended-gl()
  (wake-up-gval (cadr p-call))
  (check-suspended))

(defun wake-up-gval-p(gval)
  (eq (gval-val gval) t))

(defun wake-up-gval(gval)
  (let ((l (gval-val gval))
        (while (cons-p l) (check-suspended1 (pop l))))
    (setq (gval-val gval) t))

;---------------------------------------------------------------
; Multiple coexpressions list producing coexpression.
(defun cl-$a$&sngen-call-sr(rest sergns &rest lcoex)
  (cl-$a$&sngen-call-u rest sergns lcoex))

(defun cl-$a$&sngen-call-sr(rest sergns &rest lcoex)
  (cl-$a$&sngen-call-u () sergns lcoex))

(defun cl-$a$&sngen-call-u(rest sergns lcoex)
  (cond ((alval-p (setq sergns (strip-alval sergns))) (cons 'sus sergns))
    ((my-member-ng
        (setq lcoex (strip-alval-e lcoex)))
      (cons 'sus mmn-ivar))
    ((null-p lcoex)
      (cons 'suc ())))
    ((let ((v (make-alval-m)) mgrec pmg)
      (setq mgrec (make-mgrec
        :results v :resend v :sergns sergns))
      (setq pmg (make-proc-m :call (list '$a$&sngenmaker (makegens lcoex)
        () () () mgrec rest)
        :result v :lmu p-lmu :waitfor 0
        :suspended dumb-proc))
      (delay-proc pmg) (push pmg proc-stack)
      (cons 'suc mgrec))))))

(defun delay-proc(proc)
  (unless p-lazy (setf (proc-lazy proc) 'delay)))

(defun makegens(clist)
  (when clist (cons (cons (car clist) (ggen-results (car clist)))
    (makegens (cdr clist))))))

(defun spec-$a$&sngenmaker()
  (cond ((cadr p-call) (apply 'mgmaker-recs (cadr p-call)))
    (t (apply 'mgmaker-ret (cadr p-call))))

(defun mgmaker-ret(lcoex lres lresok lresfin mgrec rest)
  (declare (ignore lcoex))
  (setq lresok (strip-alval-e lresok))
  (cond ((my-member-ng lresok) (waitfor-val mmn-ivar))
    ((not (my-andl lresok)) (give-value rest)))
The Interpreter of ALLOY in Common Lisp

(defun wait-on-gvals (lgs proc)
  (while lgs
    (when (if-gval-sus-proc (pop lgs) proc) (incf (proc-waitfor proc))))

(defun spec-%%mgreturn()
  (mgreturn-fin (cdr p-call)))

(defun mgreturn-fin (gval)
  (wake-up-gval gval)
  (check-suspended))

(defun mgmaker-recs (lcoex lres lresok lresfin mgrec rest)
  (cond ((null (cdar lcoex)) (give-value rest))
        ((not (alval-p (cdar lcoex)))
         (mg-ok lcoex lres lresok lresfin mgrec rest))
        ((alval-init (cdar lcoex))
         (mg-ok lcoex lres lresok lresfin mgrec rest))
        ((mg-gen lcoex lres lresok lresfin mgrec rest))))

(defun mg-ok (lcoexrs lres lresok lresfin mgrec rest)
  (let (midrest (make-alval-m))
    (coexrs (strip-alval-e (pop lcoexrs)))
gvalsus p1 p2)
    (setq gvalaus (cdadr coexrs))
    (setq coexrs (cons (car coexrs) (car coexrs))
                 (cdadr coexrs)))
    (incf (proc-waitfor p-suspended))
    (setq p2 (make-proc-m
              :call (list 'apply 'merge-gvals
                             (cons (cons (car coexrs) (cdadr coexrs)) lcoexrs)
                             lres lresok lresfin mgrec rest)
              :result midrest :lmu p-lmu
              :waitfor 0
              :suspended p-suspended))
    (push p2 proc-stack)
    (delay-proc p2))
  (setq p1 (make-proc-m
              :call (list 'apply 'merge-gvals
                             lcoexrs
                             (cons (cdadr coexrs) lres)
                             (cons t lresok) (cons gvalsus lresfin)
                             mgrec midrest)
              :result p-result :lmu p-lmu :waitfor 0
              :suspended p-suspended))
  (if (and (mgrec-sergens mgrec) (if-gval-sus-proc gvalsus p1))
      (incf (proc-waitfor p1))
(defun spec-$a$mgmaker-wait()
  (let* ((args (cdr p-call))
          (coex (pop args))
          (rrres (pop args))
          (lcoexrsres (pop args))
          (lrest (pop args)))
    (setq rrres (strip-alval rrres))
    (cond ((alval-p rrres) (waitfor-val rrres))
      ((null-p rrres) (give-value (car (last lrest))))
      ((mgm-gen (cons (cons coex (cdr rrres)) lcoexrsres)
                (pop lrest)) (pop lrest) (pop lrest) (pop lrest) (pop lrest))))
  )

; Actions taken as soon as generator starts producing a result.

(defun spec-$a$mgmaker-res()
  (let* ((args (cdr p-call))
          (okres (pop args))
          (oldres (pop args))
          (finres (pop args))
          (rrres (strip-alval rrres)))
    (cond ((alval-p rrres) (waitfor-val rrres))
      ((null-p rrres) (alset-alval nil okres) ; No result
       (alset-alval al-f oldres)
       (alset-alval al-f finres)
       (check-suspended))
      (t (alset-alval t okres)
       (alset-alval (cdar rrres) oldres)
       (alset-alval (caar rrres) finres)
       (check-suspended)))))

(defun mgm-gen(lcoexrs lres lresok lresfin mgrec rest)
  (let ((midrest (make-alval-m))
        (serr (mgrec-sergusms mgrec))
        (coexrs (pop lcoexrs))
        (okres (make-alval-m)) (oldres (make-alval-m)) (finres (make-alval-m))
        (p0 p1 p2)
        (incf (proc-waitfor p-suspended))
        (unless serr (incf (proc-waitfor p-suspended)))
        (make-processv (list `$a$mgmaker-res okres oldres finres (cdr coexrs))
                     dumb-alval)
        (setf (proc-lazy (car proc-stack)) nil)
        (setq p2 (make-proc-m)
              :call (list `$a$mgmaker-wait
                         (car coexrs) (cdr coexrs) lcoexrs
                         (list lres lresok lresfin mgrec rest))
              :result midrest :lmu p-lmu
              :waitfor 0
              :suspended p-suspended))
  (push p2 proc-stack)
  (delay-proc p2)
  (setq p1 (make-proc-m)
        :call (list `$a$mgmaker
                   lcoexrs))
(cons oldres lres)
(cons okres lresok) (cons finres lresfin)
mgrec midstest
:result p-result :lmu p-lmu
:waitfor (if serr 1 0)
:suspended p-suspended)
(unless serr (push p1 proc-stack))
(setq p0 (make-proc-m
:call (list `$a$gen-next (car coexrs))
:result dumb-alval :lmu p-lmu :waitfor 0 :lazy nil
:suspended (if serr p1 p-suspended)))
(push p0 proc-stack)))
(defun cl-$a$gen-next(mgrec)
(cond ((alval-p (setq mgrec (strip-alval mgrec))) (cons 'sus mgrec))
(not (mgrec-p mgrec))
(al-error "cl-$a$gen-next: Not multiple gens record: " mgrec))
(t (mgen-next mgrec))))
(defun mgen-next(mgrec)
(let ((resend (strip-alval (mgrec-resend mgrec))))
(cond ((alval-p resend) (cons 'sus resend))
(null-p resend) (cons 'suc al-f))
(t ; Can now take a result.
(when (if-gval-sus-proc (caar resend) p-suspended)
(incf (proc-waitfor p-suspended)))
(setf (mgrec-resend mgrec) (cdr resend))
(cons 'suc (car resend))))))
(defun make-process(e v)
(make-addproc (to-al-call e) v p-lmu 0 p-suspended)
(incf (proc-waitfor p-suspended))
(car proc-stack))
(defun my-and(&rest args)
(my-andl args))
(defun my-andl(l)
(if l (and (car l) (my-andl (cdr l))) t))
A.3 Pre-processor

A.3.1 prep.l

Basic pre-processor. Handles read macro expressions
1 ;-----------------------------------------------------------------
2 ; ALLOY preprocessor. Main part.
3 ; Read macros. Syntax checking of special expressions.
4
5 (defun al-error($rest msgs)
6   (throw 'al-prep (apply 'sformat msgs)))
7
8 (defun al-rpp()
9   (let ((e (al-prep (al-read))))
10     (al-print e)
11     (print e)
12     t))
13
14 ;-----------------------------------------------------------------
15 ; Mail preprocessing call. Takes care of error conditions.
16
17 (defun al-prep(e)
18   (let ((pe (catch 'al-prep (list (al-prep-main e)))))
19     (if (cons-p pe) (car pe)
20       (progn (p-error pe) al-f))))
21
22 (defun al-prep-main(e)
23   (oprep (gprep (prep e nil))))
24
25 (defun prep(e gok)
26   (cond ((rep-p e) (prep-rep0 e gok))
27       ((gen-p e) (prep-gen0 e gok))
28       ((not (xcons-p e)) e)
29       ((eq (xcar e) 'lets) (prep (prep-lets e) gok))
30       ((eq (xcar e) 'let) (prep (prep-let e) gok))
31       ((cons-p e) (prep-call e gok nil))
32       ((scons-p e) (setf (slist-expr e)
33         (prep-call (slist-expr e) gok t) e)))))
34
35 (defun prep-rep0(e gok) ; declare (ignore gok)
36   (let ((pe (prep-rep (rep-expr e))))
37     (if (outer-gen-p pe)
38       (progn (setf (rep-expr pe) pe) e)
39       pe)))
40
41 (defun prep-rep(e)
42   (prep e t))
43
44 (defun prep-gen0(e gok)
45   (unless gok
46     (al-error "Unexpected generator call: " e))
47     (when (rep-p (gen-expr e))
48       (al-error "Unexpected replicator call: " (gen-expr e)))
49     (let ((pe (prep-gen (gen-expr e))))
50       (when (and (gen-p pe) (gen-p (gen-expr pe))))
(setq pe (gen-expr pe))
(when (and (gen-p pe) (not (xcons-p (gen-expr pe))))
  (setq pe (gen-expr pe)))
(setf (gen-expr e) pe e))
(defun prep-gen(e)
  (defun prep-list(e gok)
    (cond ((null e) nil)
      ((list e) (prep e gok))
      ((cons (prep (car e) gok) (prep-list (cdr e) gok)))))
(defun prep-call(e gok serp)
  (letrec (pcheck-length e 2 9999) (prep (prep-lerecs e serp) gok))
  (letrec (pcheck-length e 2 9999) (prep (prep-lerecs e serp) gok))
  (function (pcheck-length e 2 2) (no-rep-the (cadr e))
    '(mu () ,(make-rep :expr '(list-return ,(cadr e)) nil))
    (quote (pcheck-length e 2 2) (no-rep-the (cadr e)) (prep-list e gok))
    (delay (pcheck-length e 2 2) (no-rep-the (cadr e)) (prep-list e gok))
    (eager (pcheck-length e 2 2) (no-rep-the (cadr e)) (prep-list e gok))
    (lazy (pcheck-length e 2 2) (no-rep-the (cadr e)) (prep-list e gok))
    (return (pcheck-length e 1 2) (no-rep-the (cadr e))
      (prep-list e gok))
    (1return (pcheck-length e 1 2) (no-rep-the (cadr e))
      (prep-list e gok))
    (setfun (pcheck-length e 2 9999)
      (if (cons-p (cadr e))
        (progn (pcheck-length e 2 9999) (no-rep-the (cadr e)))
        (progn (pcheck-length e 3 9999) (no-rep-the (cadr e))))
      (setq e (check-fdef (cdr e))))
    (list '$a$setfun1 (car e)
      (make-slist-t (cons 'mu (cons (cadr e) (prep-list (cadr e) nil))) serp))
    (mu (pcheck-length e 2 9999) (no-rep-the (cadr e))
      (setq e (check-fdef (cons 'none (cadr e))))
      (cons 'mu (cons (cadr e) (prep-list (cadr e) nil)))
    (set (pcheck-length e 3 3) (no-rep-the (cadr e))
      (no-rep-the (cadr e))
      (prep-set (cadr e) gok serp))
    (when (pcheck-length e 2 9999)
      (when (not-p (cadr e))
        (al-perror "Condition cannot be a replicator in: " e))
      (unless (pcheck-length e 2 9999)
        (when (not-p (cadr e))
          (al-perror "Condition cannot be a replicator in: " e))
        (cons 'unless (prep-list (cadr e) gok)))
    (if (pcheck-length e 3 4)
(when (rep-p (cadr e))
  (al-error "Condition cannot be a replicator in: "A" e))
)
(cond ((and (symbol-p (car fdef)) (al-args-p (cadr fdef)))
    (cons (car fdef) (cons (cadr fdef) (cdr fdef)))))
(al-args-p (car fdef))) (al-args-p (cadr fdef)))
(al-args-p (car fdef))) (al-args-p (cdr fdef)))))
(al-args-p (car fdef)))
(al-args-p (cdr fdef)))
(al-perror "Function definition expected in: "A" fdef)))

(defun al-args-p (args)
  (cond ((symbol-p args))
        ((not (cons-p args)) nil)
        ((al-args-p (car args)) (al-args-p (cdr args))))

;-----------------------------------------------------------------------
; Preprocessing of letrec and letrecs expressions.

(defun prep-letrecs (e serp)
  (prep-letrec e 'lets serp nil))

(defun prep-letrec (e serp)
  (prep-letrec e 'let serp nil))

(defun prep-gletrecs (e serp)
  (prep-gletrec e 'lets serp t))

(defun prep-gletrec (e serp)
  (prep-gletrec e 'let serp t))

(defun prep-letrec (e inlet serp genp)
  (let* ((parbind (slist-p (cadr e)))
         (bind (rlist (cadr e)))
         (args (plx-args bind))
         (inlent '([inlet , (make-slist-t (plx-aparset bind) parbind) ,@ (cdr e)]))
         (make-slist-t '([let , args
                          (make-gen-t (make-slist-t inlet serp genp) genp)) serp]))

(defun plx-aparset (e)
  (cond ((null-p e) nil)
        ((symbol-p (car e)) (plx-aparset (cdr e)))
        ((list-li-p (cdr (car e))) (cons (list (caar e) '(set , (caar e) , (cadar e))
                                              (plx-aparset (cdr e))))
        ((list-li-p (car e)) (plx-aparset (cdr e)))
        ((al-perror "Binding in <let> is not a list of one or two elements: "A" (car e)))

;-----------------------------------------------------------------------
; Preprocessing of let and lets expressions.

(defun prep-let (e)
  (prep-letx e 'let))

(defun prep-lets (e)
  (prep-letx e 'lets))

(defun prep-glet (e)
(defun prep-letx (e kind)
  (let ((parb t) (parp t) (body nil) (bind nil))
    (when (alerror "Unexpected end of <let> as: \"A\" e))
      (alerror "Unexpected end of bindings in <let> as: \"A\" e))
    (list (alerror "Binding in <let> is not a list of one or two elements: \"A\" (car e))))
  (cond ((null-p e) nil)
        (list (alerror "Binding in <let> is not a list of one or two elements: \"A\" (car e))))))

(defun plx-pars(e)
  (cond ((null-p e) nil)
        (symbol-p (car e)) (cons '%non-value (plx-pars (cdr e))))
        (list-p (cdr (car e))) (cons (cadar e) (plx-pars (cdr e))))
        (list-p (car e)) (cons '%non-value (plx-pars (cdr e))))
        (alerror "Binding in <let> is not a list of one or two elements: \"A\" (car e))))
  (defun plx-body(e kind parb)
    (case kind
      ('let (if e (plx-body e) nil))
      ('let (if e (plx-body e (not parb) nil) '(lreturn)))
      ('glet (if e (plx-body e (not parb) t) '(lreturn))))
    (cond ((null-p e) nil)
        (atom-p e) (alerror "Unexpected end of <let> as: \"A\" e))
        (alerror "Unexpected end of bindings in <let> as: \"A\" e))
        (symbol-p (car e)) (alerror "Binding in <let> is not a list of one or two elements: \"A\" (car e))))
    ; Preprocessing body of let.

, (make-rep :expr (list 'lreturn lst)))
  (serb))
(the-call
  (make-gen-t (make-slist-t (cons the-mu blst serb) gen)))
(list (make-rep-t (make-slist-t
  (list 'lreturn the-call) serb) gen)))
(list (make-rep :expr (list 'lreturn lst))))))

; (the-call (make-gen-t (make-slist-t (cons the-mu blst) serb) gen)))
; (list (make-rep-t (make-slist-t (list 'lreturn the-call) serb) gen))))
(defun outer-gen-p(e)
  (cond ((xcons-p e) (or (outer-gen-p (xcar e)) (outer-gen-p (xcdr e)))))
    ((gen-p e) t)
    ((rep-p e) nil)
    ((atom e) nil))

;------------------------------
; Testing

(defun rpp-file(fname)
  (let ((f (my-open-read-file fname))
    (unless f (al-error "Cannot open file for reading: " fname))
      (prog (e)
        start
          (setq e (al-read f)) (when (eq e al eof) (go exit))
          (pprint "<--") (al-print e)
          (setq e (al-prep e)) (when (eq e al f) (go start))
          (pprint "-->") (al-print e) (go start)
          exit
            (close f))
        t))

(defun make-slist-t(e ser)
  (if ser (make-slist :expr e) e))
(defun make-gen-t(e gen)
  (if gen (make-gen :expr e) e))
(defun make-rep-t(e rep)
  (if rep (make-rep :expr e) e))
(defun rm-slist(e)
  (if (slist-p e) (slist-expr e) e))
A.3.2  gprep.l

Pre-processing of generator/replicator related forms:

```lisp
;-----------------------------------------------
; Generator and Replicator Preprocessor.
;
(defvar trace-prep nil "Trace preprocessor output if true.")
;
;-----------------------------------------------
; Main call for generator/replicator pre-processing.

(defun gprep(e)
  (reset-gen-temp)
  (let ((final (gprepl '(list ,e))))
    (if (cons-p (cdr final))
      (setq final (cadr final)))
    (when trace-prep (pprint final))
    final))

(defun gprepl(e)
  (cond ((cons-p e) (gprep-list nil e))
        ((scons-p e) (setf (slist-expr e) (gprepl-list t (slist-expr e)) e)
         (gen-p e) (setf (gen-expr e) (gprepl (gen-expr e)) e)
         ((rep-p e) (al-error "Unexpected Replicator at: " e))
         (e)))))

(defun check-sharp!(l)
  (when (and (cons-p (rep-expr l)) (eql (car (rep-expr l)) '"$"))
    (let ((et l))
      (setq et (cadr (rep-expr et)))
      (if (and (cons-p et) (member (car et) '(return lreturn))
        (gen-p (cadr et)))
        (setf (gen-expr (cadr et))
          '(passnever,(make-slist-t
            ('$'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)))

(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 (gprepl (rep-expr e)))
            (gprepl-rep serp e (gprepl-list serp l) sh))
          (cons (gprepl (pop l)) (gprepl-list serp l)))))
  ((null l) nil)
  ((gprepl l)))))

;-----------------------------------------------
; Found replicator in front of e followed by rest.
```
(defun gprep-rep(serp e rest &optional eager)
  (setq e (list e))
  (let ((gems (remove-gens e)))
    (setq e (car e))
    (setq gens (simpl-gens gens))
    (let* ((names (sg-names gens))
      (cl '(lambda (1) (replace-syms (quote ,e) (make-blist (quote ,names) 1))))
      (gexec (sg-exec-m gens))
      (exrest (if (null rest) nil
                  (make-slist-t (cons 'list rest) serp)))
      (fres (make-slist :expr '($a$rep-driver , (true-zero serp) (quote ,cl)
                               , (if eager (list 'eager gexec) gexec)
                               (quote ,exrest))))
      fres)))))

(defun sg-names(gens)
  (cond ((null gens) nil)
    ((eq (car gens) '$sym$) (cdr gens))
    (t (when (atom (car gens)) (pop gens))
      (cons (sg-names (car gens)) (sg-names (cdr gens))))))

(defun sg-exec-m(gens)
  (case (car gens)
    ('$sym$' (car (sg-exlist (list gens))))
    ('$ser$ $par$' (let ((gexec (sg-exlist (cdr gens))))
                   (serp (case (car gens) ($ser$ t) ($par$ nil)))
                   ('$a$gen-call , (true-zero serp) ,gexec))
                   (otherwise (al-error "Unknown Generator preprocessing failed.")))))

(defun sg-exlist(gens)
  (cond ((null gens) nil)
    ((eq (caar gens) '$sym$) (cons (cadr gens) (sg-exlist (cdr gens))))
    ((cons (sg-exec-m (car gens)) (sg-exlist (cdr gens))))))

(defun simpl-gens(gens)
  (cond ((null gens) nil)
    ((eq (car gens) '$sym$) gens)
    (let ((r (simpl-glist (car gens) (cdr gens))))
      (case (length r)
        (0 nil)
        (1 (al-error "simpl-gens")
          (2 (cadr r))
          (otherwise r)))))))

(defun simpl-glist(mode 1)
  (when 1 (let ((sg1 (simpl-gens (car 1)))
                 (sgr (simpl-glist mode (cdr 1))))
    (cond ((and (null sg1) (null sgr)) nil)
      ((null sg1) sgr)
      ((null sgr) sg1)
      ((and (eq mode (car sg1)) (eq mode (car sgr)))
       (simpl-glist mode (cadr 1))))))
(append sg1 (cdr sgr))
(t
(unless (eq mode (car sg1)) (setq sg1 (list mode sg1)))
(append sg1 (if (eq mode (car sgr))
 (cdr sgr) (list sgr)))))

(defun remove-gens(e)
  (rg-main e))

(defun rg-main(e)
  (cond ((cons-p e) (cons 'par$ (rg-list e)))
        ((scons-p e) (cons 'ser$ (rg-list (slist-expr e)))))

(defun rg-list(e)
  (when (cons-p e)
    (cons (if (gen-p (car e))
      (let ((e (gen-expr (car e)))
          (s (gen-temp)))
        (setf (car e) s)
      (list 'sym$ s (rem-from-gen ece)))
    (rg-main (car e))))
  (if (gen-p (cdr e))
    (let ((e (gen-expr (cdr e)))
          (s (gen-temp)))
      (setf (cdr e) s)
    (list 'sym$ s (rem-from-gen ece)))
  (rg-list (cdr e)))))

(defun rem-from-gen(ingen)
  (cond ((cons-p ingen)
          (remove-ggens ingen))
        ((remove-ggens 'passtrue ,ingen))))

(defun remove-ggens(e)
  (let ((serp (slist-p e)) gens)
    (setq gens (simpl-gens (rg-main e)))
    (when serp (setq e (slist-expr e))
      (if (null gens) '($a$gen-call-gq ,true-zero serp ,@e)
        (let* ((names (sg-names gens))
          (cl '(lambda (l) (replace-syms (quote ,make-slist-t e serp) )
            (make-blist (quote ,names) l)))
          (exec (sg-exec-m gens))
          (fres ('$a$gen-driver ,true-zero serp (quote ,c1 ,exec)))
          (fres))))))"
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:

```lisp
(defun prep(e) (oprep1 e))
(defun oprep1(e)
  (cond ((xcons-p e)
          (case (xcar e)
            (class-start (oprep-cs (cdr (rm-slist e))))
            (class-end (oprep-ce (cdr (rm-slist e))))
            (otherwise e)))))
(defun oprep-ce(defs)
  (unless (cons-p defs) (al-perror "$Name of class expected in car of: "$)
   (al-perror "$A" (car
defs)))
  (unless (symbol-p (car defs)) (al-perror "$Name of class expected in: "$ (car
defs)))
  (let ((cname (pop defs)))
    (when defs (al-perror "$Unknown features of class in: "$ (car
defs))
      (list 's$class-end cname)))
(defun prep-cs(defs)
  (unless (cons-p defs) (al-perror "$Name of class expected in: "$ (car
defs)))
  (unless (symbol-p (car defs)) (al-perror "$Name of class expected in: "$ (car
defs)))
  (let ((cname (pop defs)) (description nil) (tmp nil))
    ;; (take-item 'inherit)
    (take-item 'common)
    (take-item 'static)
    (take-item 'dynamic)
    (take-item 'import)
    (take-item 'part)
    (take-item 'methods)
    (take-interfaces)
    (when defs (al-perror "$Unexpected feature of class "$ (car
     (name desc) "$A in: "$ (car
defs))
    (list 's$desc-end desc (reverse description)))))
(defun take-interfaces()
  (let ((dl desc)
    (setq description nil)
    (take-item 'interface)
    (while (cdar description) (take-item 'interface))
    (pop description)
    (push (list 'interface all-methods ,0(cdr (assoc 'methods dl)))
          (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description))
                            (setq description (cons (cons 'interfaces (join-inters description)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))```
(defmacro take-item(type)
  `(progn (setf defs (cdr (assoc-fm defe (assoc-rm defe ,type))))
    (push (car tmp) description)
    (check-symlist (car tmp)))
)

(defun check-symlist(l)
  (cond ((null l))
    ((atom-p l) (al-perror "Unexpected end of list in: " (symbol-name l)))
    ((not (symbol-p (car l))) (al-perror "Symbol expected in: " (symbol-name (car l))))
    ((check-symlist (cdr l)))))

;;-----
;Find key in al. Create it if it does not exist and assoc it with nil.

(defun assoc-rm(al key)
  (let ((v (assoc-rm al key)))
    (if (car v) v (cons (cons key nil) (cdr v)))))

(defun assoc-rm(al key)
  (let* ((val (assoc-rm al key))
    ((al2 (assoc-rm-v al key val))))
    (cons (car val) al2)))

(defun assoc-rm-v(al key val)
  (cond ((null al) nil)
    ((atom-p al) (al-perror "End of list expected in: " (symbol-name al)))
    ((atom-p (car al)) (al-perror "List expected in: " (symbol-name (car al))))
    ((not (symbol-p (caar al))) (al-perror "Symbol expected in: " (symbol-name (caar al))))
    ((eq (caar al) key) (setf (car val) (car al)) (cdr al))
    (al2)
    (cons (car al) (assoc-rm-v (cdr al) key val))))

```
A.4 Utilities

A.4.1 io.l

Input output utilities:

```lisp
(defstruct (rep
  "A Replicator Data Structure"
  (expr nil))
(defstruct (gen
  "A generator Data Structure"
  (expr nil))
(defstruct (slist
  "A Square List"
  (expr nil))
(defstruct (rlist
  "A Round List"
  (expr nil))
(defmacro scons-p(e)
  '(slist-p ,e))
(defun s-list(&rest args)
  (make-s-list :expr args))
(defun get-cl-table() (eval '(copy-readtable nil))
(defun cl-table (get-cl-table))
(defun al-table (get-cl-table))
(defun makenamespec()
  (let ((n 255) (aa (character '|A|)))
    (while (>= n 0) (reset-mc (int-char n) aa) (decf n)))
(defun reset-mc(c optional from)
  (set-syntax-from-char c (if from from c) *readtable* cl-table))
```
(defun make-al-table()
    (setq *readtable* al-table)
    (makenospec)
    (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)
    (define-pass)

(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)))
        (if e (make-slist :expr e) nil))

(cond ((eq e end) nil)
  ((eq e other) (throw 'al-read "Unmatched parenthesis.")

  ((eq e dot) (let ((le1 (read-one stream))
    (le2 (read-any stream)))
    (when (eq other le2)
      (throw 'al-read "Mismathed Parenthesis.")
    (unless (eq end le2)
      (throw 'al-read "Parenthesis expected.")
    le1))
  (cons e (al-read-list stream end other))))

(defun read-dot (stream char)
  (declare (ignore stream))
  char)

(defun read-cpar (stream char)
  (declare (ignore stream))
  char)

(defun read-cspar (stream char)
  (declare (ignore stream))
  char)

(defun read-one (stream)
  (let ((e (read-any stream)))
    (when (or (eq e cpar) (eq e cspar))
      (throw 'al-read "Unexpected closing Parenthesis.")
    e))

(defun read-any (stream)
  (let ((e (read stream nil al-f nil)))
    (when (eq e al-f) (throw 'al-read "Unexpected End of Input.")
    e))

;-----
; Main reader function.

(defun al-read (&optional stream quiet)
  (when (null stream) (setq stream *standard-input*)
  (let ((expr nil))
    (when (= (incf al-readers 1) 1) (setq *readtable* al-table))
    (setq expr (catch 'al-read (list (read stream nil al-eof nil))))
    (when (= (decf al-readers 1) 0) (setq *readtable* cl-table))
    (if (consp expr) (car expr)
      (progn (unless quiet (p-error expr)) al-f))))

;-----
; Main printer function.

(defun al-print (e &optional stream)
  (when (null stream) (setq stream *standard-output*)
  (cond ((numberp e) (princ e stream))
    ((symbolp e) (princ e stream))
(((stringp e) (princ e stream))
((cons-p e) (al-print-list e opar cspar stream))
((slist-p e) (al-print-list (slist-expr e) opar cspar stream))
((rep-p e) (princ '|#| stream) (al-print (rep-expr e) stream))
((gen-p e) (princ '|*| stream) (al-print (gen-expr e) stream))
((alval-p e) (princ '|v| stream))
((eq al-f e) (princ '|f| stream))
((ext-function-p e) (princ '|%(object-of-class CLOSURE)| stream))
((class-p e) (princ '|%(class | stream)
 (princ (class-name e) stream ) (princ '|)| stream))
((object-p e) (princ '|%(object-of-class | stream)
 (princ (object-name e) stream ) (princ '|)| stream))
(t (princ '|%u| stream))))

(defun al-print-list(e op cl stream)
 (cond ((null e) (princ '|nil| stream))
 (t (princ op stream)
 (al-print (car e) stream) (al-print-listx (cdr e) stream)
 (princ cl stream))) t)

(defun al-print-listx(e stream)
 (cond ((null-p e) nil)
 (t ((cons-p e) (princ '| | stream) (al-print (car e) stream)
 (al-print-listx (cdr e) stream))
 (t (princ '| . | stream) (al-print e stream))))
A.4.2 values.l

Local and global environment handling functions:

```lisp
(defconstant hassvstr '**1 numOfVal*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)))))

(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 sassg-p(varname)
  `(eq (char (string ,varname) 0) char-und))

(defmacro sassg-p(varname)
  `(not (sassg-p ,varname)))

; ALLO local environments.

(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 maker-fail(vars)
  (cond ((null vars))
    ((symbol-p vars) (push (cons vars al-f) pb-bs))
    ((cons-p vars) (make-fail (car vars)) (make-fail (cdr vars)))
    ((al-error "Invalid parameters in: " vars))))

(defun new-env2(vars vals)
  (cond ((null vals))
    (if (null vars) (error "No environment match!"))
    (if (null askew) (error "No environment askew!"))
    (if (null env) (error "No environment!"))
    (if (null vars) (error "No environment match!"))
    (if (null askew) (error "No environment askew!"))
    (if (null env) (error "No environment!"))
    (if (null vars) (error "No environment match!"))
    (if (null askew) (error "No environment askew!"))
    (if (null env) (error "No environment!")))

(defun get-val-env(var e)
  (cond ((null e) nil)
    ((assoc var (car e))
      ((get-val-env var (cdr e))))))

(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)))))

(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))))
```
((symbol-p vars) ; end of vars is a var
  (new-bind vars vals))
((null vals) (make-fail vars))
((atom-p vals) ; end of vals is a val
  (struct-bind vars vals))
((cons-p vars)
  (new-env2 (car vars) (car vals))
  (new-env2 (cdr vars) (cdr vals))))
((al-error "Invalid parameters in: " vars)))

(defun new-bind(name val)
  (push (cons name (if (eq val al-n) (make-alval) val)) pb-bs))
(defun struct-bind(1 how)
  (cond ((null 1))
    ((symbol-p 1) (let ((var (make-alval)))
      (push (cons 1 var) pb-bs)
      (push (list '$_set-v1 var how) pb-s)))
    ((cons-p 1) (struct-bind (car 1) (car ,how))
      (struct-bind (cdr 1) (cdr ,how)))
    ((al-error "Invalid parameter structure in: " 1)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  Hash table.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun make-sym-table()
  (make-hash-table :test 'eq :size 10))
(defun has-sym-value(sym-table name)
  (gethash name sym-table hassvstr))
(defun has-sym-value-p(sym-table name)
  (not (eq (gethash name sym-table hassvstr) hassvstr)))
(defun get-sym-value(sym-table name)
  (let ((hval (gethash name sym-table hassvstr)))
    (if (eq hval hassvstr) (set-sym-valuec sym-table name
      (if (massg-p name) nil (make-alval)))
    hval)))
(defun set-sym-value(sym-table name value)
  (if (has-sym-value-p sym-table name)
    (set-sym-valuec sym-table name value)
    (set-sym-valuec sym-table name value)))

; name is known not to exist in table.  Must be created.
(defun set-sym-valuec(sym-table name value)
  (if create-vars (setf (gethash name sym-table) value)
    (progn (p-warning "Cannot define new variable: " name)
      al-f)))))

; name is known to exist in table.
(defun set-sym-valuex(sym-table name value)
  (setf (gethash name sym-table) value))
;-------------------------------------------------------------------------------------------------------------------
; Interpreters special function table.

(defun get-spec-value(name)
  (let ((hval (gethash name spec-table al-t)))
    (if (eq hval al-t) nil (list hval))))

(defun set-spec-value(name value)
  (setf (gethash name spec-table) value))

(defun add-spec(name value)
  (set-spec-value name value))

(defun add-specs(l)
  (when l (add-spec (caar l) (cdar l))
    (add-specs (cdr l)))))
A.4.3 alutil.l

This file contains utilities more specific to ALLOY:

1 ;---------------------------------------------------------------
2 ; General utilities to ALLOY interpreter.
3 ;---------------------------------------------------------------
4 ; Special values.
5 ;---------------------------------------------------------------
6 (defconstant al-eof "***end-of-file*** "End of file starting ALLOY Reading.")
7 (defstruct (al-failure))
8 (defstruct (al-success))
9 (defstruct (al-nonvalue))
10 (defconstant al-f (make-al-failure))
11 (defconstant al-t (make-al-success))
12 (defconstant al-n (make-al-nonvalue))
13 (defconstant lazyd 't))
14 (defconstant lazyt t)
15 (defconstant lazy1 nil)
16 (defun lazy-delay-p(e)
17 (consp e)))
18 (defun lazy-delay1-p(e)
19 (eq e 'delay))
20 (defstruct (error)
21 (expr "Some error occurred.")
22 ;---------------------------------------------------------------
23 ; Utilities.
24 ;---------------------------------------------------------------
25 (defun strip-alval(e)
26 (cond ((and (alval-p e) (alval-init e))
27 (setf (alval-value e) (strip-alval (alval-value e))))
28 (e)))
29 (defun strip-alval-e(e)
30 (cond ((consp e) (setf (car e) (strip-alval-e (car e)))
31 (setf (cdr e) (strip-alval-e (cdr e))))
32 ((slist-p e) (setf (slist-expr e) (strip-alval-e (slist-expr e))))
33 (let ((es (strip-alval e)))
34 (unless (eql es e) (setq e (strip-alval-e es)))))
35 e)
36 (defun strip-alval-1(e)
37 (cond ((consp e) (setf (car e) (strip-alval (car e)))
38 (setf (cdr e) (strip-alval-1 (cdr e))))
39 ((slist-p e) (setf (slist-expr e) (strip-alval-1 (slist-expr e))))
40 (let ((es (strip-alval e)))
41 (unless (eql es e) (setq e (strip-alval-1 es)))))
42 e)
(defun ground-p (e)
  (cond ((xcons-p e) (and (ground-p (xcar e)) (ground-p (xcdr e))))
        ((and (alval-p e) (not (alval-init e))) nil)
        (t)))
)

(defun member-ng (l)
  (cond ((xcons-p l) (or (member-ng (xcar l)) (member-ng (xcdr l))))
        ((and (alval-p l) (not (alval-init l))) l)))
)

; Macros

(deffunc noteq (x y)
  ' (not (eq ,x ,y)))

(defun compile-all-alloy() (shell (str-app4 "cat safe.1 util.1 alutil.1 prep.1 gprep.1 oprep.1"
  "load.1 values.1 io.1 inter.1"
  "fcall.1 gcall.1 rcall.1 ocall.1"
  "build.1 build2.1 obuild.1 >> allal.1"))
  (compile-file "allal.1")
)

(defun compile-all-alloy-f() (shell (str-app4 "cat fast.1 util.1 alutil.1 prep.1 gprep.1 oprep.1"
  "load.1 values.1 io.1 inter.1"
  "fcall.1 gcall.1 rcall.1 ocall.1"
  "build.1 build2.1 obuild.1 >> allal.1"))
  (compile-file "allal.1")
)

(defun compile-all-alloy() (mapc 'compile-file '("util.1" "alutil.1" "prep.1" "gprep.1"
  "oprep.1" "load.1"
  "values.1" "io.1" "inter.1" "fcall.1" "gcall.1"
  "rcall.1" "ocall.1" "build.1" "build2.1" "obuild.1")))

(defun load-alloy() (mapc 'load '("util" "alutil" "prep" "gprep" "oprep" "load"
  "values" "io" "inter" "fcall" "gcall"
  "rcall" "ocall" "build" "build2" "obuild")))

(defun compile-all-alloy-gcl() (sys:dos "makesafe"
  (compile-file "allal.1"))
)

(defun compile-all-alloy-f-gcl() (sys:dos "makefast"
  (compile-file "allal.1"))
)

(defun make-proc-m($rest args)
  `(progn (incf proc-noex) (make-proc ,@args)))
)

; Create muinfo. Preserve counter.
(defun make-muinfo-m($rest args)
  `(progn (incf muinfo-noex) (make-muinfo ,@args)))
(defmacro make-alval-m(&rest args)
  '(progn (incf alval-nosx) (make-alval ,@args)))

(defun xcons-p(e)
  (or (cons-p e) (scons-p e)))

(defun xcar(e)
  (cond ((cons-p e) (car e))
        ((scons-p e) (car (slist-expr e)))
        ((al-error "xcar expected cons or scons in "A" e)))))

(defun xcdr(e)
  (cond ((cons-p e) (cdr e))
        ((scons-p e) (let ((e (cdr (slist-expr e))))
                      (if e (make-slist :expr e nil)))
        ((al-error "xcdr expected cons or scons in "A" e)))))

(defvar rm-dot-t nil "True if last rm-dot-list acted."

(defun rm-dot-list(e)
  (setq rm-dot-t nil)
  (rm-dot-list1 e))

(defun rm-dot-list1(e)
  (cond ((null e) nil)
        ((atom e) (setq rm-dot-t t)
        (list (if (rlist-p e) (rlist-expr e) e)))
        ((cons-p e) (cons (car e) (rm-dot-list1 (cdr e))))))

(defun nput-dot-list-nc(e)
  (when rm-dot-t
    (cond ((ccdr e) (nput-dot-list-nc (cdr e)))
      (setf (cdr e) (ccdr e)))))

(defun p-error(&rest msgs)
  (terpri) (princ "ALLOY ERROR -- ")
  (princ (apply 'sformat msgs) *error-output*)
  (terpri) nil)

(defun p-errorc(&rest msgs)
  (princ " -- ")
  (princ (apply 'sformat msgs) *error-output*)
  (terpri) nil)

(defun p-warning(&rest msgs)
  (terpri) (princ "ALLOY WARNING -- ")
  (princ (apply 'sformat msgs) *error-output*)
  (terpri) nil)

(defun p-warninc(&rest msgs)
  (princ " -- ")
  (princ (apply 'sformat msgs) *error-output*)
  (terpri) nil)
A.4 Utilities

163
164 (defun al-error(str &rest msgs)
165 (apply 'error (cons (sformat ""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:

```lisp
(defun loadf (fname)
  (load (format nil "-A" fname ".l")))

(defun atom-apply (x y)
  (intern (format nil "-A" x y)))

(defun str-apply (x y z w)
  (format nil "-A" z x y w))

(defun sformat (&rest args)
  (apply 'format (cons nil args)))

(defun return-value
  '((return-from 'exit ,value))

(defun rpush (stack val)
  '(push ,val ,stack))

(defun set-list-end (l end)
  '(append ,l ,end))

(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) (cadr (queue-tail q)))
    x))

(defun queue-put (x q)
  (setf (cadr (queue-head q)) (cons x nil)))
```
(setf (queue-head q) (cdr (queue-head q))) x)

(defmacro while (cond &rest args)
  '(do ()
    ((not ,cond)
      ,@args))
)

(defmacro until (cond &rest args)
  '(do ()
    ,(cond
      ,@args))
)

(defmacro cons-p (e)
  '(consp ,e))

(defmacro atom-p (e)
  '(atom ,e))

(defmacro null-p (e)
  '(null ,e))

(defmacro number-p (e)
  '(numberp ,e))

(defmacro symbol-p (e)
  '(symbolp ,e))

(defun list-nil-p (e)
  (and (cons-p e) (null-p (cdr e))))

(defun full-list-p (e)
  (cond ((null e)
        ((cons-p e) (full-list-p (cdr e))))))

(defun my-open-read-file (fname)
  (open fname :direction :input :if-does-not-exist nil))

(defun my-open-write-file (fname)
  (open fname :direction :output))

(defun my-close-file (fname)
  (close (open fname)))

(defun p-msg (&rest msgs)
  (princ (apply 'sformat msgs) *error-output*)
  (terpri) nil)

(defun remainder (x y)
  (* (cdr (multiple-value-list (floor (/ x y)))) y))

(defun last-p (e)
  (cond ((cons-p e) (atom-p (cdr e)))
    ((al-error "last-p: List expected in: " e))))

(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 1)) 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:

```lisp
(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 global2global(al-name)
  (let ((args-body (strip-alval (get-sym-value tab al-name))))
    (set-global 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 's$funcall-vs 'al-funcall-vs)
(add-spec 's$alfunc-fail 'spec-alfunc-fail)
(add-spec 's$alfunc-ffail 'spec-alfunc-ffail)
(add-spec 's$if-v 'spec-$s$if-v)
(add-spec 's$set-v1 'spec-$s$set-v1)
(add-spec 's$pass-v 'spec-$s$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 's$when-v 'spec-$s$when-v)
(add-spec 'if 'spec-if)
(add-spec 'delay 'spec-delay)
(add-spec 'eager 'spec-eager)
(add-spec 'lazy 'spec-lazy)
(add-spec1 'unless)
(add-spec1 'fail)
(add-spec1 'lfail)
(add-spec1 's$no-result)
(add-spec1 's$reset1)
(add-spec1 's$setfun1)
(add-spec1 's$class-start)
(add-spec1 's$class-end)

;;;;-------------------------------------------------------------------------------
 ;; Generator specific.
;;;;-------------------------------------------------------------------------------
(add-spec1 's$csuspended-g1)
(set-noninst 's$gen-call-er t)
(set-noninst 's$gen-call t)
(set-noninst 's$gen-next t)
(set-noninst 's$gen-nextw t)
(set-noninst 's$gen-results t)
(add-spec1 'a$gendlmakep)
(add-spec1 'a$gendlmake)
(add-spec1 'a$gndllist)
(add-spec1 'a$gend calls-dl)

(set-noninst 'a$gende call-ar t)
(set-noninst 'a$gende call t)
(set-noninst 'a$gende next t)
(add-spec1 'a$gend maker)
(add-spec1 'a$gend maker-wait)
(add-spec1 'a$gend maker-res)
(add-spec1 'a$gend return)

(add-spec1 'a$gende call-gq)
(add-spec1 'a$gred-loop)
(add-spec1 'a$gred-exec)

(add-spec1 'a$gend next)
(add-spec1 'a$gend loop)
(add-spec1 'a$gend loop-cont)

(set-noninst 'a$grep driver t)
(set-noninst 'a$gend driver t)
(set-noninst 'a$gendgen next t)

(set-argone 'a$gende call t)
(set-noninst 'a$gend next t)
(add-spec 'a$gend call-nomu 'al-funcall-nomu)
(set-noninst 'a$gend call-nomu final t)
(set-noninst 'a$gend call-nomu direct g t)
(set-noninst 'a$gend dirg final t)

(add-spec1 'a$shar)

;----------------------------------------------------------
; Build in basic non-instant Functions.

(defmacro list-l1(1)
  '(endp (cdr ,1)))

(set-noninst 'a$end finish eval t)
(defun cl-1 a$end finish eval(1)
  (let ((val (strip-alval-e arg)))
    (unless (and (symbol-p val) (eq val 'a$no-result))
      (princ "" => ") (al-print val) (terpri))
    (cons 'suc t)))

(set-noninst 'a$pass1 t)
(defun cl-1 a$pass1(arg &rest args) (declare (ignore args))
  (cons 'suc arg))

(defun spec-unless()
  (setq p call (when (not , (cdr p call)) , (cddr p call)))
  (next-the-same))
(set-noninst 'caar t)
(set-noninst 'cadr t)
(set-noninst 'cdar t)
(set-noninst 'cdr t)

(defun cl-caar(arg &rest args)  (declare (ignore args))
(let ((p (make-pprocess ' (car (car ($a$pass-v ,arg)))))
  (cons 'suc (proc-result p))))

(defun cl-cadr(arg &rest args)  (declare (ignore args))
(let ((p (make-pprocess ' (car (cdr ($a$pass-v ,arg)))))
  (cons 'suc (proc-result p))))

(defun cl-cdar(arg &rest args)  (declare (ignore args))
(let ((p (make-pprocess ' (cdr (car ($a$pass-v ,arg)))))
  (cons 'suc (proc-result p))))

(defun cl-cddr(arg &rest args)  (declare (ignore args))
(let ((p (make-pprocess ' (cdr (cdr ($a$pass-v ,arg)))))
  (cons 'suc (proc-result p))))

(set-globalb 'passtrue
'((x) (when x (return x))) t)

(set-globalb 'passnever
'()) t)

(defun spec-$a$sharp())
(setq p-call (cdr p-call))
(check-proc-call)
(next-the-same))

;-----------------------------------------------
; argone system functions.

(defun condval($rest args)
  '(let ((eval 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 'sus sval))
          ,0 args))
)

(defun condvar($rest args)
  '(let ((eval nil))
    (cond ((alval-p (setq args (strip-alval args)))
      (cons 'sus args))
          ((atom-p args)
           (cons 'suc al-f))
          ((prog1 nil (setq sval (strip-alval (car args))))
           ,0 args))
)

(set-argone '$a$call-lisp t)
(defun cl-$a$call-lisp(args)
  (condval ((cons 'suc (eval sval)))))

(set-argone '$a$make-bi t)
\begin{verbatim}
215  (defun cl-$a$make-bi(args)
216    (condval ((cons 'suc (global2globalb sval)))))
217
218  (set-argone 'load t)
219  (defun cl-load(args)
220    (condval ((cons 'suc (suc-al-load sval nil)))))
221
222  (set-argone 'include t)
223  (defun cl-include(args)
224    (condval ((cons 'suc (suc-al-load sval t)))))
225
226  (defun suc-al-load(sval output)
227    (let ((pl (construct-proc))
228      (v (al-load sval output)))
229      (setq proc pl) (expand-proc)
230      v))
231
232  (set-argone 'provide t)
233  (defun cl-provide(args)
234    (condval ((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              (if (my-open-read-file pfname))
246                (if f (progl 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 'suc 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    (condval ((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)
269                      nil)
270)
(if (has-sym-value-p globale-tab key) t al-f))))
    al-f))))

; ------------------------------------------------------------------
; Other System functions.

(set-argone 'trace t)
(defun cl-trace(args)
  (declare (ignore args))
    (cond (al-trace (setq al-trace nil) (cons 'suc al-f))
    (t (setq al-trace t) (cons 'suc t)))))

(set-argone 'time-prompt)
(defun cl-time-prompt(args)
  (declare (ignore args))
    (cond (al-time-prompt (setq al-time-prompt nil) (cons 'suc al-f))
    (t (setq al-time-prompt t) (cons 'suc t)))))

(set-argone 'stats-prompt)
(defun cl-stats-prompt(args)
  (declare (ignore args))
    (cond (al-stats-prompt (setq al-stats-prompt nil) (cons 'suc al-f))
    (t (setq al-stats-prompt t) (cons 'suc t)))))

(set-argone 'warning-exec)
(defun cl-warning-exec(args)
  (declare (ignore args))
    (cond (warning-exec (setq warning-exec nil) (cons 'suc al-f))
    (t (setq warning-exec t) (cons 'suc t)))))

(set-argone 'warning-load)
(defun cl-warning-load(args)
  (declare (ignore args))
    (cond (warning-load (setq warning-load nil) (cons 'suc al-f))
    (t (setq warning-load t) (cons 'suc t)))))

(set-argone 'exit t)
(defun cl-exit(args)
  (declare (ignore args))
    (setq exit-alloy-lisp t)
    (cons 'suc (setq exit-alloy-loop t)))

(set-argone 'bye t)
(defun cl-bye(args)
  (declare (ignore args))
    (cl-exit 1))

(set-argone 'quit t)
(defun cl-quit(args)
  (declare (ignore args))
    (cl-exit 1))

(set-argone 'quit-al)
(defun cl-quit-al(args)
  (declare (ignore args))
    (cons 'suc (setq exit-alloy-loop t)))

(set-argone 'scheduler-switch-steps)
(defun cl-scheduler-switch-steps(args)
  (let ((aval nil))
    (cond ((alval-p (setq args (strip-alval args))) (cons 'sus args))
    ((null-p args) (cons 'suc gtp-max))
    ((atom-p args) (cons 'suc al-f))
(alval-p (setq sval (strip-alval (car args)))) (cons 'sus sval))
((and (integerp sval) (> sval 0))
 (cons 'suc (setq gtq-max sval)))
((cons 'suc al-f))))
(set-argone 'block1 t)
(defun cl-block1(args)
 (condvar ((cons 'suc sval))))
(set-argone 'blockn t)
(defun cl-blockn(args)
 (cl-last-arg args))
(set-argone 'block t)
(defun cl-block(args) (declare (ignore args))
 (cons 'suc t))
A.5.2 build2.l

Other basic built in functions:

1 ;************************************************************************************************
2 ; Other built in predicates.
3 ;************************************************************************************************
4 ; Functions with one argument containing all actual arguments. Do not suspend.
5 ;************************************************************************************************
6 (set-argone 'list t)
7 (defun cl-list(args)
8 (cons 'suc args))
9
10 (set-argone 'nl)
11 (defun cl-nl(args &optional stream)
12 (terpri stream) (force-output stream)
13 (cons 'suc t))
14
15 ;************************************************************************************************
16 ; Functions with one argument containing all actual arguments. May suspend.
17 ;************************************************************************************************
18 (set-argone 'var-p)
19 (defun cl-var-p(args)
20 (cl-var args))
21
22 (set-argone 'var)
23 (defun cl-var(args)
24 (let ((sval nil))
25 (cond ((alval-p (setq args (strip-alval args)))
26 (cons 'suc args))
27 ((atom-p args)
28 (cons 'suc nil)))
29 ((cons 'suc null))))
30
31 (set-argone '$a$funcall t)
32 (defun cl-$a$funcall(args)
33 (cond ((alval-p (setq args (strip-alval args)))
34 (cons 'suc args))
35 ((atom-p args)
36 (cons 'suc nil)))
37 (let ((p (make-process (make-list-t (cons '$a$funcall-vs args) p-serial)))
38 (cons 'suc (proc-result p)))))
39
40 (set-argone 'read)
41 (defun cl-read(args &optional stream)
42 (cons 'suc (let ((e (al-read stream)))
43 (if (eq e al-eof) al-f e))))
44
45 (set-argone 'print)
46 (defun cl-print(args &optional stream)
47 (cond ((alval-p (setq args (strip-alval args)))
48 (cons 'suc args))
49 ((atom-p args)
50 (cons 'suc nil)))
51 (let ((val (strip-alval-e (car args)))
52 (al-print val stream) (force-output stream)
53 (cons 'suc val)))))
54
55 (set-argone 'println)
56 (defun cl-println(args &optional stream)
(cond ((alval-p (setq args (strip-alval args))) (cons 'sucs args)))
((atom-p args) (terpri stream) (cons 'suc al-f)))
((let ((val (strip-alval-e (car args))))
   (al-print val stream) (terpri stream) (force-output stream)
   (cons 'suc val))))

(set-argone 'random t)
(defun cl-random(args)
 (condval ((number-p sval) (cons 'suc (random sval)))
 (t (cons 'suc al-f))))

(set-argone 'car t)
(defun cl-car(args)
 (condval ((cons p sval) (cons 'suc (car sval)))
 (t (cons 'suc al-f))))

(set-argone 'cdr t)
(defun cl-cdr(args)
 (condval ((cons p sval) (cons 'suc (cdr sval)))
 (t (cons 'suc al-f))))

(set-argone 'scar t)
(defun cl-scar(args)
 (condval ((slist-p sval) (cons 'suc (car (slist-expr sval)))
 (t (cons 'suc al-f))))

(set-argone 'scdr t)
(defun cl-scdr(args)
 (condval ((slist-p sval) (cons 'suc (let ((c (cdr (slist-expr sval))))
 (if (cons-p c) (make-slist :expr c) c))))
 (t (cons 'suc al-f))))

(set-argone 'data)
(defun cl-data(args)
 (condval (t (cons 'suc sval))))

(set-argone 'null-p t)
(defun cl-null-p(args)
 (condval ((null sval) (cons 'suc sval))
 (t (cons 'suc al-f))))

(set-argone 'number-p)
(defun cl-number-p(args)
 (condval ((number-p sval) (cons 'suc sval))
 (t (cons 'suc al-f))))

(set-argone 'string-p)
(defun cl-string-p(args)
 (condval ((string-p sval) (cons 'suc sval))
 (t (cons 'suc al-f))))

(set-argone 'closure-p t)
(defun cl-closure-p(args)
 (condval ((ext-function-p sval) (cons 'suc sval))
 (t (cons 'suc al-f))))
(set-argone 'atom-p t)
(defun cl-atom-p(args)
  (condval ((consp sval) (cons 'suc al-f))
            ((alist-p sval) (cons 'suc al-f))
            (t (cons 'suc sval))))
(set-argone 'cons-p t)
(defun cl-cons-p(args)
  (condval ((consp sval) (cons 'suc sval))
            (t (cons 'suc al-f))))
(set-argone 'scons-p t)
(defun cl-scons-p(args)
  (condval ((alist-p sval) (cons 'suc sval))
            (t (cons 'suc al-f))))
(set-argone 'not t)
(defun cl-not(args)
  (condval ((eq sval al-f) (cons 'suc t))
            (t (cons 'suc al-f))))

; Other functions. May need an argument to have a value.

(deffunc wait-out(e)
  '(catch 'out (cons 'suc ,e)))
(deffunc get-out(e)
  '(throw 'out (cons 'suc ,e)))
(setq al-cons (set-argone 'cons t))
(defun cl-cons(args)
  (cond ((alval-p (setq args (strip-alval-l args))) (cons 'suc args)
         ((atom-p args) (cons 'suc nil))
         ((alval-p (cdr args)) (cons 'suc (cdr args)))
         ((atom-p (cdr args)) (cons 'suc (car args)))
         ((cons 'suc (cons (car args) (cadr args)))))
(setq al-scons (set-argone 'scons t))
(defun cl-scons(args)
  (cond ((alval-p (setq args (strip-alval-l args))) (cons 'suc args)
         ((atom-p args) (cons 'suc nil))
         ((alval-p (cdr args)) (cons 'suc (cdr args)))
         ((atom-p (cdr args)) (cons 'suc (car args)))
         ((cons 'suc (make-slist :expr (cons (car args) (cadr args)))))))
(set-argone 'last-arg t)
(defun cl-last-arg(args)
  (cond ((alval-p (setq args (strip-alval-l args))) (cons 'suc args)
         ((atom-p args) (cons 'suc al-f))
         ((wait-out (cl-last-argl args)))))
(defun cl-last-argl(args)
  (let ((rest (cdr args)))
    (not (null rest))))
(cond ((alval-p rest) (get-out rest))
  ((atom-p rest) (car args))
  ((cl-last-arg1 (cdr args))))

`; Arithmetic functions.

(defun mng-var nil "Variable containing value of last call to my-member-ng."
  (setq mng-var (member-ng 1))

(defun cl-arith(op x rest)
  (cond ((alval-p rest) (get-out rest))
    ((atom-p rest) x)
    ((alval-p (car rest)) (get-out (car rest)))
    ((not (numberp (car rest))) al-f)
    ((cl-arith op (funcall op x (car rest)) (cdr rest)))))

(set-argone 'sum)
(defun cl-sum(args)
  (cond ((alval-p (setq args (strip-alval-1 args))) (cons 'sum args))
    ((wait-out (cl-arith '+ 0 args)))))

(set-argone 'times)
(defun cl-times(args)
  (cond ((alval-p (setq args (strip-alval-1 args))) (cons 'sum args))
    ((wait-out (cl-arith '* 1 args)))))

(set-argone 'diff)
(defun cl-diff(args)
  (cond ((alval-p (setq args (strip-alval-1 args))) (cons 'sum args))
    ((atom-p args) (cons 'suc 0))
    ((alval-p (car args)) (cons 'suc (car args)))
    ((alval-p (cdr args)) (cons 'suc (cdr args)))
    ((not (numberp (car args))) (cons 'suc al-f))
    ((atom-p (cdr args)) (cons 'suc (- 0 (car args))))
    ((wait-out (cl-arith '- (car args) (cdr args))))))

(set-argone 'div)
(defun cl-div(args)
  (cond ((alval-p (setq args (strip-alval-1 args))) (cons 'sum args))
    ((atom-p args) (cons 'suc 1))
    ((alval-p (car args)) (cons 'suc (car args)))
    ((alval-p (cdr args)) (cons 'suc (cdr args)))
    ((not (numberp (car args))) (cons 'suc al-f))
    ((atom-p (cdr args)) (cons 'suc (/ 1 (car args))))
    ((wait-out (cl-arith '/ (car args) (cdr args))))))

(defun cl-quotx2(x y) (floor (/ x y)))))

(set-argone 'quotient)
(defun cl-quotient(args)
  (cond ((alval-p (setq args (strip-alval-1 args))) (cons 'sum args))
    ((atom-p args) (cons 'suc 1)))
((alval-p (car args)) (cons 'sus (car args)))
((alval-p (cdr args)) (cons 'sus (cdr args)))
((not (number-p (car args))) (cons 'suc al-f))
((atom-p (cdr args)) (cons 'suc (cl-quot2 1 (car args))))
((wait-out (cl-arith 'cl-quot2 (car args) (cdr args))))

(set-argone 'remainder)
(defun cl-remainder(args)
  (cond ((alval-p (setq args (strip-alval-l args))) (cons 'sus args))
        ((atom-p args) (cons 'suc 1))
        ((alval-p (car args)) (cons 'sus (car args)))
        ((alval-p (cdr args)) (cons 'sus (cdr args)))
        ((not (number-p (car args))) (cons 'suc al-f))
        ((atom-p (cdr args)) (cons 'suc (remainder 1 (car args))))
        ((wait-out (cl-arith 'remainder (car args) (cdr args)))))

;set-argone 'eql t
(defun cl-eql(args)
  (cond ((alval-p (setq args (strip-alval-l args))) (cons 'sus args))
        ((atom-p args) (cons 'suc t))
        ((alval-p (car args)) (cons 'sus (car args)))
        ((alval-p (car args)) (cons 'sus (car args)))
        ((wait-out (if (cl-comp 'eql (car args) (cdr args))
                       (car args) al-f)))

;set-argone 'lt
(defun cl-lt(args)
  (cond ((alval-p (setq args (strip-alval-l args))) (cons 'sus args))
        ((atom-p args) (cons 'suc t))
        ((alval-p (car args)) (cons 'sus (car args)))
        ((not (number-p (car args))) (cons 'suc al-f))
        ((wait-out (if (cl-arcomp '< (car args) (cdr args))
                       (car args) al-f)))

;set-argone 'gt
(defun cl-gt(args)
  (cond ((alval-p (setq args (strip-alval-l args))) (cons 'sus args))
A.5 Built in functions and objects

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-1 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-1 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-1 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:

```lisp
(defun make-bclass(name interfaces)
  (let ((cl-names (cons (atom-app 'cl-c-name) (atom-app 'cl-o-name)))
        (make-class :origdef 'interface (name ,0 name) (interfaces ,0 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)))))))

(defvar bclass-p(c)
  'eq (class-ndynnames ,c '**bclass**))

(defvar 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)))

(defvar bobject-value(obj)
  'object-inienv ,obj))

(defvar condsvs1(&rest args)
  'let ((eql-nil))
  (cond ((alval-p (setq args (strip-alval args)))
         (waitfor-val args))
        ((atom-p args)
         (give-value al-f))
```

The Interpreter of ALLOY in Common Lisp
A.5 Built in functions and objects

```
((alval-p (setq sval (strip-alval (pop args)))) (wait-for-val sval))
,@ (args)))

(defun condsvl2(&rest args)
  'let ((sval nil) (sval2 nil))
  (cond ((alval-p (setq args (strip-alval args))) (wait-for-val args))
        ((atom-p args) (give-value al-f))
        ((alval-p (setq sval (strip-alval (pop args)))) (wait-for-val sval))
        ((alval-p (setq args (strip-alval args))) (wait-for-val args))
        ((atom-p args) (give-value al-f))
        ((alval-p (setq sval2 (strip-alval (pop args))))(wait-for-val sval2))
        ,@ (args))))
```

; Library objects, with build in special representation.

;---------------------------------------------------------------

; Closure

(set-bclass 'closure '((all-methods val eval)) t)

(defun cl-c-closure(msg c args) (declare (ignore args c))
  (give-value (case msg
       (new al-f)))
  (new (make-gfunction '(() (return))))))

(defun cl-o-closure(msg value args) (declare (ignore value))
  (case msg
       (val (condsvl
             (t (give-value sval)))))
       (eval (condsvl
             (t (give-value nil)))))))

;---------------------------------------------------------------

; Number

(set-bclass 'number '((all-methods val string)) t)

(defun cl-c-number(msg c args) (declare (ignore args c))
  (give-value (case msg
       (new 0))))

(defun cl-o-number(msg value args) (declare (ignore value))
  (case msg
       (val (condsvl
             (t (give-value sval)))))
       (string (condsvl
             (t (give-value (format nil ""A" sval)))))))

;---------------------------------------------------------------
108 ; String
109 (set-bclass 'string '((all-methods val atom number explode append)) t)
110 (defconstant dumb-o-string (make-bobject (get-sym-value globale-tab 'string) 0))
111 (defun cl-c-string(msg c args) (declare (ignore args c))
112 (give-value (case msg
113     (new "")))))
114 (defun cl-o-string(msg value args) (declare (ignore value))
115     (case msg
116         (val (condsval
117             (t (give-value sval)))))
118         (atom (condsval1 (t (give-value (intern sval)))))
119         (number (condsval1 (t (give-value (let ((n (al-read
120             (make-string-input-stream
121             sval) t)))
122             (if (number-p n n al-f)))))
123             (explode (condsval1 (t (give-value (explode sval))))))
124             (append (let ((x (someend 'nstringp (setq args (strip-alval-e args)))))
125                 (cond ((null x) (give-value (implode args)))
126                     ((alval-p x) (waitfor-val x))
127                     ((and (cons-p x) (alval-p (car x))) (waitfor-val (car x))))
128                     ((give-value al-f)))))))))
129 (defun nstringp(x) (not (stringp x)))
130
131 ;---------------------------------------------------------------
132 ; Cons
133 (set-bclass 'cons '((all-methods val car cdr)) t)
134 (defconstant al-cons-cl (get-sym-value globale-tab 'cons))
135 (defconstant dumb-o-cons (make-bobject al-cons-cl 0))
136 (defun round-cons-p (e) (eql al-cons-cl e))
137 (defun cl-c-cons(msg c args) (declare (ignore c))
138     (case msg
139         (new (condsval2
140             (t (give-value (cons sval sval)))))))
140 (defun cl-o-cons(msg value args) (declare (ignore value))
141     (case msg
142         (car (condsval1
143             (t (give-value (car sval))))))
144         (cdr (condsval1
145             (t (give-value (cdr sval))))))
146         (val (condsval1
147             (t (give-value sval))))))
148 ;---------------------------------------------------------------
149 ; Scons
(set-bclass 'scons '((all-methods val car cdr)) t)
(defconstant al-scons-cl (get-sym-value globale-tab 'scons))
(defconstant dumb-oscons (make-bobject al-scons-cl '() ))
(defun square-cons-p (e) (eql al-scons-cl e))
(defun cl-c-scons(msg c args) (declare (ignore c))
  (case msg
    (new (condsvs2
      (t (give-value (make-slist :expr (cons sval sval2))))))
    (defun cl-o-scons(msg value args) (declare (ignore value))
      (case msg
        (car (condsvs1
          (t (give-value (car (slist-expr sval))))))
        (cdr (condsvs1
          (t (give-value (let ((c (cdr (slist-expr sval)))
            (if (cons-p c) (make-slist :expr c) c))))))
        (val (condsvs1
          (t (give-value sval)))))
    (case msg
      (val (condsvs1
        (t (give-value sval))))
      (string (condsvs1
        (t (give-value (string sval)))))))))
);---------------------------------------------------------------
; Symbol
(set-bclass 'symbol '((all-methods val string)) t)
(defconstant dumb-osymbol (make-bobject (get-sym-value globale-tab 'symbol) 0))
(defun cl-c-symbol(msg c args) (declare (ignore args c))
  (give-value (case msg
    (new nil))))
(defun cl-o-symbol(msg value args) (declare (ignore value))
  (case msg
    (val (condsvs1
      (t (give-value sval))))
    (string (condsvs1
      (t (give-value (string sval)))))))))
);---------------------------------------------------------------
; Special-Class
(set-bclass 'special-class '((all-methods val)) t)
(defconstant dumb-ospecial-class (make-bobject (get-sym-value globale-tab 'special-class) 0))
(defun cl-c-special-class(msg c args) (declare (ignore args c))
  (give-value (case msg
    (new al-f))))
(defun cl-o-special-class(msg value args) (declare (ignore value))
  (case msg
(val (condsv1
     (t (give-value sval)))))

;---------------------------------------------------------------
; Main Body of built-in objects driver:

(defun send-bi-obj(msg obj args orig-obj)
  (cond ((number-p obj) (send-message msg dumb-onumber (cons obj args) orig-obj))
        ((symbol-p obj) (send-message msg dumb-osymbol (cons obj args) orig-obj))
        ((string-p obj) (send-message msg dumb-ostring (cons obj args) orig-obj))
        ((cons-p obj) (send-message msg dumb-oscons
                         (cons obj args) orig-obj))
        ((ext-function-p obj)
         (send-message msg dumb-oclosure (cons obj args) orig-obj))
        (t (send-message msg dumb-ospecial-class
                         (cons obj args) orig-obj)))))

;---------------------------------------------------------------
; Library objects.

;---------------------------------------------------------------
; Multiple assignment variable.

(defun cl-c-ma-var(msg c args)
  (case msg
    (new (if (alval-p (setq args (strip-alval args)))
       (waitfor-val args)
       (give-value (make-bobject
                    c (make-gval :val (if (cons-p args) (car args)
                                    al-f)))))
    (get (give-value (gval-val value)))
    (put (if (alval-p (setq args (strip-alval args)))
       (waitfor-val args)
       (give-value (setf (gval-val value) (if (cons-p args) (car args)
                                    al-f)))))
  ))

(defun cl-o-ma-var(value args)
  (case msg
    (get (give-value (gval-val value)))
    (put (if (alval-p (setq args (strip-alval args)))
       (waitfor-val args)
       (give-value (setf (gval-val value) (if (cons-p args) (car args)
                                    al-f)))))
  ))

;---------------------------------------------------------------
; Single assignment variable.

(defun cl-bc-class 'sa-var '((all-methods get put) (reader get) (writer put)))

(defun cl-c-sa-var(msg c args)
  (case msg
    (new (if (alval-p (setq args (strip-alval args)))
       (waitfor-val args)
       (give-value (make-bobject c (if (cons-p args)
                                    (cons t (car args))
                                    (cons nil (make-alval-m)))
       ))
    (get (give-value (gval-val value)))
    (put (if (alval-p (setq args (strip-alval args)))
       (waitfor-val args)
       (give-value (setf (gval-val value) (if (cons-p args) (car args)
                                    al-f))))
  ))
(defun cl-o-sa-var (msg value args)
  (case msg
    (get (give-value (cdr value)))
    (put (if (alval-p (setq args (strip-alval args)))
      (waitfor-val args)
      (give-value (if (car value)
          al-f
          (propl (car args) (setf (car value) t)
            (alset-alval (if (cons-p args) (car args) al-f)
              (cdr value)))))))
  ;---------------------------------------------------------------
  ; Single Assignment Dictionary.
  (set-bclass 'dictionary '((all-methods get put) (reader get) (writer put)))
  (defun cl-c-dictionary (msg c args) (declare (ignore args))
    (give-value (case msg
      (new (make-bobject c (make-hash-table
        :size 10 :test 'equal)))))))
  (defmacro symnap (v)
    `(or (numberp ,v) (symbolp ,v)))
  (defun cl-o-dictionary (msg value args)
    (case msg
      (get (conds1
        ;
        ((not (symnap sval)) (give-value al-f))
        ((give-value (let ((v (has-sym-value sval)))
          (cond ((eq v '**1*noval*1**)
            (set-sym-value sval (make-alval-m))
            (v)))))))
      (put (conds2
        ;
        ((not (symnap sval)) (give-value al-f))
        ((give-value (let ((v (has-sym-value sval)))
          (cond ((eq v '**1*noval*1**)
            (set-sym-value sval sval2))
            ((alval-p (setq v (strip-alval v)))
              (alset-alval sval2 v) sval2)
            (al-f))))))))#
  ;---------------------------------------------------------------
  ; Single Assignment Vector.
  (set-bclass 'vector '((all-methods get put) (reader get) (writer put)))
  (defun cl-c-vector (msg c args)
    (case msg
      (new (conds1
        ;
        ((not (numberp sval)) (give-value al-f))
        ((give-value (let ((v (make-array (list sval)
          :initial-element '**1*noval*1**)))
          (make-bobject c v)))))))
(defun cl-o-vector(msg value args)
  (case msg
    (get (conds1)
      ((not (symmp sval)) (give-value al-f))
      ((not (array-in-bounds-p value sval)) (give-value al-f))
      ((give-value (let ((v (aref value sval))))
        (cond ((eq v '***inval*1**)
           (setf (aref value sval) (make-alval-m)))
           (v))))))
    (put (conds2)
      ((not (symmp sval)) (give-value al-f))
      ((not (array-in-bounds-p value sval)) (give-value al-f))
      ((give-value (let ((v (aref value sval))))
        (cond ((eq v '***inval*1**)
           (setf (aref value sval) sval2))
           ((alval-p (setq v (strip-alval v)))
            (alset-alval sval2 v) sval2)
           (al-f))))))))))

;-------------------------------------------------------------
; Multiple Assignment Vector.
;-------------------------------------------------------------
(set-bclass 'ma-vector '((all-methods get put) (reader get) (writer put)))

(defun cl-c-ma-vector(msg c args)
  (give-value
   (case msg
     (new (conds1)
       ((not (numberp sval)) (give-value al-f))
       ((let ((v (make-array (list sval)
                       :initial-element 'nil)))
         (make-object c v))))))

(defun cl-o-ma-vector(msg value args)
  (case msg
    (get (conds1)
      ((not (symmp sval)) (give-value al-f))
      ((not (array-in-bounds-p value sval)) (give-value al-f))
      ((give-value (aref value sval))))))
    (put (conds2)
      ((not (symmp sval)) (give-value al-f))
      ((not (array-in-bounds-p value sval)) (give-value al-f))
      ((give-value (setf (aref value sval) sval2)))
      (v))))))

;-------------------------------------------------------------
; Multiple Assignment Dictionary.
;-------------------------------------------------------------
(set-bclass 'ma-dictionary '((all-methods get put del) (reader get) (writer put del)))

(defun cl-c-ma-dictionary(msg c args)  (declare (ignore args))
  (give-value (case msg
    (new (make-object c (make-hash-table
                        :size 10 :test 'equal)))))
(defmacro symnamp(v)
  `(or (numberp ,v) (symbolp ,v)))

(defun cl-o-ma-dictionary(msg value args)
  (case msg
    (get (condsv1
      ; ((not (symnamp sval)) (give-value al-f))
      ((give-value (let ((v (has-symn Value sval)))
          (cond ((eq v '**1=imonial1** al-f)
            (v))))))
      (del (condsv1
        ; ((not (symnamp sval)) (give-value al-f))
        ((give-value (let ((v (has-symn Value sval)))
          (cond ((eq v '**1=imonial1** al-f)
            (t (remhash sval value) v))))))
      (put (condsv2
        ; ((not (symnamp sval)) (give-value al-f))
        ((give-value (set-symn Value sval sval2))))))
    ))

; Top Level Caller. Used to make calls charged at the top level.
; Useful to define monitors without suspending the creator.
(set-bclass 'top-level-caller
  '((all-methods top-level-funcall) () ()))

(defun cl-c-top-level-caller(msg c args) (declare (ignore args))
  (give-value (case msg
    (new (make-object c nil)))))

(defun cl-o-top-level-caller(msg value args) (declare (ignore value))
  (case msg
    (top-level-funcall
      (let ((proc (top-level-proc (make-slist-t (list '$a$pass1 args) p-serial))))
        (push proc proc-stack)
        (give-value (proc-result proc))))))

; Input.
(set-bclass 'input
  '((all-methods read-read-line read-string1
    eof empty active close) () ()))

(defconstant in-con (make-object (get-symn Value global-tab 'input) (cons nil nil))
  (set-symn Value global-tab 'input-console in-con)

(defun cl-c-input(msg c args)
  (case msg
    (new
      (if (null args) (give-value (make-object c (cons nil nil)))
        (condsv1
          ((not (stringp sval)) (give-value al-f)))))

A.5  Built in functions and objects
(equal sval "console") (give-value (make-bobject c (cons nil nil)))
((give-value (let ((stream (my-open-read-file sval)))
    (if (eq al-f stream al-f
      (make-bobject c (cons nil stream)))))))
(defun cl-o-input(msg value args)
  (declare (ignore args))
  (let ((stream (cdr value)))
    (case msg
      (read-line (give-value (read-line stream nil al-f)))
      (read-string1 (give-value (let ((c (read-char stream nil al-f)))
        (if (eq c al-f) al-f
          (string c)))))
      (read (give-value (let ((e (al-read stream)))
        (if (eq e al-eof) al-f e)))))
      (eof (give-value (if (eq al-eof (peek-char nil stream nil al-eof))
        t al-f)))
      (empty (give-value (if (peek-char nil stream nil al-f t))))
      (close (setf (car value) t) (when stream (close stream)))
      (value t))
    (active (give-value t)))))
;set-bclass 'output
'(all-methods print printnl nl active close) () ()
(defconstant out-con (make-bobject (get-sym-value global-tab 'output) (cons nil nil)))
(set-sym-value global-tab 'output-console out-con)
(defconstant err-con (make-bobject (get-sym-value global-tab 'output)
  (cons nil *error-output*))
(set-sym-value global-tab 'error-console err-con)
(defun cl-c-output(msg c args)
  (case msg
    (new
      (if (null args) (give-value (make-bobject c (cons nil nil)))
        (condsvl
          ((not (stringp sval)) (give-value al-f))
          ((equal sval "console") (give-value (make-bobject c (cons nil nil)))
            (give-value (make-bobject c (cons nil *error-output*)))))
          (give-value (let ((stream (my-open-write-file sval)))
            (if (eq al-f stream al-f
              (make-bobject c (cons nil stream)))))))
    (print (cl-print args stream) (give-value args))
    (printnl (cl-printnl args stream) (give-value args))
(nl (cl-nil nil stream) (give-value t))
(close (setf (car value) t) (when stream (close stream))
(give-value t))
(active (give-value t))))))
A.6 Library in ALLOY

A.6.1 alloylib.a

Higher level built in functions written in ALLOY:

```lisp
;-----------------------------
; Alloy Library written in Alloy.
; Change these to point to your own directories of ALLOY program utilities.

(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 (xmake-bi 1)
  (when (cons-p 1) ($a$xmake-bi (car l)) (xmake-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:

```lisp
;--------------------------------------
;; 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 l) (nth (diff n 1) (cdr l)) (car l))))

(setfun cdrn n l)
  (return (if (ge n l) (cdrn (diff n 1) (cdr l)) l)))

(setfun carn n l)
  (return (if (ge n l) (carn (diff n 1) (car l)) l)))

(setfun first-n-list n l)
  (return (if (ge n l) (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 l) 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  boola

Utilities dealing with boolean expressions:

```
1 ; Boolean expressions
2
3 (setfun (and . 1)
4   (if (cons-p 1)
5       [block (block (unless (car 1) (fail))
6          (unless (and . [cdr 1]) (fail)))
7          (return (car 1))]
8     (return))
9
10 (setfun (and-call . 1)
11   (if (cons-p 1)
12      [lets (xv) (block [unless (set xv ((car 1))) (fail)]
13          [unless (and-call . [cdr 1]) (fail)])
14          (return xv))
15     (return))
16
17 (setfun (or . 1)
18   (when (cons-p 1) (lets (rv xv)
19      (when (set xv (car 1)) (return xv))
20      (when (set rv (or . [cdr 1])) (return rv)))))
21
22 (setfun (or-call . 1)
23   (when (cons-p 1) (lets (rv xv)
24      [when (set xv ((car 1))) (return xv)]
25      [when (set rv (or-call . [cdr 1])) (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)))
40                  (return x))
41     [return (data x)]))
42
43 (xxmake-bi ’(ground and or and-call or-call equal))
```
A.6.4 `loop.a`

Utilities dealing with loops:

```alloy
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 ; Miscellaneous.
3
4 (setfun gvalue()
5 (return (let ((v ('new sa-var)))
6 (list (writer v) (get v)))))
7
8 (rxmake-bi '(gvalue))
A.6.6  higherf.a

Higher order functions:
1 ;----------------------
2 ; Support functions for higher level functional programming.
3 ;----------------------
4 (provide "higherf")
5 ;----------------------
6 ; Drivers of functions
7 (setfun reduce(f init l)
8     (return (if (cons-p l) (f (car l) (reduce f init (cdr l)))
9         init)))
10 (setfun filter(pred l)
11     (return (if (cons-p l)
12         (if (pred (car l)) (cons (car l) (filter pred (cdr l)))
13         (filter pred (cdr l))))
14     nil)))
15 (setfun mapcar2(f2 l1 l2)
16     (return (if (cons-p l1)
17         (cons (f2 (car l1) (car l2)) (mapcar2 f2 (cdr l1) (cdr l2)))
18         ()))
19 (setfun mapcarn(fn . l1) (return (mapcarn-fix fn l1)))
20 (setfun mapcarn-fix(fn l1)
21     (return (if (cons-p (car l1))
22         (cons (let ((cars (mapcar car l1))) (fn . cars))
23         (mapcarn-fix fn (mapcar cdr l1)))
24     ()))
25 ;----------------------
26 ; These Produce functions
27 (setfun h-bu(f2 a1)
28     (return (mu(a2) (lreturn (f2 a1 a2))))
29 (setfun h-rev2(f2)
30     (return (mu(x y) (lreturn (f2 y x))))
31 (setfun h-rev(f)
32     (return (mu x (lreturn (f . [reverse 1])))))
33 (setfun h-red(f2 a)
34     (return (mu1) (lreturn (reduce f2 a l))))
35 (setfun h-dup(f2)
36     (return (mu(x) (lreturn (f2 x x))))
37 (setfun h-comp(f g)
38     (return (mu x (lreturn (f (g . x))))))
(setfun h-const(a)
  (return (mu x (lreturn a)))))

(setfun h-map(f1)
  (return (mu(1) (lreturn (mapcar f1 1)))))

(setfun h-map2(f2)
  (return (mu(1 12) (lreturn (mapcar2 f2 11 12))))))

(setfun h-mapn(fn)
  (return (mu 11 (lreturn (mapcar n fn . 11))))))

(setfun h-construct lfs
  (return (mu(arg) (lreturn ((h-map (mu(f) (lreturn (f arg)))) lfs))))))

(setfun h-compn lfs
  (return (mu (arg) (lreturn (ccallse lfs arg))))))

(setfun ccallse (lfs e)
  (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 (mapcar 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 section 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:

```lisp
# Make the ALLOY interpreter.
# Tested with AKCL Common Lisp interpreter.
# If another version of Common Lisp is used correct file alakcl.l
# If lisp system does not require a CR to give line for processing
# (e.g. Golden CL) check function interactive in file load.l
# You must change the path in file alloylib.a to point to your library dir.
# This is the library of ALLOY sources (in progs.tar.Z)
# In this Makefile, replace all occurrances of ".o"
# with the suffix of your lisp's compiled files.

# Or if alloy is not available use these:
LISP=cl
PREEXEC='(load "alloy.l")'

# If an old version of ALLOY is available use these:
LISP=alloy
PREEXEC="(quit-al)"

# Main part of Makefile

.SUFFIXES: .l.o .a
SHELL=/bin/sh

OBJS=fast.o util.o alutil.o prep.o gprep.o oprep.o load.o values.o io.o
     inter.o fcall.o gcall.o rcall.o ocall.o build.o build2.o obuild.o
LSFS=fast.l util.l alutil.l prep.l gprep.l oprep.l load.l values.l io.l
     inter.l fcall.l gcall.l rcall.l ocall.l build.l build2.l obuild.l
ALIB=alloylib.a lists.a loop.a bool.a misc.a higherf.a fp.a

akclalloy: sysal
@echo DONE
sysal: $(OBJS) $(ALIB)
makesysal
.l.o: execfirst $(LISP) $(PREEXEC) "(compile-file "$*.l")"

Compile all files after putting them into one file. SLOW.
onesysal: allal.o $(ALIB)
```
A.7 Creating ALLOY

48 makesall
49 mv -f sysal onesysal
50
51 allal.o: allal.1
52 execfirst ${LISP} ${PREEXEC} "(compile-file ""$*.1"")"
53
54 allal.1: ${LPS}
55 rm -f allal.1
56 cat ${LPS} > allal.1
57
58 clean:
59 rm -fr *.o * allal.1 RCS SCCS
A.7.2 execfirst

The following executable file is used by make and allows automated execution of lisp commands from UNIX:

```bash
#!/bin/csh -f
# Call first arg and give as input rest args.
set EXEC=$1
shift

${EXEC} << ENDEMD
$*
ENDEMD
```
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`:

```bash
#!/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`:

```lisp
(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")
```

241
(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, whut88 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(11 12)
  (return (if (cons-p 11)
              (cons (car 11) (app (cdr 11) 12))
              12))

(setfun nrev(1)
  (return (if (cons-p 1)
              (app (nrev (cdr 1)) (list (car 1)))
              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-list1(n)
  (return (if (lt n 1) nil (cons n (make-list1 (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-list1 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.
; Benchmark 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
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 output 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 ssort1)
12  (return (if (null-p 1)
13       nil
14       (let ((rest ('new putlist))
15                  (cons (ssortx (car 1) (cdr 1) rest)
16                  (ssort ('get-orig-head rest)))))))
17
18 (setfun ssortx(x 1 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 lazily.
7 ;--------------------------------------------------------------------------
8
9 (setfun lprimes()
10  (return (lazy (primes))))
11
12 (setfun primes()
13  (return (cons 1 (sieve (ints 2)))))
14
15 (setfun sieve()
16  (return (cons (car 1) (sieve (filter (car 1) (cdr 1))))))
17
18 (setfun filter(x 1)
19  (if (equal (remainder (car 1) x) 0) (return (filter x (cdr 1))))
20  (return (cons (car 1) (filter x (cdr 1))))))
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 1)
4  [when (gt i 0) [print (data (car 1))] (nl) (printn (diff i 1) (cdr 1))])
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.
B.4 Generators

; If called appropriately it will return all permutations of the list.
;---------------------------------------------------------------
; This program demonstrates how a non-deterministic function can be made
; to return all possible solutions.
;---------------------------------------------------------------
(provide "perms")

(defun permutations (l)
  (if (null l)
      (list nil)
      (let ((rest (rest l)) (x (first l)))
        (append (mapcar #'(lambda (p) (cons x p))
                       (permutations rest))
                (permutations rest)))))

(defun lpermutations (l)
  (if (null l)
      (list nil)
      (let ((rest (rest l)) (x (first l)))
        (append (mapcar #'(lambda (p) (cons x p))
                         (lpermutations rest))
                (lpermutations rest)))))

(defun delete-one (l)
  (when (null l)
    (list l))
  (let ((rest (rest l)) (x (first l)))
    (cons x (delete-one rest))))

B.4.3 Queens

;---------------------------------------------------------------
; Benchmark set no 10.
; 4 queens. One solution and all solutions.
;---------------------------------------------------------------
(load "queens.a")

(defun bench10a ()
  (return (queens 4)))

(defun bench10b ()
  (return (list #(queens 4))))

(file "queens.a":
  ;----------------- ALLOY version 0.1 ----------------- 10/10/89 ------------------
  ; queens.a
  ; Solves the n queens problem.
  ; Evaluates and returns one or all possible places, as requested.
  ;---------------------------------------------------------------
  ; This program demonstrates the ability to find multiple solutions
  ; while at the same time maximum parallelism is used.
  ; Here, Checking of a configuration proceeds in parallel with the
  ; creation of the permutation that describes the configuration.
  ;---------------------------------------------------------------
  (require "perms")

(defun queens (n)
15  (let ((l (make-list n)))
16    #\{(let ((poslist (board lst *(permutations lst))))
17       (when (not (unsafeall poslist)) (printl poslist) (return poslist))))
18    )
19  )
20  (setfun unsafeall(board)
21    (when (cons-p board)
22       (when (unsafe (car board) (cdr board)) (return))
23       (when (unsafeall (cdr board)) (return)))
24    )
25  )
26  (setfun unsafe(q board)
27    (when (cons-p board)
28       (when (unsafe q (car board)) (return))
29       (when (unsafe q (cdr board)) (return))
30    )
31  )
32  (setfun abs(x)
33    (return (if (lt x 0) (diff x) x)))
34  )
35  (setfun unsafe(q p)
36    (when (eq (abs (diff (car q) (car p)))
37       (abs (diff (cdr q) (cdr p)))) (return))
38  )
39  (setfun board(lst places)
40    (return (if (null-p lst) ()
41       (cons (cons (car lst) (car places))
42       (board (cdr lst) (cdr places))))))
43  )
44  (setfun make-list(n)
45    (if (ge n 1)
46       (return (cons n (make-list (diff n 1))))
47       (return ())))
48  )

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 Intersection.

(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 ; prolap.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 ; Prolog Program.
12 ; append([], L, L).
13 ; append([H|T], L, [H|R]):- append(T, L, R).
14
15 ; Simplified Prolog Program.
16 ; appendP1, P2, P3):- unifyP1, P2, L, unifyP3, L.
17 ; appendP1, P2, P3):- unifyP1, [H|T], unifyP3, [H|R], append(T, P2, R).
18
19 ; The result of a mechanical transformation. No optimizations.
20
21 (require "unify")
22
23 [setfun p-app(p1 p2 p3)
24  [lets ([L ('new logvar))]
25  #s *(unify p1 () #s *(unify p2 1) #s *(unify p3 1) (return))]
26  [lets ([L ('new logvar)) (T ('new logvar)) (R ('new logvar)) (H ('new logvar))]
27  #s *(unify p1 (cons h t)) #s *(unify p3 (cons h r)) #s *(p-app t p2 r)
28 (return))]
29
30 (setfun (s x))
31
32 (setfun test-app())
33 (return [let ([L ('new logvar) ('new logvar)])
34  [list #*block *p-app_v1_v2 *'(1 2) [rstrip (list_v1_v2)]]])
35
36 ;-------------------------------------------------------------------------------
37 ; OR-parallel prolog style append functions.
38 ; When called as a generator returns all solutions.
39 ; Usually called with logical variables as arguments.
40 ;---------------------------------------------------------------------------------------------------------------------
41 ; Shows how or-parallel prolog style functions can be written in ALLOY.
42 ; To add a cut anywhere in the program, insert "*(unless *(cut) (fail)"
43 ; Where function cut is defined as "(setfun cut() (return))"
44 ;---------------------------------------------------------------------------------------------------------------------
45 ; Prolog Program.
46 ; append([], L, L).
47 ; append([H|T], L, [H|R]):- append(T, L, R).
48
49 ; Simplified Prolog Program.
50 ; appendP1, P2, P3):- unifyP1, P2, L, unifyP3, L.
51 ; appendP1, P2, P3):- unifyP1, [H|T], unifyP3, [H|R], append(T, P2, R).
52
53 ; Mechanical transformation. No optimizations.
21 ; Cut would mean that all branches of the predicate failed.
22 (require "prolog")
23 (setq p-app-or (clp1 lp2 lp3)
24   (let [[[p1 p2 p3] [copyloge (list lp1 lp2 lp3)]
25          ((1) (1vs 1))
26          (ast #(unify p1 ()) #(unify p2 1) #(unify p3 1)
27          *(return (list p1 p2 p3)))]
28       (let [[[p1 p2 p3] [copyloge (list lp1 lp2 lp3)]
29              ((1 t r h) (1vs 4))
30              (ast #(unify p1 (cons h t)) #(unify p3 (cons h x))
31                *(let s (((T2 L2 R2) *(p-app-or t p2 r))]
32                              (return (list (cons h T2) (12 (cons h R2)))))])])
33   *(setq test-app-or()
34       (return (let (((v1 v2) [list ('new logvar) ('new logvar)])
35                    [list #[print1 (rstrip *(p-app-or v1 v2 '(1 2))))]))))
36
clp-impor.
1 ;---------- ALLOY version 0.1 ----------- 10/10/89 -----------
2 ; proimsp.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 "(setq cut () (return))"
10 ;-----------------------------------------------------------------
11
12 ; Prolog Program.
13 ; member(X, [X | _]).
14 ; member(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 ; member(X, P2):- unify(P2, [X | _]).
20 ; memberp(X, _ | 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 [setq member (clp1 lp2)
30   [let s (((p1 p2) (copyloge (list lp1 lp2))))
31     (ast #(unify (cons p1 ('new logvar)) p2)
32       *(return (list p1 p2)))]
33   (let s (((x p2) (copyloge (list lp1 lp2))))
34     *(return (list p1 p2)))]
35   *(ast #(unify p2 (cons h x))
36      *(let s (((X2 R2) *(memberp x r)))]
37   *(setq test-app-or()
38       (return (let (((v1 v2) [list ('new logvar) ('new logvar)])
39                    [list #[print1 (rstrip *(p-app-or v1 v2 '(1 2))))]))))
40
file p-inter.s.a:
(return (list X2 (cons h r2))))

(setfun commop(1p1 1p2 1p3)
  [let ([(x 11 12) (copyloge (list 1p1 1p2 1p3))])
    ![let ([(X2 11_2) * (memberp x 11))]
    ![let ([(x3 12_2) * (memberp x2 12)])
    (return (list x3 11_2 12_2))])])

(setfun intersectionp(1p2 1p3)
  (return (let [((x) (1vs 1))]
    [list #*[rstrip *(commonp x 1p2 1p3)])]))

B.6 Miscellaneous
B.6.1 Dining Philosophers

;----------------------------------------------------------
; Benchmark set no 8.
; Philosophers

(load "aphil.a")

(setfun bench8()
  (dine 3 4))

file phil.a:

;----------------------------------------------------------
; N Philosophers.

4 (class-start philosopher
  (static id fork1 fork2)
  (import print nl gt diff println sum
   (common faa semaphore)
   (methods live))

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

14 (setfun live(n
  (loop n))

17 [setfun loop(n
18  ('p c-semaphore) [println (list "Philosopher" id "entered room.")]])
19  ('p fork1) [println (list "Philosopher" id " took left fork")]
20  ('p fork2) [println (list "Philosopher" id " took right fork")]
21 [println (list "Philosopher" id "started eating")]
22 (eat 2)
23 [println (list "Philosopher" id "stopped eating")]
24 [println (list "Philosopher" id " put down left fork") ('v fork1]
25 [println (list "Philosopher" id " put down right fork") ('v fork2]
26 ('v c-semaphore) [println (list "Philosopher" id "exited room.")]

(if (gt n 0) (loop (diff n 1)))]

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

(class-end philosopher)

-------------

; Dinner for p philosophers n times.

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

(setfun dinen2(p n)
  (let ((fork1 ('new semaphore 1)) create)
    (set create (mu (previous i)
      (if (gt i 1)
        (let ((f ('new semaphore 1))
          ('live ('new philosopher previous f) n)
          (create f (diff i 1)))
        ('live ('new philosopher previous fork1) n)))))
    (create fork1 p))])
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


